HYDRA/C.mmp

An Experimental Computer System

McGraw-Hill Computer Science Series

Allen: *Anatomy of LISP*
Bell and Newell: *Computer Structures: Readings and Examples*
Donovan: *Systems Programming*
Feigenbaum and Feldman: *Computers and Thought*
Gear: *Computer Organization and Programming*
Givone: *Introduction to Switching Circuit Theory*
Goodman and Hedetniemi: *Introduction to the Design and Analysis of Algorithms*
Hamacher, Vranesic, and Zaky: *Computer Organization*
Hamming: *Introduction to Applied Numerical Analysis*
Hayes: *Computer Architecture and Organization*
Hellerman: *Digital Computer System Principles*
Hellerman and Conroy: *Computer System Performance*
Kain: *Automata Theory: Machines and Languages*
Katzan: *Microprogramming Primer*
Kohavi: *Switching and Finite Automate Theory*
Liu: *Elements of Discrete Mathematics*
Liu: *Introduction to Combinatorial Mathematics*
MacEwen: *Introduction to Computer Systems: Using the PDP-11 and Pascal*
Madnick and Donovan: *Operating Systems*
Manna: *Mathematical Theory of Computation*
Newman and Sproull: *Principles of Interactive Computer Graphics*
Nilsson: *Problem-Solving Methods in Artificial Intelligence*
Rice: *Matrix Computations and Mathematical Software*
Rosen: *Programming Systems and Languages*
Salton: *Automatic Information Organization and Retrieval*

Stone: *Introduction to Computer Organization and Data Structures*
Stone and Siewiorek: *Introduction to Computer Organization and Data Structures: PDP-11 Edition*
Tonge and Feldman: *Computing: An Introduction to Procedures and Procedure-Followers*
Tremblay and Bunt: *An Introduction to Computer Science: An Algorithmic Approach*
Tremblay and Manohar: *Discrete Mathematical Structures with Applications to Computer Science*
Tremblay and Sorenson: *An Introduction to Data Structures with Applications*
Tucker: *Programming Languages*
Watson: *Timesharing System Design Concepts*
Wiederhold: *Database Design*
Winston: *The Psychology of Computer Vision*

McGraw-Hill Advanced Computer Science Series

Davis and Lenat: *Knowledged-Based Systems in Artificial Intelligence*
Feigenbaum and Feldman: *Computers and Thought*
Kogge: *The Architecture of Pipelined Computers*
Lindsay, Buchanan, Feigenbaum, and Lederberg: *Applications of Artificial Intelligence for Organic Chemistry: The Dendral Project*
Nilsson: *Problem-Solving Methods in Artificial Intelligence*
Watson: *Timesharing System Design Concepts*
Winston: *The Psychology of Computer Vision*
Wulf, Levin, and Harbison: *Hydra/C.mmp: An Experimental Computer System*

HYDRA/C.mmp

An Experimental Computer System

William A. Wulf
Carnegie-Mellon University

Roy Levin
Xerox Palo Alto Research Center

Samuel P. Harbison
Carnegie-Mellon University

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland Bogotá Hamburg
Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

This book was set in Times Roman by the authors. The editors were
Diane D. Heiberg and Madelaine Eichberg; the production supervisor was
Dominick Petrellese. The drawings were done by J & R Services, Inc.
Fairfield Graphics was printer and binder.

HYDRA/C.mmp
An Experimental Computer System

Copyright © 1981 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

1 2 3 4 5 6 7 8 9 0 FGFG 8 9 8 7 6 5 4 3 2 1 0

Library of Congress Cataloging in Publication Data

Wulf, William Allan.
HYDRA/C.mmp, an experimental computer system.

(McGraw-Hill advanced computer science series)
Bibliography: p.
Includes index.
1. HYDRA/C.mmp (Computer system) I. Levin, Roy, joint author. II. Harbison, Samuel P., joint author. III. Title. IV. Series.
QA76.6.W 84 001.64 80-18424
ISBN 0-07-072120-3

To designers and builders
of real programming systems

CONTENTS

Preface xv

I Background and Hardware 1

1 Introduction 3

2 C.mmp 9

2-1 Structure 9
2-1.1 Processors 10
2-1.2 Shared Memory and Address Translation 14
2-1.3 The Interprocessor Bus 16
2-2 The Actual C.mmp Configuration 17
2-3 Implementation Features 19
2-3.1 The Crosspoint Switch 19
2-3.2 Processor Modifications 22
2-3.3 Extensions for Error Detection 23
2-3.4 Caches 23
2-3.5 The Interprocessor Bus 24
2-3.6 Peripherals 24
2-4 Technology and Costs 25
2-5 Hardware Performance 25
2-6 Retrospective 27

II The System Design 29

3 The Hydra Philosophy 31

3-1 Achieving the Goal 32
3-2 Protection 34
3-3 Policy/Mechanism Separation 35
3-4 An Aside on Data Abstraction 37
3-5 Types and Resources 38
3-6 Parallelism 40
3-7 Summary of the goal 41
3-8 Further Reading 41

4 Fundamental Concepts 43

4-1 Objects 44
4-2 Types 45
4-3 Capabilities 46
4-4 Representation of Objects 47
4-5 The Local Name Space 49
4-6 Procedures 51
4-7 Processes 52
4-8 Procedures and Access Rights 52
4-9 Templates and the Merge Operation 53
4-10 The Call Mechanism and Rights Checking 55
4-11 A Note on Implementation 55
4-12 Protection vs. Flexibility 58
4-13 Retrospective 58
4-14 Further Readings 60

5 Kernel Facilities 63

5-1 Notation 63
5-2 Kernel Rights 65
5-3 Kernel Operations 68
5-3.1 Informational Kalls 70
5-3.2 Generic Kalls 71
5-3.3 Kalls for Creating Objects 73
5-3.4 The Call Mechanism 74
5-3.5 GST Kalls 77
5-4 Kernel Subsystems 78
5-5 A Complete Example 79
5-5.1 The Programming Environment 80
5-5.2 Programming Subsystems 80
5-5.3 CreateBoxSubsystem 82
5-5.4 Deposit 85
5-5.5 Withdraw 86
5-5.6 Using the Subsystem 86
5-5.7 Some Design and Implementation Issues 86
5-6 Retrospective 88
5-7 Further Readings 89

6 The Message System 91

6-1 Overview of the Message System 91
6-2 An Example: Data Base Management 93
6-3 Ports 95
6-4 Connections 96
6-5 Messages 97
6-6 Operations on Messages 98

6-7 A View of the Reply Mechanism 101
6-8 Retrospective 102
6-9 Further Readings 104

III The System in Use 105

7 Using the Protection Mechanisms 107

7-1 Kernel Rights 108
7-2 The Mutual Suspicion Problem 109
7-3 The Modification Problem 110
7-4 The Conservation Problem 112
7-5 The Confinement Problem 113
7-6 The Initialization Problem 115
7-7 Retrospective 116
7-8 Further Readings 117

8 A File System 119

8-1 Files and Subfiles 120
8-2 Operations on Files 122
8-2.1 File Operations 122
8-2.2 Subfile Operations 123
8-2.3 File I/O 123
8-3 Implemented Subfiles 124
8-4 Protection 124
8-5 Retrospective 126

9 A Network Control Program 129

9-1 The Problem 129
9-2 The Hydra NCP 131
9-2.1 IMP-Host Communication 131
9-2.2 Host-Host Communication 133
9-2.3 Connection Management 134
9-3 Retrospective 135

10 A User-level Operating System 139

10-1 Anatomy of the User-Level Operating System 141
10-1.1 Connecting to the System 141
10-1.2 Logging In 142
10-1.3 The Command Language 144
10-1.4 Logging out 144
10-1.5 Subsystem interactions 145
10-2 The Job System 146
10-3 Reliability Mechanisms 147
10-4 Other Subsystems 148
10-4.1 Directory and Catalogue 148

10-4.2 Device Allocation System 148
10-4.3 Fork 148
10-4.4 Commands 149
10-4.5 SYSMON 149
10-5 Retrospective 149

IV The System Implementation 151

11 The Object Store 153

11-1 A Virtual Memory System 153
11-1.1 The Representation of Objects and Capabilities 154
11-1.2 Mapping Capabilities to Objects 156
11-2 Storage Management in the GST 157
11-2.1 Active GST Maintenance 157
11-2.2 Passive GST Maintenance 158
11-3 Mechanisms for Reliability 159
11-4 Retrospective 160

12 Scheduling and Synchronization 163

12-1 Scheduling Parameters 164
12-2 Process and Policy Objects 167
12-3 Synchronization Mechanisms 171
12-4 Implementation 174
12-5 Retrospective 184

13 Paging 187

13-1 The User's View of Paging 188
13-2 The Working Set and Scheduling 190
13-3 Implementation 191
13-3.1 Page Replacement Policy 193
13-4 Retrospective 194

14 Input/Output 195

14-1 The Hardware Environment 195
14-2 The User's View of I/O 196
14-3 Implementation 198
14-3.1 Interprocessor interrupts 200
14-3.2 DMA Transfers 200
14-3.3 Error Recovery 201
14-4 Kernel I/O 202
14-5 Retrospective 203

15 Error Recovery 207

15-1 Validation Mechanisms 208
15-2 Fault-Tolerant Mechanisms 208

15-3 Detection Mechanisms 209
15-4 Error Diagnosis 210
15-5 Recovery Mechanisms 211
15-6 Autorestart 212
15-7 Retrospective 213

V Measurements and Evaluation 217

16 Experimental Measurements 219

16-1 Performance Measurement Tools 220
16-1.1 The Hardware Monitor 221
16-1.2 The Kernel Tracer 223
16-1.3 The Snapshot Taker 228
16-1.4 Hercules: The Script Driver 229
16-2 Experiments and Results 230
16-2.1 Oleinick's Rootfinder Experiment 230
16-2.2 Baudet's Relaxation Experiment 234
16-2.3 Oleinick's HARPY Experiment 237
16-2.4 Marathe's Memory Interference Experiment 239
16-2.5 McGehearty's Memory Contention Experiment 241
16-2.6 Marathe's Lock Contention Experiment 243
16-2.7 Jain's Semaphore and Port Experiment 245
16-2.8 Marathe's Small-Address Effect Experiment 253
16-2.9 The Small Address Effect on HARPY 255
16-2.10 McGehearty's Kall Measurements 255
16-2.11 Size of the Hydra Kernel 260
16-2.12 McGehearty's "Stretch Factor" Experiment 262
16-2.13 Almes' Study of the Active GST 263
16-2.14 Almes' Study of the Passive GST 268
16-3 Retrospective 271

17 Reflections 275

17-1 Reflection on the Reflections 281

References 283

Index 289

PREFACE

The authors are three of the designer-implementors of Hydra/C.mmp, a unique computing system. C.mmp is a multiprocessor composed of 16 minicomputers and a large shared memory; Hydra is the kernel of its operating system.

Together, C.mmp and Hydra have demonstrated that multiprocessors can be extremely cost effective; Hydra/C.mmp does not suffer from most of the performance problems experienced by some earlier multiprocessor systems. They have also demonstrated the power and flexibility of what is possibly the most sophisticated protection facility ever implemented. Finally, they have provided a vehicle for exploring algorithms and program structures that exploit asynchronous parallel processing. This book is a detailed examination of this ambitious system—its structure, its facilities, its usability, and its performance.

Hydra/C.mmp, like all large systems, was the result of an enormous number of interrelated decisions. Sometimes we knew the design alternatives and could evaluate them. Sometimes we knew the alternatives, but were frustrated by a lack of data on which to base an objective choice. More often than one might suspect, we defaulted decisions because we did not even appreciate that alternatives existed.

The result is a computing system that works well. It's a system that we are extremely proud of, even though it is far from perfect. We and our colleagues have invested a great deal of effort in using it, in measuring it, and in analyzing its performance—and we have discovered many of its faults. In his book *The Mythical Man-Month* [Bro75], Fred Brooks advises building a system twice, and throwing the first version away. If we had the opportunity of doing that, we could now correct many of the system's faults. Like many other designers, however, we did not have that luxury. Instead, we are using this book to disseminate our experience to other designers and implementors of computing systems. We believe these people will have some of the same goals and will face some of the same problems that we did. Thus, we have two objectives:

- We want to describe the system from both the user's and the implementor's perspective, detailing the esthetic goals and pragmatic choices which led us to the final structure.
- We want to provide both objective data and our own subjective evaluations so that our readers may analyze the consequences of our decisions and appreciate the strengths and weaknesses of the design.

We have tried to organize this book to complement these objectives. First, we must describe the system, and thus our design decisions. There are three aspects to this: background information, the external model presented by the system (the "user's view"), and the internal organization (the "implementor's view"). Second, we must provide information on which the reader may base an evaluation. There are also three aspects to this: the usability of the external model, the performance of the implementation, and our subjective impressions of what we did right and what we did wrong. With one exception, we have tried to isolate each of these kinds of information into separate chapters; the exception is that most descriptive chapters conclude with a "Retrospective" section that captures our subjective reactions. Otherwise, the organization is as follows:

Part I (Chapters 1 and 2) provides background information. It includes a brief history of the project and a description of the C.mmp hardware design.

Part II (Chapters 3 to 6) describes the external, user-visible model provided by Hydra. It begins by stating and amplifying a major goal of Hydra—to allow operating system facilities to be defined by users. It then goes on to show how this goal is achieved by an extensible, capability-based protection mechanism.

Part III (Chapters 7 to 10) illustrates the use of the facilities provided by Hydra. We really have two objectives for these chapters. One is to describe some specific, interesting problems and facilities. The other is to give the reader a "feel" for what it's like to build such facilities and thus, indirectly, to provide information on which to base a judgment of the usability of the external model.

Part IV (Chapters 11 to 15) describes the implementation of some of the major components of the Hydra kernel. Again, our objective is to be descriptive and thus to record our design and implementation decisions.

Part V (Chapters 16 and 17) returns to evaluation. Chapter 16 summarizes the results of a number of studies of the performance of the system. Chapter 17 collects the major reflections from the previous chapters.

In our descriptions, we have not tried to be exhaustive. We have not, for example, defined all the operations provided by the Hydra kernel. Neither have we tried to describe all the data structures and algorithms used throughout the implementation. Instead, we have attempted to present

enough detail so that the reader can project at least one plausible implementation from the information presented. Our evaluation material is similarly not exhaustive; it is intended to provide the intuitive "gist" of some actual experiments and their results. The complete details can be found in the cited papers, theses, and reports.

Hydra/C.mmp, like most research efforts, contains some innovative ideas and some re-engineering of existing ones. Even innovative notions, however, are generally derived from previous work in related areas. It is impossible for us to acknowledge specifically the source of every technical idea used or exploited in Hydra. Rather, in those chapters in which we believe our work represents a significant innovation (primarily in Part II), we have appended a brief survey of related papers from which our work derives or with which it may be contrasted. In chapters in which we describe Hydra facilities or components that are largely manifestations of the "common wisdom," we have simply cited major or representative works in the relevant areas.

Acknowledgements

Constructing the hardware and software of a major computing system requires the dedicated effort of many people. The authors have had the privilege of recording these efforts, but the people who made it possible are:

Ellis Cohen, Bill Corwin, David Jefferson, Tom Lane, and Fred Pollack—who were our comrades in the design and implementation of Hydra.

Anita Jones—who provided the insight that led to the Hydra protection structure.

Bill Broadley, Chuck Pierson, and Jim Teter—who were largely responsible for the detailed hardware design and implementation.

Gordon Bell—who provided some of the initial hardware design concepts, and was always a goad.

Hank Mashburn and Joe Newcomer—who joined the Hydra team later, but without whom the system would not have been finished, and would be neither as robust nor as polished.

Guy Almes, Rick Gumpertz, Pat McGehearty, George Robertson, Peter Schwarz, and Mary Thompson—who provided user-level subsystems that made Hydra/C.mmp into a usable system.

Gerard Baudet, Navindra Jain, Madhav Marathe, Don McCracken, and Peter Oleinick—who provided performance analyses of the system.

Bruce Leverett—who maintained and improved the Bliss/11 compiler for us.

The Defense Advanced Research Projects Agency and the Office of Naval Research—who provided the funding, and especially Steve Crocker and Bill Carlson, our DARPA program directors, whose critiques were incisive and challenging.

L*, Production Systems, ZOG, SUS, IUS, STAROS, Algol68, and the FSAS

Benchmark Validation—projects whose sympathetic members struggled to use the system while it was still maturing.

Many others, but especially: Jerry Apperson, David Babcock, D. P. Bhandakar, Kitty Fischer, Sam Fuller, Wayne Gramlich, Peter Hibbard, Paul Hilfinger, Andy Hisgen, Steve Hobbs, Richard Johnsson, Dan Klein, Paul Knueven, Phil Karlton, David Lamb, John McCredie, Roger Needham, Brian Reid, Andy Reiner, Bob Schwanke, Bill Strecker, Rick Snodgrass, Leland Szewerenko, and Chuck Weinstock. Among these are application programmers, system programmers, performance evaluators, proofreaders, users, critics, and above all, friends.

Finally, we would like to acknowledge our debt to the Carnegie-Mellon University Computer Science Department, one of the few places which permits, and indeed encourages, the designing and building of realistic experimental systems.

We don't know how to thank these friends enough. Mere mention here seems inadequate for those with whom we have shared the predawn light, hot on the trail of an elusive bug.

Wm. A. Wulf
Roy Levin
Samuel P. Harbison

PART ONE

BACKGROUND AND HARDWARE

CHAPTER

ONE

INTRODUCTION

This monograph discusses Hydra/C.mmp, an experimental computer system built by the authors and their colleagues at Carnegie-Mellon University between 1971 and 1977. C.mmp is a multiprocessor computer, consisting of up to 16 minicomputers and 32 megabytes of shared memory connected by a central processor/memory switch. Hydra is the kernel of the operating system for C.mmp.

The Hydra/C.mmp project began, slowly, in 1970. It grew out of a study at Carnegie-Mellon University, sponsored by DARPA,[1] into suitable computers for future research in artificial intelligence. The recommendation of that study was a multiprocessor, called C.ai [Cai72]. Even though this machine was never built, the study served to emphasize the potential benefits of interconnected, small, inexpensive computers. It also emphasized how little had been published on the design and performance of this kind of machine.

The potential advantages of multiple computer structures were obvious in 1970; they included improved cost/performance, greater absolute performance, incremental expansability, and improved reliability and availability. The single-chip microprocessor was also on the horizon; although there were clearly many single-processor applications for these machines, it seemed possible that there would also be great advantages to multiple microprocessor systems.

It was also obvious in 1970 that these were only *potential* advantages. Whether or not such systems would *actually* be more cost effective, etc., was an open question. A number of interconnected computer structures had been built—in fact, the idea was an old one. There were examples of both loosely coupled systems (e.g., the IBM ASP) and tightly coupled multiprocessors (e.g., the Burroughs D825), but there wasn't a great deal of data in the literature about the performance of these systems. The stories that one heard were quite discouraging, but it was difficult to determine what aspect of the system design or application programs were responsible for the problems.

It was in this climate that we undertook to study interconnected computer structures, and multiprocessors in particular. We wanted to understand

[1]The Defense Advanced Research Projects Agency, DOD

what kinds of interconnection structures, what kinds of software structures, what kinds of user facilities, and what kinds of programming languages and algorithms would achieve the advantages these systems offered. We realized that this research would be difficult in several ways. First, we suspected that no single design would simultaneously achieve all the advantages. Second, we knew that we would have to build, use, and instrument fairly large, realistic systems, and measure them running realistic applications in order for our conclusions to be meaningful. Finally, we knew that we would have to build more than one system in order to explore and contrast the alternatives.

These realizations determined a research strategy that we have pursued for nearly a decade. That strategy has been to build a number of multiprocessor systems, each exploring a distinct point in the design space. We began with a tightly coupled multiprocessor, C.mmp, in 1971. In 1975, with the bulk of the Hydra/C.mmp development behind us, another group at C-MU undertook to build Cm*, [Swa77], a more loosely coupled multiprocessor. As of this writing, a third group is beginning an even more distributed system. With each system, augmented by numerous efforts elsewhere, we get closer to our goal of understanding the available alternatives.

This book marks the endpoint of our first major experiment. It is not, in itself, a complete exploration of all the alternatives in the design space. On the contrary, it describes only one particular point in that space—an attempt to optimize the usability and performance of a tightly coupled multiprocessor. This book attempts to describe and evaluate the system in sufficient detail that its properties can be related and compared to those of other systems. The actual task of making those relations and comparisons, however, we leave to our readers.

There are, of course, two distinct aspects in any computing system, including Hydra/C.mmp—the hardware and the software. Each of these poses a set of research issues that has to be addressed relative to the general goal of understanding multiprocessor systems. The hardware issues are constrained by the tightly coupled nature of C.mmp and primarily revolve around avoiding contention for access to the primary memory. The issues of incremental expansion and reliability were considered secondary, since the nature of the processor-memory switch places an *a priori* bound on the number of processors in the system and constitutes, in principle, a reliability bottleneck.[2]

The software issues surrounding the C.mmp hardware are more complex than those for the hardware for two reasons: (1) there is a much larger set of possible design choices for the software, and (2) we must be concerned with "usability," a subjective issue, as well as performance. Some of the software design decisions are related to the fact that C.mmp is a multiprocessor; many are not. A whole class of decisions, for example, follow from using a

[2]Interestingly, however, the processor-memory switch proved to be the least of C.mmp's reliability problems, as will be seen in Chapters 2 and 15.

particular minicomputer, the PDP-11, as the system's processing element. A more consequential class of decisions, however, followed from a chain of reasoning that went something like this:

> We want to learn about the consequences of different designs on the usability and performance of multiprocessors. Unfortunately, each decision we make precludes us from exploring its alternatives. This is unfortunate, but probably inevitable for the hardware. Perhaps, however, it is not inevitable for the software, and especially for the facilities provided by the operating system.
>
> Suppose that we build only the "kernel" of an operating system and allow most operating system facilities to be built as user programs—then it would be easy to build and experiment with different kinds of facilities. We would learn more this way and would not lock our users into a single model of how to use C.mmp; two users, for example, could use completely different file systems—each tuned to the special needs of that user.
>
> We also want a "general purpose," "multi-access," "time-sharing" system; only such a system will allow several experimenters to be developing applications for C.mmp simultaneously. The advantages of the user-level definition of operating system facilities would be lost if all users had to use the same version of a facility just because they happen to be running at the same moment. Therefore, it must be possible for each user to have a private and independent version of the facilities.
>
> To satisfy these objectives, we are going to need a pretty clever protection structure. It's going to have to have at least two properties:
>
> - It will have to be strong enough to allow sensitive facilities, such as the file system, to be defined and protected by user-level programs.
> - It will have to be extensible, so that new kinds of facilities can be created dynamically and still be covered by the protection system. Simple hierarchical protection, such as that in Multics [Sal74], will be inappropriate for this system; one cannot say whether one facility is "more privileged" than another when they are dynamically and independently created.
>
> Obviously, this protection will have to be enforced by the kernel; we can't trust that to user-level programs.
>
> The capability model originally defined by Dennis and Van Horn [Den66], modified a bit to allow for dynamic extension, does what we need. And, by the way, there are some interesting protection problems that we can also solve if we use capabilities.

The result was Hydra. Initially motivated by a desire to maximize the information to be gained from the C.mmp experiment, we were led by this chain of reasoning to a second and almost independent research goal: the user-level definition of operating system facilities.

Hydra's first goal, of course, was simply to be a good, general-purpose, multiprocessor operating system—one which could support many different users who wished to take advantage of the multiprocessor nature of C.mmp without incurring undue overhead. (To make the system maximally available for experimentation, we also felt that it should be a time-sharing system.) It was not obvious in the beginning that all this could be done; other multiprocessor systems had experienced debilitating software overheads and contention. Thus the research issues derived from this goal were:

1. Can we devise a set of facilities with which users can easily develop and

measure multiprocessing programs?
2. Can we devise a system structure free of (serious) software overheads and contention?
3. Can we devise resource allocation and scheduling policies that work well in a multiprocessor environment?

The second goal of Hydra was to permit essentially all the facilities one normally associates with an operating system to be defined by user-level programs—programs without special privileges. Moreover, we wanted to allow an arbitrary number of such definitions to exist (and be used) simultaneously. Except for the initial line of reasoning sketched above, this goal has nothing to do with multiprocessors. Yet, if one must rank them, we came to believe that this was the more important goal. The research issues revolving around it were:

1. What semantic model for the kernel will allow for user extension?
2. How does one provide protection in a system where even such fundamental facilities as files, catalogues, and schedulers are provided by (unprivileged) user programs?
3. What other protection problems can be solved in our model, and at what cost?
4. How can all this be implemented efficiently?

Roughly midway through the development of Hydra, a third research goal emerged—reliability. Because of the structure of C.mmp, we initially believed that the overall system reliability could be expected to be similar to that of other systems of comparable (total) size and performance. This led us to an initial design of the Hydra kernel that attended to reliability in a manner similar to other (uniprocessor) systems. A major lesson from C.mmp, however, has been that multiprocessors present special reliability problems as well as special opportunities for solving them. When we finally learned this lesson, reliability became an explicit goal.

It would be nice to be able to assert that the system emerged full-blown from this context and those goals. In practice, however, a system of the size and complexity of Hydra/C.mmp evolves in a more or less controlled way over time. The final system, described in the following chapters, existed in a relatively stable form during 1977-79. To appreciate how it came to have its final form, one must look at its development history.

The study of C.ai in 1970 provoked discussions of both the overall research strategy and the possibility of constructing a multiprocessor. These discussions led, in 1971, to fairly specific proposals for the structure of C.mmp, including the architecture of the crosspoint switch, the interprocessor communication facility, and the address mapping hardware. Also during this period, a number of analytic studies were conducted to determine the potential memory contention; these studies resulted in the choice of the

16 × 16 configuration and the relative processor and memory speeds.

Also during 1971, an informal group met to discuss the requirements of an operating system for the machine. It was during these meetings that Anita Jones proposed the type-extension addition to capability-based protection that became the core of Hydra's top-level design. In the fall of 1971, an internal memo was circulated sketching the design in a form quite similar to the final result.

During 1972, hardware and software development proceeded in parallel. Because the principal hardware project would be the construction of the processor-memory switch, the switch design was tested in three prototypes. The first version was a 1 × 1 switch; that is, it was capable of connecting only one processor to one memory module. The second and third prototypes were 2 × 2 and 4 × 4 versions, respectively. The 2 × 2 switch functioned in a test environment by the middle of 1972 and was debugged with real processors and memories by the end of that year.

Software development began, using a "simulator" developed on the PDP-10. This simulator did not attempt to emulate the instruction set of the PDP-11—it merely allowed programs written in a dialect of Bliss/11 (the implementation language for Hydra) to be run on what appeared to be a multiprocessing PDP-10. Although its capabilities were limited, the simulator allowed the first portions of Hydra to be debugged before the C.mmp hardware was available. Specifically, two components of the Hydra kernel were actively tested—KMPS (the low-level scheduler) and the GST (the object/capability system).

In early 1973 the 2 × 2 switch, with two processors and one memory, became available to the software group. During a two-week period the Hydra kernel was moved from the PDP-10 simulator and real multiprocessing began. By May 1973 the 4 × 4 switch was operational, and the Hydra kernel was executing on a prototype system with two processors and three memory ports. At this point, the implementation team had expanded to about nine people, including programmers working on user-level subsystems that were to run "on top of" Hydra and provide the first traditional operating system facilities: a command language, a scheduler, and a directory system.

In the next year, the 16 × 16 crosspoint switch became functional (but not fully populated) and non-kernel software began running on a routine basis. In the spring of 1975, virtually all the kernel facilities were complete and a 6-processor, 8-memory hardware configuration was in daily use. By 1977, the 16 × 16 switch was fully populated with processors and memories; at this point we had 11 PDP-11/40 processors, 5 PDP-11/20 processors, and about 2.5 megabytes of primary memory. Most of the eventual I/O devices were also available at about this time; it is difficult to pinpoint the exact dates at which various devices became operational, but the final system included seven special paging disks, seven moving head disks, a magnetic tape drive, several DECtapes, a line printer, an interface to the ARPANET, several

special real-time analog input devices, and a connection to a "front end" terminal server.

The software was also essentially complete by 1977; most of the user facilities that were eventually constructed were available for use. Again, it is difficult to pinpoint precisely when many of these facilities became available, but they ultimately consisted of:

Two Policy Modules (the user-level schedulers)
A file system
Two text editors
Two catalogue (directory) systems
A user-level debugger
Language processors for Algol '68, Fortran, C, and L* (a list-oriented system building language)
Two command languages
A large number of utilities for logging in and out, spooling printed output, managing multiple terminals, allocating I/O devices, creating subsystems, and so on

In 1978 we removed the five PDP-11/20 processors from the system—they were the oldest processors and had become unreliable. The remaining 11 PDP-11/40 processors had been extended with a writable control memory, and special instructions had been implemented to improve system performance. To take full advantage of this, Hydra needed all the processors to execute identical instruction sets.

Since 1976-77, the major efforts on Hydra/C.mmp have been to improve its reliability and performance. Reliability enhancements included adding detection and recovery strategies to cover hardware malfunctions. Performance improvements were guided by several performance evaluation studies (reported in Chapter 16) which led to a number of enhancements in the kernel—notably better paging and storage allocation algorithms. Only one functional improvement was made in the system—the addition of a parallel garbage collector for removing objects that were no longer referenced.

Throughout the 1974-1979 period there was a continuing effort to develop applications programs and measure their performance. These efforts produced three kinds of results: new algorithms especially suited to asynchronous multiprocessors, performance improvements in Hydra, and ideas about suitable programming language features for expressing these algorithms.

In March 1980 C.mmp was removed from operational service to make room for new projects.

CHAPTER

TWO

C.MMP

C.mmp[1] is a simple multiprocessor; it consists of a number of equal, asynchronous central processors that share a large primary memory. C.mmp differs from earlier multiprocessors such as the Burroughs D825, the IBM 360/67, and the Honeywell 645 (Multics) in two essential respects:

1. C.mmp is designed to have up to 16 processors while the other computers usually had no more than 4 processors.
2. C.mmp is constructed from minicomputers rather than the larger (32 to 48 bits/word) processors used in the other systems.

The effective use of C.mmp requires that we find and exploit a much higher degree of parallelism than was needed by earlier multiprocessors. In the past few years, the number of existing multiprocessors has increased significantly to include BBN's Pluribus [Orn75, Kat78] and C-MU's Cm* [Swa77] systems. However, C.mmp still remains notable for its uniform structure and support of a general-purpose operating system.

2-1 STRUCTURE

A block diagram of C.mmp is shown in Figure 2-1. At a gross functional level, C.mmp consists of three parts:

1. *Processors*, which are modified PDP-11/40E minicomputers, each with its own UNIBUS and various peripheral devices[2]
2. *Shared memory*, including the actual memory modules, a 16 · 16 crosspoint switch, and address relocation hardware on each processor
3. The *Interprocessor Bus*, which provides interprocessor communication

We will consider each of these parts in more detail.

[1]"C.mmp" stands for "multi-mini-processor computer"; we pronounce it "See-dot-em-em-pee." This chapter is a slightly revised version of [Ful78].

[2]PDP and UNIBUS are trademarks of Digital Equipment Corporation.

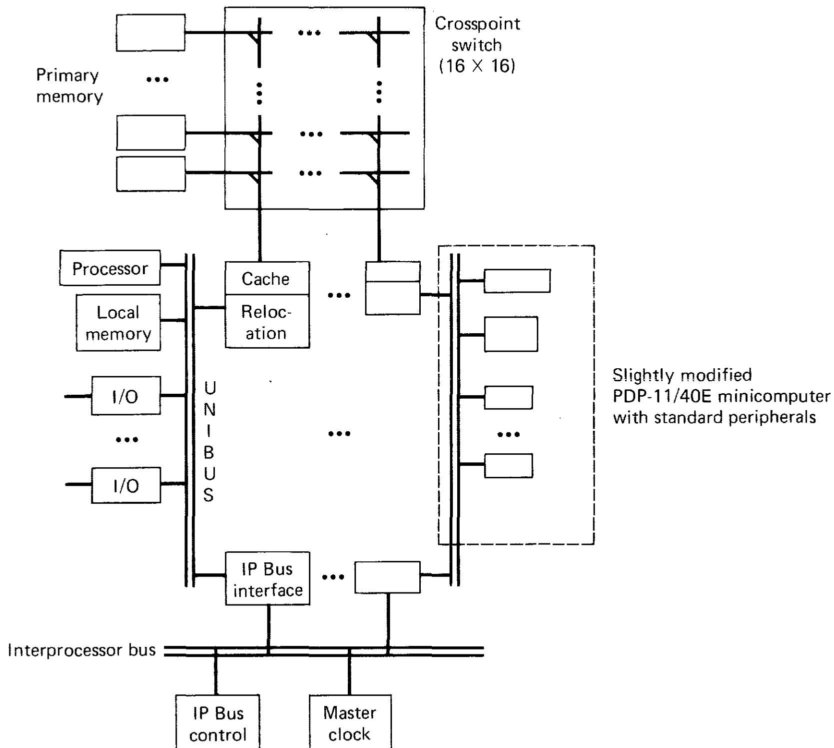

Figure 2-1 Block diagram of C.mmp

2-1.1 Processors

PDP-11 minicomputers, manufactured by the Digital Equipment Corporation, are the processing elements of C.mmp. PDP-11/20 models were originally used, and were later replaced by PDP-11/40Es. The PDP-11 has become sufficiently ubiquitous that a detailed explanation of its architecture is unnecessary [DEC73].

The PDP-11/40E minicomputer differs from the standard PDP-11/40 in having a 1K-word (80 bits/word) writable microstore. This feature was not strictly required by Hydra, but we expected to achieve significantly better performance by implementing frequently executed functions in microcode. The processors on C.mmp are further modified to provide special features for protection and addressing, including additional address spaces, instruction protection, and stack protection.

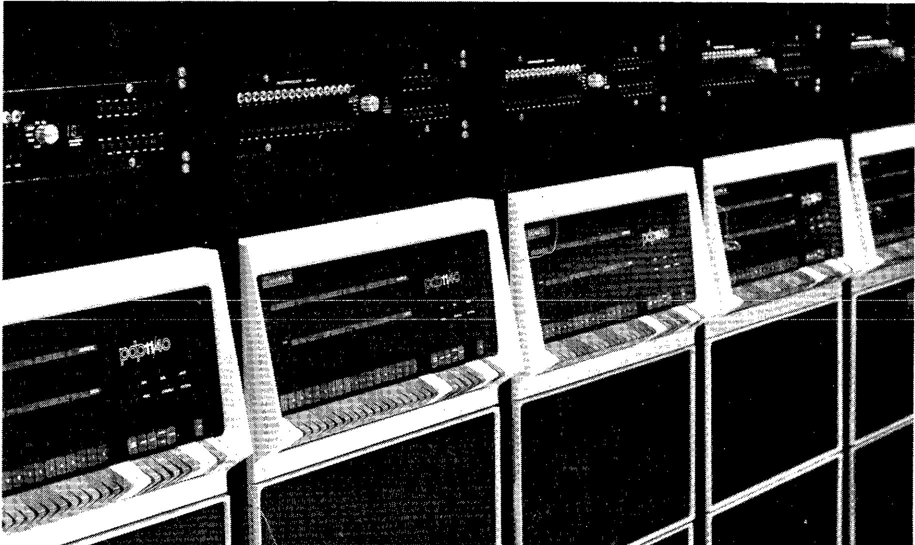

Figure 2-2 C.mmp processors

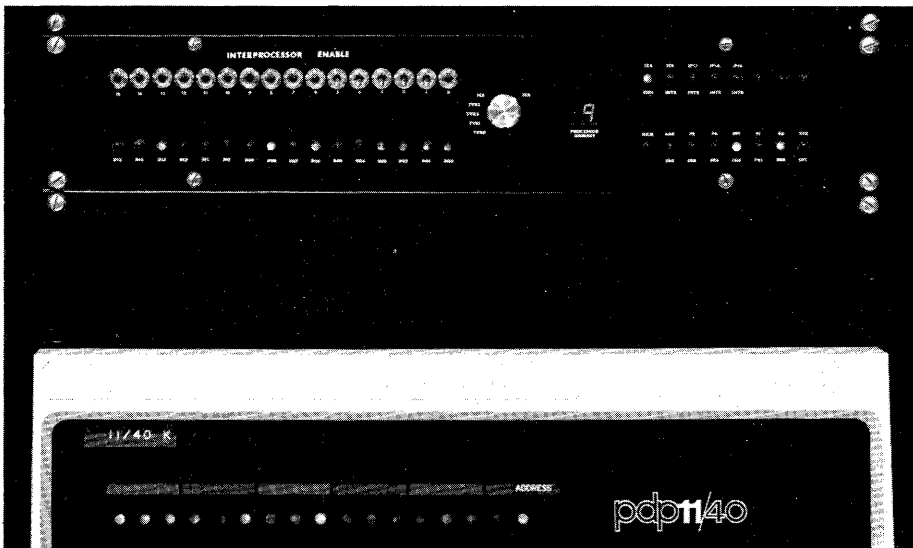

Figure 2-3 PDP-11/40E and interprocessor bus interface panel

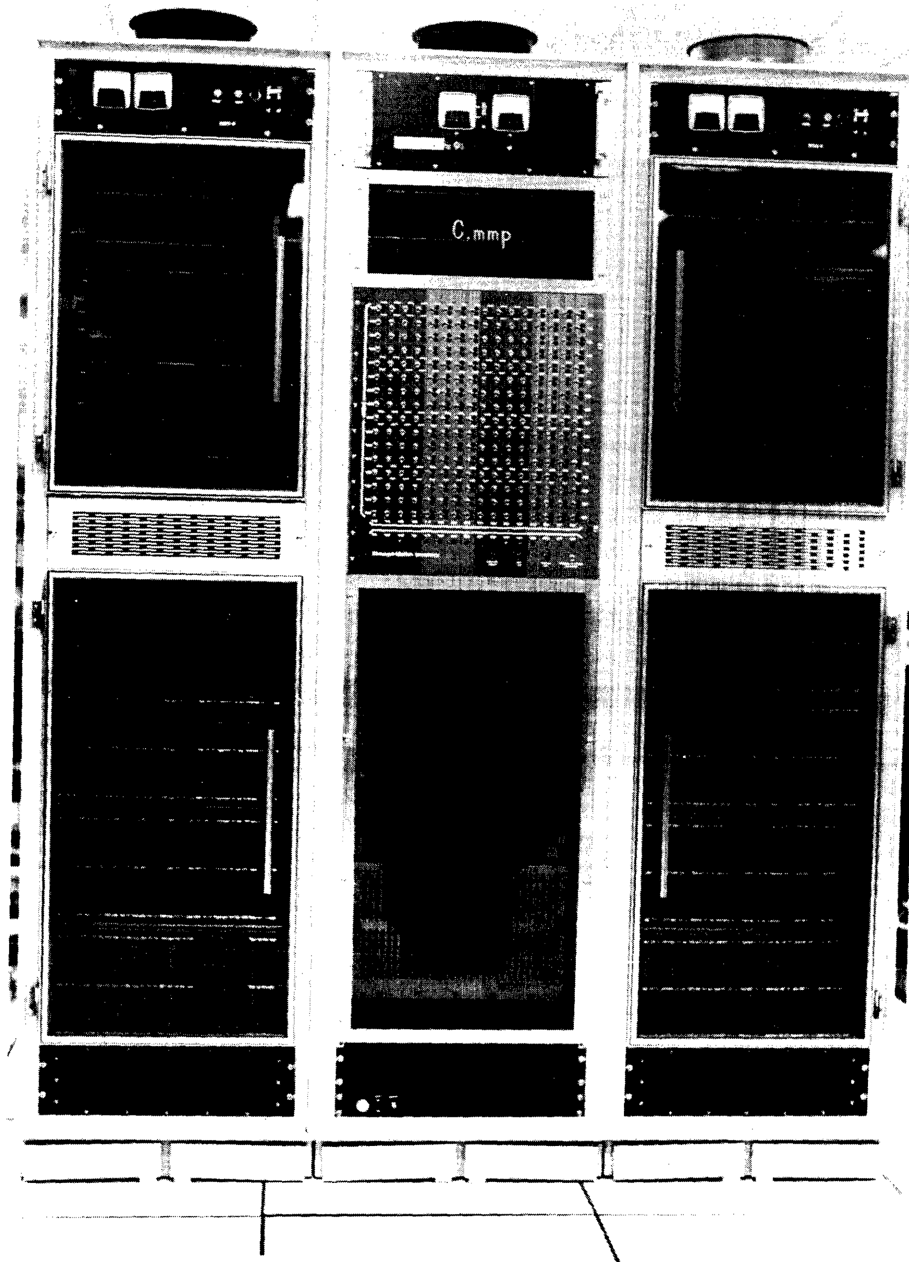

Figure 2-4 Crosspoint switch and primary memory

Address spaces A PDP-11 program can generate only a 16-bit address, but the UNIBUS supports an 18-bit address. We therefore implemented two "space bits" in the processor's Program Status (PS) register; the value of these bits is concatinated to the high-order end of the user's 16 bit address to form the 18 bit UNIBUS address. The setting of the space bits determines in which *address space* we are executing. Hydra defines '11'-space to be *kernel space*; '00'-space is *user space*; '10'-space is used for mapping addresses during I/O; and '01'-space is used for special applications.

All protection at the hardware level is controlled by the address spaces, so we were careful to ensure that user programs could not change the PS space bits. There are three possible ways to alter the PS, and we have secured each one:

1. A hardware interrupt causes new values for the PS and PC (Program Counter) registers to be fetched from fixed addresses. We modified the processors so that these fixed addresses are located in kernel space, where they are unreachable by users. (The same solution was applied to the software trap instructions, TRAP, EMT, BKT, and IOT.)
2. "Return from interrupt or trap" instructions, RTI and RTT, fetch new PS and PC values from the top of the stack. Except when we are executing in kernel space, we force these instructions to trap to the operating system, which will simulate the instructions after verifying that the PS and PC are "safe."
3. The PS register is itself addressable and hence may be written like any memory word. However, its address is in kernel space, and so is protected.

Instruction set modifications The HALT, WAIT, RESET, BKT, RTT, and RTI instructions were made illegal when executing in any but kernel space. They cause a trap to Hydra, which will reflect the error in a standard way to the user (in the first three cases) or will validate and simulate the instruction (in the case of BKT, RTI and RTT).

Stack protection The PDP-11 has several addressing modes which facilitate managing a stack, and both programming and hardware convention dictate the use of a standard stack area for interrupt processing, subroutine calls, and parameter passing. This stack area is pointed to by PDP-11 register 6, which is also called the *stack pointer register*, or just *SP*.

The stack introduces some problems in switching address spaces, since the stacking of the old (PS,PC) at interrupt time occurs in the old (e.g., user) space while the unstacking by RTI or RTT occurs in the new (kernel) space. Taking the simplest solution for a PDP-11/20,[3] we decided to force all

[3]Although C.mmp was ultimately composed of PDP-11/40E computers, many early design decisions were made to deal with the PDP-11/20 processors available at the start of the project.

address spaces to use the same stack. We do this by establishing the convention that the low-order 8K bytes of each address space are to be used for the stack and by constructing the relocation registers so that the stack page register in each space must hold the same value. (The detailed operation of the relocation registers is discussed below.) Additional modifications force the SP register to be "well-behaved" when executing in user space: any attempt to store a value in this register which would not be a legal stack address is prohibited. Having the kernel and the user share the same stack makes changing address spaces easy and allows users to pass arguments to the kernel simply and efficiently.

A programmable *stack underflow register* is used by the operating system to prevent users from accessing data belonging to their callers or to the operating system. A fixed *stack limit* further restricts the stack and defines an area in the lower portion of the stack page which can be used for the communication of global information between the kernel and the user.[4]

2-1.2 Shared Memory and Address Translation

What is functionally thought of as "shared memory" is actually implemented in three pieces: a number of off-the-shelf memory modules, a central crosspoint switch, and individual address translation units on each processor.

The crosspoint switch directs single-word transfers between the memory subunits and the processors, and up to 16 simultaneous accesses to memory are possible if all 16 processors request words in different memory subunits. Each of the 256 processor/memory crosspoints can be enabled or disabled either manually (from a front panel) or under program control (by setting a flip-flop addressable from a UNIBUS). This allows either Hydra or human operators to remove a faulty processor or memory module or to partition the system into two smaller multiprocessors.

Probably the greatest problem in building a large computing system from minicomputers is their small address space [Wul78]. On C.mmp we must be able to address several million bytes of primary memory from processors which can generate only an 18-bit address. We have already discussed how the processors divide up the UNIBUS address into four spaces; for address translation, these spaces are further divided into 8K-byte segments called *pages*, 8 pages per space, or 32 pages for the total 18-bit UNIBUS address. Shared memory, with its 25-bit address, can therefore contain up to 4,096 pages. Address translation fundamentally consists of mapping the 32 (virtual) UNIBUS pages to the 4,096 (real) shared memory pages.

C.mmp's address translation mechanism is different from other PDP-11 memory management techniques in three ways:

[4]PDP-11 stacks grow downward, from high addresses to low ones. Therefore the stack underflow register contains the highest address in the current user's stack, and the stack limit value is a low address.

1. C.mmp maps from (virtual) UNIBUS addresses to (real) shared memory addresses, whereas other PDP-11s map from (virtual) processor addresses to (real) UNIBUS addresses.
2. C.mmp maps addresses generated by peripheral devices; other PDP-11s do not.
3. C.mmp has pages of fixed size, while other PDP-11s allow pages to be of variable size.

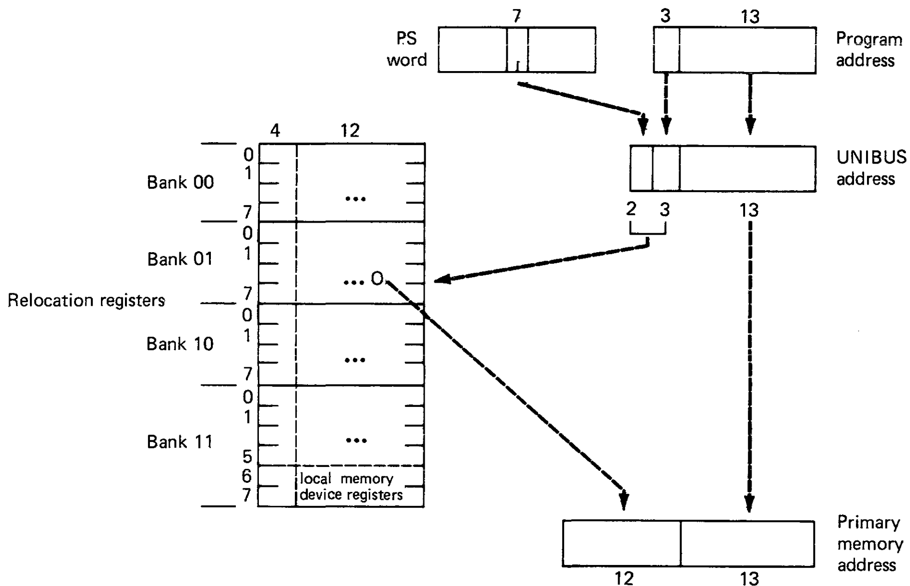

Figure 2-5 Address translation in C.mmp

The address translation process is shown in Figure 2-5.[5] Thirty of the 32 UNIBUS pages have associated *relocation registers*, whose format is shown in Figure 2-6. The low-order 13 bits of the UNIBUS address are concatenated with 12 bits from the relocation register indexed by the high-order 5 bits of the UNIBUS address. The resulting 25-bit address is sent to the crosspoint switch and there selects a byte or word of shared memory, depending on the type of access.

Two UNIBUS pages (in kernel space) have no associated relocation register, and hence addresses in those pages remain untranslated. The first page addresses the small 8K-byte memory local to each processor, and the

[5]The two missing registers leave room for the processor's local memory and for the device register page. Four of the 30 registers are for the stack pages and are wired together so as to act as one.

second is left for the processor and device registers implemented by the PDP-11 and its peripheral devices. (The relocation registers themselves are addressed in this page.)

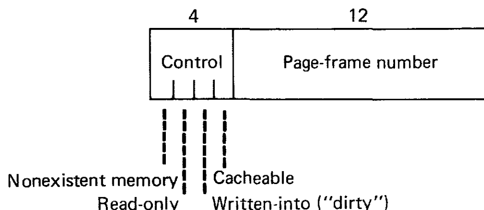

Figure 2-6 Format of the relocation registers

Each relocation register also contains a field of control and status bits, as shown in Figure 2-6. The *non-existent memory bit* can be set by the kernel to prevent access to that portion of the virtual address space. Any attempt to reference memory through a relocation register with this bit set will cause a trap. This permits the system to place a small user job in the machine without allocating a full 64K-byte address space. The *write-protect bit*, when set, permits read cycles to proceed through the register but blocks write cycles. This feature can be used to guarantee the integrity of code pages. The *written-into bit* (or *"dirty" bit*) in a register is set to '1' by any write cycle through that register. This mechanism is used by Hydra to avoid updating on secondary storage a page that has not been altered. The *cacheable bit* is used in conjunction with the processor cache to indicate that the page may be buffered. (The cache design is discussed in more detail later in this chapter.)

2-1.3 The Interprocessor Bus

The Interprocessor Bus provides a symmetric communication mechanism that allows any processor to invoke any of several control functions on any other processor or set of processors. Each processor has an interface to the Bus which resembles a normal peripheral device with several control registers (Figure 2-7). Each interface implements:

1. A programmable interval timer with 16-microsecond resolution
2. Access to the system's 56-bit time-of-day clock
3. Six interprocessor control registers

The interval timer consists of a time count register and a control register. Hydra can store a value into the count register, which will then be decremented every 16 microseconds as long as the "run bit" is set in the control register. Additionally, the timer can generate an interrupt when the count

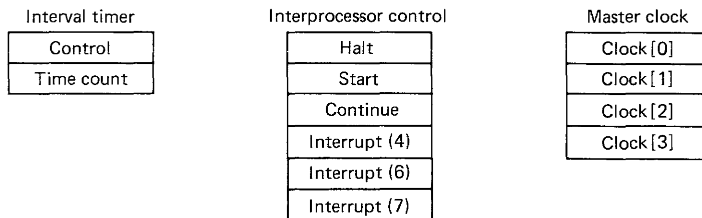

Figure 2-7 Interprocessor Bus interface registers

register reaches zero. Because the interrupt might not be serviced right away, the count register keeps decrementing so that precise timings can be obtained. Should the count be decremented to zero a second time before the interrupt is serviced, a status bit in the control register is set to indicate counter wrap-around.

The 56-bit time-of-day clock, also known as the "master" or "global" clock, has a resolution of 4 microseconds and is an important resource for Hydra. The Interprocessor Bus controller continuously broadcasts this clock value on part of the Bus. When a processor wishes to know the time, it reads the first of four registers in the interface, causing the interface to load all four registers from the Interprocessor Bus. The processor can then read the remaining three registers without fear of the value changing. The interface extends the clock value in the registers to 64 bits by adding the processor number and a "system version number," thus providing a unique value on all processors.

Each bus interface implements six control registers corresponding to the functions "halt," "start," "continue," "interrupt-at-level-4," "interrupt-at-level-6," and "interrupt-at-level-7." (See Figure 2-7.) If a processor sets bit *i* in one of these registers, the function associated with that register is invoked on the *i*th processor. Thus, for instance, a processor can halt the entire system by storing a word of all ones into the "halt" register.

Like the crosspoint switch, the Interprocessor Bus can be configured manually to partition the system. The 256 possible interconnections are controlled by 16 switches on each of the 16 processors. (A processor's switches indicate which other processors it may interrupt.)

2-2 THE ACTUAL C.MMP CONFIGURATION

Tables 2-1 and 2-2 detail the actual configuration of C.mmp in 1979: 11 processors, 2.6 million bytes of shared memory, 768K bytes of swapping storage, 700M bytes of secondary storage, and a normal complement of other peripheral devices.

C.mmp is not an isolated resource. It is connected to both the ARPA-

Table 2-1 C.mmp processor configuration

Processor	Peripherals
0	Operator's console, DECtape controller (2 drives), Line printer, 40M-byte disk controller (4 drives), Line frequency clock
1 - 5	(Processors not present)
6	Magtape controller (1 drive), Swapping disk
7	ARPANET interface, 2 swapping disks
8	Swapping disk
9	Swapping disk
10	Front end interface
11	Special applications
12	Swapping disk
13	(None)
14	130M-byte disk controller (3 drives)
15	Special applications

Table 2-2 C.mmp memory configuration

Memory unit	Technology	Size
0	(Unused)	
1	Core	64K x 18
2	Core	64K x 18
3	Core	64K x 18
4	Core	64K x 18
5	MOS	128K x 18
6	Core	64K x 18
7	Core	64K x 18
8	MOS	128K x 18
9	MOS	128K x 18
10	MOS	128K x 18
11	MOS	128K x 18
12	Core	64K x 18
13	Core	64K x 18
14	Core	64K x 18
15	Core	64K x 18

NET [Hea75] and to a front-end terminal multiplexor, as shown in Figure 2-8. The ARPANET provides a reasonably high speed link to each of three DECsystem-10s, considerably facilitating software development on C.mmp. The connection to the front-end multiplexor makes C.mmp immediately available to over a hundred terminals.

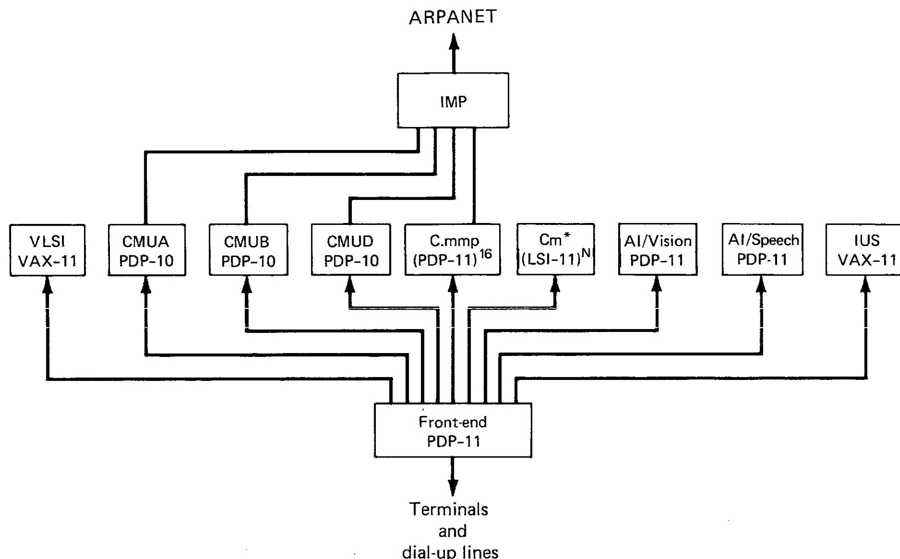

Figure 2-8 Interconnection of C.mmp to other computer systems

2-3 IMPLEMENTATION FEATURES

Descriptions of computer systems too often fail to point out those construction details which materially affect the final system structure. We now look at some of the most important aspects of the implementation of C.mmp. As a general comment we note first that, with the exception of a few off-the-shelf components purchased later, C.mmp was built entirely with 1970-1972 technology.

2-3.1 The Crosspoint Switch

The switch is the largest component of C.mmp. Unlike some other crosspoint switches which are distributed in memory (e.g., Pluribus), this one is located centrally. The central switch requires a larger initial configuration and implies some non-modularity, but the cost of a complete system is less than a distributed switch for larger configurations.[6] A centralized switch also has fewer cable delays than a functionally equivalent distributed switch.

The construction of the switch was simplified by building it with only four basic module types:

[6]Cable costs are a large component of these switches. The centralized structure requires only 16 + 16 cables, as opposed to 16 x 16 cables required for a fully connected distributed switch.

Switching modules
Processor interface modules
Memory control modules
Processor priority resolution modules

Each module is simple enough to be implemented on a single printed circuit board.

The main processor-memory data paths in the switch are 72 bits wide and are implemented with the switching modules in a bit-slice fashion. Figure 2-9 shows a single bit-slice of the switch. 16-to-1 multiplexors (SN74150s) implement the 256 crosspoints. Sixteen of the multiplexors are used to implement the paths from the processors to the memory units, and the other 16 multiplexors are used to implement the return paths from memory to the processors. The symmetry between the multiplexors forming the forward and return points allows two switch modules, each consisting of 16 multiplexors, to implement the bit slice shown in Figure 2-9. Control of the multiplexors comes from the processor priority resolution modules. The 144 switch modules needed to construct the data paths in the switch form the bulk of the logic in the crosspoint switch.

The processor interface module connects the switch to the memory relocation units on each processor; it contains the steering logic to partially decode the address lines and route the memory request to the designated memory module. This module also sets the memory-to-processor selection lines in the switch, thus determining which memory the processor will read. Finally, this module buffers data read from memory, allowing the switch to overlap the end of a read cycle with the start of the next cycle for another processor.

The memory control modules are quite straightforward and provide three important functions:

1. They check the address parity that was generated in the relocation hardware and report any errors back to the processor.
2. They detect missing portions of memory, so that the physical address space need not be contiguous within a memory subunit.
3. They communicate with the processor priority resolution modules in order to generate the timing and control pulses for the actual memory modules.

Within each memory subunit, individual memory modules reside on a central bus and may be interleaved. Each core memory module consists of two pages (16K bytes) and is independently driven. Semiconductor modules have a single driver for their four 64K-byte boards and therefore offer less chance for interleaving. Core and semiconductor technologies may be mixed within a subunit, but typically are not.

The processor priority resolution module is the most complex component in the switch design, maintaining a request buffer whose operation is illus-

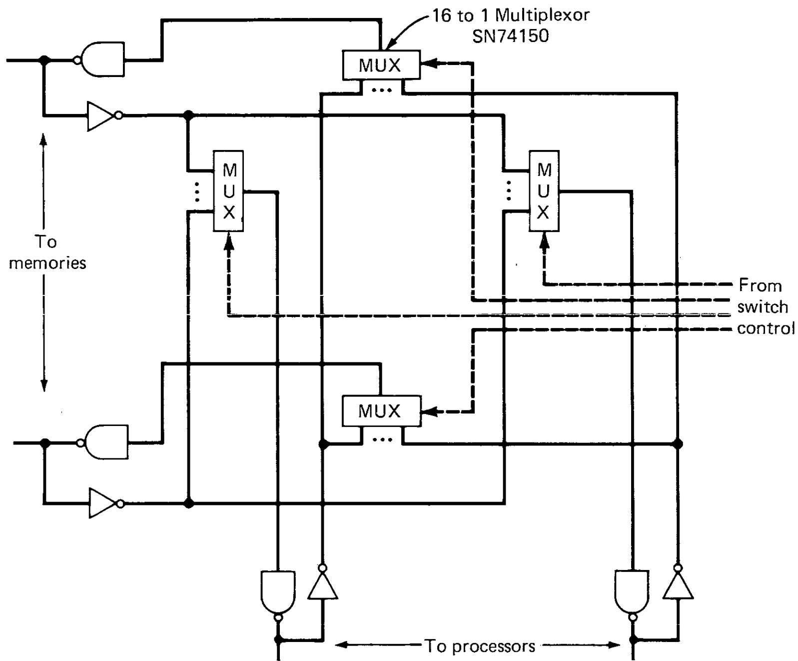

Figure 2-9 Bit-slice of crosspoint switch data paths

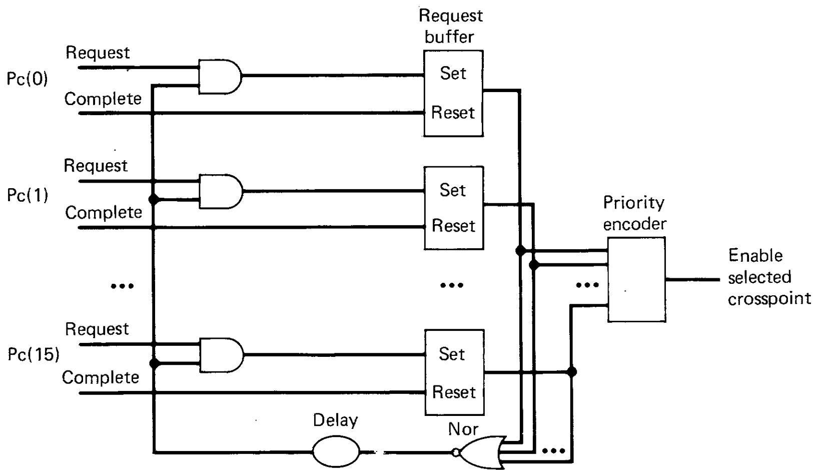

Figure 2-10 Simplified processor arbitration logic

trated in Figure 2-10. This module arbitrates between processors that are simultaneously requesting access to the same memory port and queues those

requests that must wait for other requests to complete. The arbitration logic shown in Figure 2-10 works in the following manner. When processor *i* requests access to a particular memory port (as indicated by the value of the four most significant address bits), it attempts to set bit *i* of the request buffer. However, the AND gate in front of the SET input to the buffer prevents processor *i* from setting latch *i* until the request buffer contains all zeros. When the request buffer is empty all 16 AND gates feeding the SET inputs of the request buffer are enabled via the OR gate and DELAY shown at the bottom of the diagram. Now those processors with outstanding requests will set their corresponding latches in the request buffer. As long as a single processor is making a request, and sets its corresponding latch, the column of AND gates will be disabled since the request buffer is no longer empty. Now the outputs of the 16 latches of the request buffer are fed into a priority encoder that indicates on four output lines the highest numbered latch that is set. It is this priority encoder, therefore, that ultimately does the arbitration. After a processor has been selected and has read or written a word of memory, the processor asserts its "access complete" line that clears the processor's latch in the request buffer. The priority encoder now selects the highest-numbered remaining request. Hence, processors are serviced in priority order from 15 to 0, and each processor should wait no more than 15 memory cycles before gaining access to memory. The scheduling discipline induced by the priority resolution modules can be thought of as a quasi-round-robin discipline.[7] (Unfortunately, the actual behavior may be different; see Section 16-2.5.)

2-3.2 Processor Modifications

The modifications to the processors can be considered to be in two classes: additions and alterations. For the PDP-11/40E, only a very small percentage of the work is in altering existing logic. For instance, the detection and trapping of reserved instructions in user space requires only the addition of two ICs and the replacement of two others on the instruction-decode module of the processor.

The addition of the other features requires that about 30 processor-generated signals be acquired from the backplane of the processor. Additions to each processor are all contained on one new PDP-11 system board. (A standard PDP-11/40 is implemented on five such boards.)

[7]This discussion of the priority resolution module is a simplification of the actual operation. In reality, there is also a high-priority input to each latch in the request buffer that circumvents the column of AND gates. This high-priority feature is intended for very fast I/O devices which may not be able to tolerate a high level of memory interference during DMA transfers. Although this feature is exploited by the operating system, its efficacy has never been proved conclusively.

2-3.3 Extensions for Error Detection

The most significant step in overall system error detection was the implementation of parity bits in shared memory. The relocation hardware computes parity bits for each byte written to memory and for every address sent to the memory. To catch common failure modes of "all ones" and "all zeros," we use even parity on one byte of each data word and odd parity on the other byte. Address parity is checked by the memory controller on the memory side of the switch, and data parity is checked on each "read" cycle by the relocation hardware at the processor. The switch actually has data paths wide enough for single-bit error-correcting codes on each data word, but such a mechanism was never implemented.

The two PDP-11 system boards that implement the memory relocation logic are also the site of much of the error-detection circuitry. Upon detection of a switch-related error (parity errors, writing a read-only page, etc.), the logic causes the processor to take a normal NXM ("non-existent memory") trap by blocking the acknowledgement signal ("SSYN" in PDP-11 terminology) from memory. The fact that a *trap* (rather than an interrupt) is taken is important, because traps can take effect before the completion of an instruction. Status bits in a control register allow the software to determine the actual cause of the error and can cause later errors to be ignored until the processor's state is recorded. Other exceptional conditions, including stack underflow, violation of the SP conventions, and attempting to execute an illegal instruction, cause normal interrupts.

Other error-related mechanisms were added to C.mmp later in response to observed failures. For instance, the PDP-11's variable-length instructions and its rich set of addressing modes makes locating the exact source of an error (e.g., a parity error) difficult. For this reason, we implemented two *tracking registers.* The *bus address tracking register* is latched upon the occurrence of a switch-detected error (e.g., a data or address parity error) and thus accurately specifies the UNIBUS address causing the error. The *PC tracking register* latches the address of the current instruction under the same circumstances.

Maintenance functions are also implemented in the relocation hardware, including the ability to simulate address parity errors and the ability to write incorrect parity into shared memory.[8]

2-3.4 Caches

The original design of C.mmp included a 1K-word cache on each PDP-11 UNIBUS. As of July 1979 only one cache was installed, and it is not used by

[8]The tracking registers and maintenance features were implemented in 1978 on the third (and last) versions of the relocation hardware. They were never completely integrated into Hydra's error-handling mechanisms.

Hydra. Therefore, we can describe only the intended operation of the caches.

Caches present a potential difficulty for multiprocessor systems because data shared between processors may be modified in one processor's cache without the modification being reflected to other processors. We chose to solve this problem by avoiding it; pages that are both shared and writeable are never cached. The operating system can designate (via the *cacheable* bit in the relocation registers) those pages which are safe to cache. Studies on the PDP-11 indicate that about 70% of all memory references are to code pages, which can be read-only and hence cacheable. Stack pages are private to a process and hence are also cacheable. In addition, the user may explicitly designate other cacheable pages.

It should be noted that the caches designed for C.mmp do not have to be fast; their importance lies in their ability to eliminate switch contention by catching a significant fraction of the memory fetches. This is especially important because Hydra encourages the sharing of code pages among cooperating processes, thus inviting significant contention.

2-3.5 The Interprocessor Bus

The Interprocessor Bus controller performs three functions: it implements and broadcasts the time-of-day clock value discussed above; it generates and broadcasts the timing pulses that are used by the interval timers in the interfaces; and it generates and broadcasts the timing and control signals necessary to time-multiplex the various interprocessor control functions on the bus. By using a time-sliced function bus, we reduced a potential 1500+ wire requirement to 16 cables of 20 wires each; however, we give up knowing which processor invoked a function. This is not a significant restriction in practice.

2-3.6 Peripherals

C.mmp has an extremely high I/O bandwidth; each processor can support independent DMA transfers from multiple devices. Assuming each processor hosts a device with a transfer rate of 4 μs per 16-bit word, this amounts to a potential I/O bandwidth of 64×10^6 bits/sec.[9] Hydra exploits this potential by using a collection of fast disks, distributed over several processors, for swapping storage.

These fixed-head swapping disks are perhaps worthy of special note. C.mmp's page size is exactly equal to the capacity of one track on the disk, and by modifying the controller slightly, this coincidence can be exploited in such a way that there is no significant rotational latency on disk transfers of

[9]This assumes a 16-processor system on which the I/O traffic is distributed fairly evenly over the 16 memory ports to avoid memory contention.

exactly one page. Latency is avoided by having the controller start the transfer at the beginning of the next physical disk block (16 words) and transferring 8K bytes without track switching. By causing the first transfer to its correct memory address, and inhibiting the appropriate carry propagation when incrementing the memory address register, the trasfer will "wrap around" within its proper 8K memory area. This scheme provides better service than "shortest-latency-time-first" or any of the other scheduling disciplines that have been developed to optimize the performance of paging disks with latency.

2-4 TECHNOLOGY AND COSTS

C.mmp is a mixture of off-the-shelf and custom-built hardware. Table 2-3 gives an approximate breakdown of the equipment in terms of complexity and cost.

The portions of C.mmp built at C-MU use a mixture of TTL and Schottky TTL technology. ECL was not used because at the time of C.mmp's construction (1971) ECL did not offer the range of MSI components available in TTL. Likewise the large amount of ferrite core memory on C.mmp is due to the state of MOS memory technology in 1972.

The cost figures given in Table 2-3 are only estimates. The cost for the PDP-11/40 and for memory was the purchase price of the equipment when we bought it. The other hardware was built at C-MU, and the figures given are our rough estimates of the replication cost in 1975, excluding design and setup costs.

Using these figures, the total replication cost of a 16-processor C.mmp, excluding peripherals, is about $500,000. Of this total, about $285,000 is for the modified processors and relocation hardware, $165,000 is for 2.3 million bytes of primary memory, and $50,000 is for the crosspoint switch.

2-5 HARDWARE PERFORMANCE

The performance of a computer system cannot always be calculated from the speed of its components, but for comparison purposes some of C.mmp's vital statistics are shown in Table 2-4. The processor and memory speeds are taken from actual measurements on the running system.[10] Chapter 16 gives more data on the performance of the hardware and software.

On a PDP-11/40, one instruction requires about 2.5 memory references on the average, so 0.68 million memory references per second translates to

[10]The processor speed is the average speed of a single processor executing out of a single memory port with no other processors contending. The memory speeds are averages at a single memory port with all processors contending on that port.

Table 2-3 C.mmp technology and costs

Part	No. boards	No. ICs	Unit cost	No. in system
11/40	5	332	$12,000	11 (16 max)
μstore	2	200	$1,300	1/Pc
Pc mods	1	57	$600	1/Pc
Relocation Hardware	3	120	$1,500	1/Pc
Crosspoint Switch			$50,000	
SW16	1	24		144
P.I.M.	1	26		1/Pc
M.C.M.	1	20		1/Memory unit
P.R.M.	1	54		1/Memory unit
Interprocessor bus:				
Control	2	200	$3,000	1
Interface	1	200	$3,000	1/Pc
Memory (core)	unit=8K x 18		$1,300	80
Memory (MOS)	unit=128K x 18		$12,000	5

Table 2-4 C.mmp hardware performance

Parameter	Value
PDP-11/40 execution speed	0.68 x 10^6 memory references/second
Memory (core)	1.5 x 10^6 memory references/second
Memory (MOS)	1.7 x 10^6 memory references/second
130-Mbyte disk	2.5 μs/word transfer rate 28 ms average seek 8 ms average latency
20 and 40-Mbyte disks	7.5 μs/word transfer rate 29 ms average seek 12.5 ms average latency
Paging disks	4.1 μs/word transfer rate 17 ms page read time 34 ms page write-and-verify time

about 0.27 MIPS (million instructions per second) for each processor, or about 3 MIPS for the 11-processor configuration and 4.3 MIPS for a full

16-processor configuration (with no memory contention). The above figures are averages. Studies by Oleinick [Ole77] have indicated that individual processors and memories may vary from this average by as much as 10% (see Chapter 16).

2-6 RETROSPECTIVE

If we were to build C.mmp again we would do a number of things differently. The hardware designer now has many more options than were available in 1972, including more powerful processors and a wider range of faster components. Still, we have had two chronic complaints about the realization of C.mmp that are relevant to contemporary design: its 16-bit virtual address and its disappointing reliability. The problems came largely from two assumptions we made at the outset which turned out to be wrong:

1. We assumed that large applications would run efficiently on a multi-mini-processor because large uniprocessor programs could be broken down into several small concurrent processes.
2. We assumed that we would be able to construct C.mmp easily because we were using mostly off-the-shelf components with minimal modifications.

The first assumption was wrong; we discovered that large applications almost invariably want to address large amounts of data, even when they are decomposed for a multiprocessor. The 16-bit virtual address provided by the minicomputers simply did not provide enough freedom for manipulating data. This is not just a matter of efficiency; it seriously affects the ease with which large programs can be designed to run on C.mmp [Ole77].

The second assumption was correct in principle, but could not withstand the realities of minicomputer architecture in 1970. We never expected to have so many problems with our original PDP-11:

1. The PDP-11 had several original design errors that vastly complicated hardware debugging. Some of its more complicated instructions did not always work correctly.
2. With no comprehensive documentation of the PDP-11 other than the logic diagrams, some subtle points of the implementation were not discovered until our modifications tickled them.
3. The UNIBUS turned out to be unexpectedly fragile for a device ostensibly designed to accept a diverse set of peripherals. The UNIBUS has no parity checking, it is prone to noise, and devices on it cannot be powered down without affecting it.

For the most part, the C-MU-built portions of C.mmp have performed well. The crosspoint switch has had very few problems, perhaps because of

the extreme care taken to avoid "glitches" in the design.[11] The Interprocessor Bus exhibited unexplained problems when we attempted to run the master clock at its full 1 μs resolution. Even after slowing down the clock by a factor of four we observed intermittent periods during which the clock values received at a processor were incorrect. Software mechanisms eventually had to be introduced to validate the clock values, nullifying some of the expected advantages of the clock to the operating system in the first place.

Many of these problems had indirect effects on the system. The fragility of the UNIBUS had two effects. First, any reconfiguration of the system that involves altering a UNIBUS configuration cannot be attempted while the system is running. This thwarts many approaches to improving system reliability. Second, because we designed the crosspoint switch to work closely with the UNIBUS, we left the entire system susceptible to single UNIBUS failures. A malfunctioning peripheral device interface can "hang" its UNIBUS and the entire crosspoint switch. Fortunately, this type of error is rare except when interfacing new (undebugged) devices.

Finally, the decision to modify the processors as little as possible meant that any extensive alteration of functionality was impractical. This is one reason why the address translation mechanism is so rigid, and why the stack was implemented the way it was (instead of, for instance, making separate user and kernel stacks).

In spite of these problems, we believe the crosspoint architecture remains a good basic design for the tightly coupled multiprocessor.[12] Our experience with C.mmp suggests that in a general-purpose system the ease of software construction made possible by the underlying symmetry of the hardware more than compensates for the lack of easy expansion to more processors. Subsequent chapters describing Hydra's scheduling and error-handling mechanisms will demonstrate how C.mmp's structure may be exploited.

[11] *Glitch*, as used here, is a technical term, referring to the propensity of an arbitration circuit to remain in a meta-stable intermediate state for more than a prespecified settling time.

[12] A comprehensive survey of multiprocessors appears in [Ens77].

PART TWO

THE SYSTEM DESIGN

CHAPTER

THREE

THE HYDRA PHILOSOPHY

A basic goal of the Hydra design was to permit nearly all the facilities that one normally associates with an operating system to exist as normal user-level programs, and in addition, to allow an arbitrary number of user-level definitions of a single facility to coexist simultaneously. This goal arose in part from Hydra's position as a vehicle for experimenting with the C.mmp multiprocessor and in part from a set of attitudes held by the designers about what constitutes good software design. These attitudes are partially reflected in the following paragraphs.

Facilities The Hydra host machine, C.mmp, is a multiprocessor. It is not immediately obvious what facilities are appropriate in such an environment. On the contrary, one suspects that the "right" facilities (e.g., for debugging a collection of cooperating processes) will be found only after considerable experience. By allowing these facilities to be provided at the user level (by unprivileged programs written by ordinary users), one gains considerable freedom to grow and evolve the system in unanticipated directions.

Extensibility All practical systems evolve under the pressure of usage patterns and hardware innovation (especially new peripherals). The proliferation of "access methods" in OS/360 and its offspring is a prime example of evolution under both of these types of pressure. Such evolution is a fact of life in practical systems; the original design should anticipate it. By providing nearly all facilities at the user level, without special privilege or status, Hydra addresses this problem directly. Adding a new facility consists simply of providing a user-level program that implements that facility; the difficulty in doing so is directly related to the inherent complexity of new facility. It may be simplified by the prior existence of some facilities, but is never hindered by them.

Structure Most people now recognize the intimate relation among the structure of a program, its probability of being correct, and the ease with which it can be modified. Providing facilities at the user level does not, of itself, guarantee a well-structured system, but it does assure a uniform interface between the various pieces of a system. The specific method we

have chosen in Hydra for allowing facilities to be defined at the user level is strongly related to modern notions of good program structure; we shall have more to say about this below.

Multiple usage patterns Most operating systems attempt to cater to a variety of coexisting usage patterns. A simple example appears in systems whose schedulers employ different strategies for batch, time-sharing, and real-time jobs. However, there are more subtle cases—once again, the multiple access methods of OS/360 exemplify the situation. By providing the ability to define facilities at the user level, one obtains much more freedom to refine, adapt, tune, and extend these facilities to match a specific application.

Our image, then, of the Hydra environment was that there would evolve a collection of these facilities defined at the user level, possibly with many that were functionally similar but with different performance properties, security properties, or whatever. An individual user would select from among the available facilities, or create new ones where the existing ones were inappropriate. At any instant we expected to see a large number of users on the system—each possibly using quite different facilities—but all coexisting without interference. It seemed to us to be the ideal environment in which the user might experiment with multiprocessing.

3-1 ACHIEVING THE GOAL

These are, of course, all "motherhood" statements. Whether or not it is *in fact* desirable to build operating system facilities as user programs depends strongly upon the specific mechanisms used to achieve the goal—their cost and convenience. For the moment let's assume that the goal is desirable and examine the implications of the goal on these mechanisms.

The central goal suggests that at the heart of the system one should build only basic, or "kernel," mechanisms—a set from which arbitrary user-visible operating system facilities can be conveniently, flexibly, efficiently, reliably, and quickly constructed. Moreover, lest the flexibility be constrained at any instant, (1) the kernel mechanisms should not preempt important decisions, and (2) it should be possible for an arbitrary number of systems created from these mechanisms to coexist simultaneously.

This is obviously a tall order. Nevertheless, in the remainder of this chapter we shall assume that such a set of kernel mechanisms exists. Hydra is an attempt to provide just such a set, and subsequent chapters detail its properties. Whether or not the particular Hydra mechanisms satisfy these criteria and whether or not they form the best set are, of course, debatable. The reader is encouraged to answer these questions for himself. We can, however, help to provide some insight into the answers; to that end, the

remainder of this chapter addresses the rationale for the mechanisms, the remainder of Part II deals with the mechanisms in general terms, and the remainder of the book gives examples of their use.

We can easily rationalize two properties that the kernel mechanisms must possess: (1) protection and (2) minimal policy. Consider for the moment two common descriptions of the purpose of an operating system:

1. An operating system provides a "virtual machine" which is more hospitable than the base hardware for two reasons: (a) It makes available certain "virtual resources" such as files, directories, virtual memory, etc., that are absent from the base hardware. (b) It masks certain unpleasant hardware features—such as interrupts—from the user and maps them into more acceptable ones, such as synchronization primitives.
2. An operating system manages the physical resources of the computer, such as primary memory, processor, channels, etc., so as to improve their utilization.

Even though these descriptions are quite different, they are not incompatible—they merely express two quite different views of a single entity with multiple goals.

From the first of these descriptions we see that the user must be able to view some collection of facilities as a virtual machine—a closed environment in which he can program. Facilities (or users) outside the collection should not be able to perturb the machine's behavior. That is, the user program must be able to behave as though it were running in isolation (except for possible differences in real-time behavior). Thus, a uniform requirement of all multi-user operating systems is that they provide *protection.* In our case, since operating systems are themselves user programs, the only candidate for providing the necessary protection is the kernel. Moreover, the protection provided must be both strong enough and flexible enough to permit user programs to implement operating system functions.

From the second description we can derive a negative requirement on the kernel mechanisms: they should not impose a policy on the way in which physical resources are used. If the kernel mechanisms were to do this they would preempt the possibility of specifying these at the user level—and hence preclude an important dimension of operating system variation. As we shall discuss below, there are practical problems with allowing arbitrary policy decisions to be made by user-level programs; these difficulties force us to a compromise goal: the separation of policy from mechanism.[1]

[1]Brinch Hansen [Bri70, Bri71] has made cogent arguments for this separation.

3-2 PROTECTION

The most prevalent views of protection in operating systems are quite narrow. Often, for example, it is presumed that

- There are only a small, fixed number of kinds of things that need to be protected, e.g., "file" and "memory."
- There are only a small, fixed number of kinds of access to the protected objects, e.g., "read," "write," and "execute."
- The right to perform a specific access to an object is a property of the "user" (a person) making the request.

None of these assumptions are appropriate for the Hydra goals.

We will discuss the Hydra protection mechanism and its use later in this chapter as well as later in the book. For the moment, however, will simply note some of its properties, contrast them with the more traditional views, and point out how these properties support the Hydra goals.

First, Hydra does not predefine a fixed collection of things that can be protected. Instead, it defines a general notion of a "typed object." One type of object, for example, might be a file; another might be a page of memory; another might be a catalog. New types of objects can be defined at will, and each of these types can be protected. This is essential to the system's goal; since we wish to permit users to define new operating system facilities—a new type of file system, for example—the protection mechanism must extend to cover these new facilities in the same way as it covers the old ones.

Second, Hydra does not predefine a fixed collection of access rights. Instead, it defines a general notion of "applying an operation to an object"; the right to apply an operation is the fundamental protection check in the system. Since the concept of an "operation" is type-specific, a totally different set of operations—and hence protection rights—can be associated with different types of objects. For example, "read" and "write" are indeed defined notions for both files and memory pages; however, the notion of "execute" is not defined for files—and the notions of read and write may not be defined for "programs," which are a different type from "files." Furthermore, "catalogs" have the additional concept of "rename."

Finally, the "right" to perform an access (an operation) is not necessarily a property of the (human) user in Hydra. Rather, it is a property of the program that is executing on the user's behalf.[2] The right to call such a program, naturally, is at least initially inherited from the user, but associating access rights with the program itself has several important advantages. In effect, it allows the program to perform operations on behalf of the user on objects to which the user does not have direct access. In a conventional

[2]The details of this are a good deal more subtle than this overview can hope to cover, but we will have more to say in subsequent chapters.

system the user calls upon the operating system to perform these kinds of operations; for example, he invokes the operating system to update his file directory because he is not allowed to write into it himself. In Hydra, this notion is extended to all object types. The user may, for example, create a special data base and programs to manipulate it. He may then allow other users to invoke these programs, thus updating the data bases, while never permitting direct access to the files (or whatever) on which the data is stored.

3-3 POLICY/MECHANISM SEPARATION

To enable the construction of operating system facilities as normal user programs, we must allow user-level control of the policies that determine the utilization of the system's resources. The resources of primary interest are those required by every program: processor cycles, memory, and input/output. The policies that govern the allocation of these resources are a major dimension of operating system variability. As many of us know from bitter experience, the policies provided in extant operating systems, which are claimed to work well and behave fairly "on the average," often fail to do so in the special cases important to us. A goal of Hydra is to allow these policies to be defined by user-level (i.e., unprivileged) programs, thus making them more amenable to adaptation and tuning than they would be if buried deep in the system's kernel. Moreover, to permit each application to tune the system to its own needs, we wish to allow multiple policies governing the same class of resource to exist simultaneously.

At this point, practicality intrudes; in fact, it intrudes in several ways. First, we must assume that any user-level program contains bugs and may even be malevolent. We therefore cannot allow any single user or application to "commandeer" the system to the detriment of others. By implication we must prevent programs that define policies direct access to hardware or data that could be (mis)used to destroy another program. That is, such programs must execute in a protected environment.[3] Further, we must not permit these programs to monopolize any resource, whether they do so intentionally or not. We must assure some "fairness" among competing policies. In addition, we must recognize that many policy decisions must be made rapidly (e.g., fast scheduling decisions are essential in order to achieve reasonable response). Given that user-level policy programs must execute in their own protection domains and that domain switching is costly, it is impractical to invoke such programs each time a policy decision is required.

Thus, we compromise. We give this compromise a name: the principle of *policy/mechanism separation.* Policies are (by definition) encoded in user-level software that is external to the kernel. Mechanisms are provided

[3]Obviously, all programs must be denied such liberties, but policy-making programs frequently require access to information that might normally be considered privileged.

in the kernel to implement these policies. In this context we use the phrase "kernel mechanisms" to mean two distinct but related things.

In the first instance we mean simply a safe (protected) analog of an unsafe hardware operation. Thus, for example, we never allow a user program to manipulate directly input/output device control registers. To do so would allow that user program, possibly inadvertently, to overwrite an arbitrary portion of memory. We do, on the other hand, provide a mechanism, a kernel operation, whose only effect is to manipulate such device control registers after appropriate validation.[4] Mechanisms such as this exist purely to insulate the system and other users from a misbehaving policy program.

In the second instance a kernel mechanism may actually be a parametrized policy. We shall deal with several examples of such mechanisms subsequently, but it is convenient to introduce one here to illustrate the point. A portion of the kernel called KMPS ("Kernel Multi-Programming System") provides primitive scheduling and synchronization facilities. KMPS uses a simple, priority-driven scheduling scheme; processes at the same priority level are treated in a "round-robin" fashion, and preemption of a process may occur either when a higher-priority process becomes feasible (unblocked) or at the end of a time-slice.

This description of the KMPS scheduling strategy may sound familiar; it is similar to those employed in many other systems. There is an important difference, however. The priority of a process, its time-slice, and other parameters (to be discussed later) are determined by a user-level policy program associated with the process, called a *Policy Module*, or *PM*. Several PMs can exist simultaneously, each controlling a different set of processes.

At intervals specified by the PM (and at other times to be described later), KMPS relinquishes control of each process to the PM associated with that process. At such points the PM may elect to alter the process' behavior as it chooses. Possible actions include changing the process' scheduling parameters and returning it to KMPS, or removing the process' pages from core and making the space available to other programs.

Mechanisms such as KMPS, which are really parametrized policies, provide the means by which overall, long-term policies can be enforced by user-level software, while at the same time avoiding the need to invoke a ponderous domain-switching mechanism for decisions that must be made rapidly. Such mechanisms also provide a point at which fairness among competing policies can be enforced. KMPS, for example, could attempt to provide each PM with an equitable share of the processing power of the machine.

[4]To perform such an operation the program must have appropriate access rights. Although it is too soon to be able to explain in detail, it is interesting to note that the Hydra protection system is used uniformly for both virtual resources such as files and physical resources such as I/O devices.

3-4 AN ASIDE ON DATA ABSTRACTION

The view that the kernel mechanisms should provide protection and should not define resource policies does not of itself provide sufficient information on which to base a design; it merely specifies some properties that the design must have. To develop an appropriate basis, we choose to turn away from traditional operating system design considerations and to look instead at some of the more recent results of "structured programming."

It is unfortunate that the term "structured programming" has too often been equated with "goto-less programming" or "top-down design." Far more central to the issue is the concept of "abstraction." The whole rationale for structured programming is that programs, even "simple" ones, are often too complex for human beings to comprehend. A classical technique that humans use when faced with complexity is to abstract from it—to ignore the details of a problem and deal instead with only its "essence." Avoiding the "goto" and using top-down design are both abstraction techniques—they both provide a means for localizing the implementation of an abstract concept within a well-defined region of the text of a program. There are, however, other techniques for abstraction in programs, and some are much more powerful than either of these.

Several authors have noted the close relation between many programming abstractions and the concept of "type" as it appears in programming languages [DDH74, Bri75, Wul74]. This has led to considerable interest in ways in which the programmer might express these abstractions in a program and, in particular, what sort of language constructs support this "abstract data type" definition. Specifically, the concept of a "class" in Simula '67 [Dah66] and its extension to "monitors" [Hoa74] seems especially well suited to expressing these abstractions. A class in Simula defines an abstract data type by specifying both an underlying storage structure and a set of operations that operate on it. Thus, for example, the abstract concept of a *set-of-integers* might be introduced into a language by a definition of the form

```
type intset =
  begin
    var a: array[1:100] of integer, n: integer;
    op union(u,v: intset) returns intset;
        begin ... end;
    op intersect(u,v: intset) returns intset;
        begin...end;
  end;
```

(We have chosen a neutral syntax whose meaning should be clear; it is not Simula '67 or any other specific language.) Such a definition is intended to describe how any particular variable of type *intset* is to be represented and

how operations on this type of variable are to be performed. Thus the declaration

```
var a: array[1:100] of integer, n:integer;
```

defines how storage is to be allocated for each variable of type *intset*. The operator definitions, e.g., that for *union*, define how such variables are manipulated. An important property of such definitions is that all the representational information is localized and "hidden" in the type definition; the *only* way to manipulate variables of a defined type is by invoking the operations defined in the type definition.
After having made such a definition, the programmer may write such things as declarations of variables of type *intset* and statements that operate on these sets, e.g.,

```
var a,b,c: intset;
a := union(b,c)
```

It is important to note that the newly introduced type, *intset*, can be given the same status as the predefined types (e.g., integer); one can declare variables of the new type and perform operations with essentially the same syntax as is used for the predefined ones. The ability to deal with the new types in this way aids our human capacity to deal with them as abstract ideas.

This style of programming captures an essential aspect of abstraction: it effectively separates the application of the abstract "primitives" from the details of their implementation. The programmer, working at a level where *intsets* are an appropriate medium of expression, need never concern himself with the details of how they are represented or manipulated. Conversely, the implementor of the realization of the type *intset* may freely alter that realization (to improve efficiency, for example) without concerning himself with the details of how it is used, as long as he preserves the functional properties of the operations.

It is not our purpose here to advocate a particular approach to structuring programs. However, the brief description given above is the model on which Hydra is based. Except for a slight change in terminology, extensions to provide protection, and a more dynamic definition of types than is common in programming languages, the Hydra kernel mechanisms were chosen to support this model.

3-5 TYPES AND RESOURCES

Earlier we used the phrase "virtual resources" to describe some of the facilities provided by an operating system (e.g., files). The meaning of this phrase is essentially identical to that of "type," or "abstract data type," as

used in the immediately preceding discussion. A virtual resource (e.g., file, directory, semaphore, etc.) is an abstract concept with a set of operations defined on it (e.g., for files: read, write, append, open, close, etc.). Moreover, the virtual resource has some realization in terms of more primitive concepts (e.g., disk segments). Just as with structured programs, we want the user of the file system to be unconcerned with the details of its implementation. Conversely, we want the implementor of the file system to focus on the issues related to that specific realization without concern for the details of the idiosyncratic use of a particular file.

Without yet concerning ourselves with the details of the Hydra mechanisms, we proceed by analogy with the programming language model and list properties that these mechanisms must have (the first two are copied from the previous discussion for completeness):

Protection
Policy/mechanism separation
Creation of new kinds of virtual resources (new types)
Specification of the representation of, and the operations on, a virtual resource
Creation of instances of a resource
Application of operations to an instance of a resource
Certain "generic" operations, e.g., "storing," that are applicable to all resources

We shall often use the phrase *subsystem* when speaking of operating system facilities; it will mean essentially the same thing "type definition" meant in the previous discussion. That is, a subsystem is a collection of information that specifies the representation of a virtual resource (type) and the nature of the implementation of various operations on that type of resource. All knowledge about these representational and operational details are contained and "hidden" within the subsystem. In those cases where resource allocation (policy) issues are involved, these policies are also embedded in the subsystem. Global knowledge about a specific type of virtual resource is limited to that supplied in the external specifications of the subsystem that implements that resource. Manipulation of the representation of a resource is restricted by the protection mechanism to only that code that defines the operations within a subsystem.

At this point we can pose a question, which we purposely avoided previously, about the protection structure of the system: "What should be protected, and against what?" This apparently simple question is complicated by two issues: one endemic to operating systems; the other arising from the primary goal of Hydra.

First, we recognize that sharing is as important as protection. That is, we don't really want *complete* isolation of the virtual machines seen by various users. Users want to share selectively files, pages, directories, semaphores,

or any of the other virtual resources provided to them. This is true in any "computing utility" but especially so in a multiprocessor, where a single user will wish to divide his job into parallel cooperating processes and share resources between these processes. Second, because we wish to provide virtual resources through user-level programs, we don't know *a priori* what kinds of resources will exist. Hence we don't know what sorts of things will need to be protected or what sorts of access should be granted (or prohibited) to them.

Both of these questions can be answered in terms of the data abstraction, or "abstract data type," model discussed above. The objects to be protected are instances of virtual resources. We shall insist that *only* the operations defined to operate on a type may manipulate the representation of objects of that type. In addition, the protection mechanism provides the means of selectively granting or prohibiting application of these type-specific operations to particular objects of the type. Thus, for example, suppose type "file," with associated operations "read," "write," "append," "open," etc., has been defined. The protection mechanism will allow application of, for example, the "append" operation to specific instances of files to be selectively granted or inhibited.

3-6 PARALLELISM

Obviously, C.mmp provides the opportunity for true parallelism, and thus a major goal of Hydra was to exploit this possibility. There are two aspects to this goal: to exploit parallelism within the Hydra kernel itself, and to provide sufficient facilities that users can write asynchronous parallel algorithms.

With respect to Hydra itself, there are several consequences of the goal. For example, Hydra was written to operate in a distributed fashion—that is, it is *not* a master-slave system. Any processor can execute the kernel and, in fact, an arbitrary number of them can execute it simultaneously. Further, resources such as the processors are treated as an anonymous pool; user processes can execute on any processor and, in fact, may switch from processor to processor many times during their lifetime. The kernel hides most asymmetries of the hardware, such as the binding of specific devices to specific processors; the user process does not need to be executing on the same processor that controls the I/O devices it uses. In order to maximize the potential parallelism in the kernel, Hydra places locks on data structures, not on the code that accesses them; thus, parallel execution of the same code on different data is common.

With respect to facilities for user-level parallelism, we took a conservative approach. As in other areas, we provided (only) a minimal set of facilities that we believed were adequate as a basis for user-level extension, and we attempted to encourage such extension. We felt that we did not then know

enough to dictate a particular style of parallel program structuring; even now, after considerably more experience both with C.mmp and other multiprocessors, our attitude on this has not changed. Thus, we provided a rather traditional notion of processes together with facilities for sharing of arbitrary objects and basic synchronization and communication.

3-7 SUMMARY OF THE GOAL

Before proceeding to more details of Hydra, let's briefly recap the central goal and its external manifestation. The primary goal is to permit conventional operating system facilities to be built as normal user-level programs. To do so we conceive of an operating system as partitioned into several pieces. One distinguished piece is called the *kernel*, whose basic function is to support the existence of the remaining pieces. The kernel provides a uniform protection mechanism and avoids arbitrary policy decisions. The remaining pieces—and there may be an arbitrary number of them—are called *subsystems*. Each subsystem defines the representation of a virtual resource and the implementation of operations on instances of that resource. It also is responsible for all policy decisions relative to that resource.

It is crucial to note that this model makes no assumptions about the number or kind of resources provided by the subsystems. It is *not* implicit in the data abstraction model that there be one (or one hundred) subsystems that implement a "file"; in fact, it's not necessary, from a theoretical point of view, that there be a file system at all! Nor is there any implicit grouping of subsystems. The "operating system" as seen by any particular user is merely a collection of subsystems that implement the facilities that the user needs. The collection might be the same as that for another user, partially overlapping with it, or totally disjoint. It might include a single file system, or several if the different properties of the files suggest this is appropriate.

This, then, is the goal of Hydra. We hope that at this point the reader has a general impression of the system's aims. We realize the reader may be at a loss for concrete information. Perhaps he may also feel disquieted about the cost and usability of all the flexibility implicit in this discussion. The following chapters will attempt to provide the information and data on which a rational evaluation of Hydra can be based.

3-8 FURTHER READING

The philosophy described in this chapter did not evolve in a vacuum; it was contemporaneous with much of the work on programming methodology and modern "data abstraction" languages. These areas remain subjects of vigorous investigation, with new results appearing continually. At the time Hydra

was being designed, however, the principal influences were the emerging notions of modular decomposition and structured programming, as reflected in [Par71, Par72a, Par72b] and [DDH74]. The only language supporting these notions directly was Simula [Dah68]. Programming languages, program verification, and formal specification all evolved from essentially this same context. CLU [Lis77], Alphard [Wul76], and most recently Ada [Ich79], are good examples of the results of language research; [Lon75] provides a good survey of the direction and results in verification; [Gut78] and [Gut80] provide surveys of the status of formal specifications.

Although in contemporary research, languages, specification, and verification seem more closely allied to each other than to operating systems, it is interesting to observe their reconvergence. The interested reader may wish to consider the interaction of abstract data types and protection [Jon76], and the use of verification in building secure systems [Wal79].

CHAPTER

FOUR

FUNDAMENTAL CONCEPTS

In this and the following chapter we will attempt to make the philosophy espoused in the last chapter more concrete. First, we will discuss most of the basic concepts on which Hydra is based and define the technical meaning of these concepts within the Hydra framework; then, in the next chapter, we will discuss the actual mechanisms available to the Hydra user.

In some respects the facilities of Hydra are quite different from those of more traditional operating systems, and this has presented some problems to people initially trying to learn about the system. This problem is, unfortunately, aggravated by the apparently circular nature of the definition of the most basic notions in the system—and a firm grasp of the notions is prerequisite to an understanding of the system.

Fortunately, in large measure the concepts of concern are simply the analog, in the operating system domain, of familiar concepts in programming languages—variables, types, subroutines, and so on. Indeed, one reasonable intuitive image of Hydra is that it is just the "run time support system" for a data abstraction language of the type discussed in the last chapter. Thus, as both a gentler introduction to these concepts and to help avoid the apparent circularity, we will first present a short lexicon of Hydra terms and draw analogies between them and their analogs from programming languages. The subsequent sections will then give a more precise, if somewhat terse, definition.

Object. The analog of a *variable* in programming languages; an object is the abstraction of a typed storage cell. It has a "value" or "state." Often the representation of an object will be constructed from a number of other objects; in this sense an object strongly resembles a "record" in a programming language.

Type. The analog of the notion of *type* in programming languages; the major difference is that typed objects, and hence types, persist longer than a single program execution. As in a programming language, the type of an object determines which operations may legally be applied to it (that is, type checking is performed.)

Capability. The analog of a *reference*, or *pointer* in programming languages; the major difference is that a capability, in addition to pointing to an object,

contains protection information.

Local name space (LNS). The analog of an *activation record* in programming language implementations. A local name space, or LNS, contains (capabilities for) the local objects (i.e., "variables") of a procedure invocation. Thus, an LNS defines the "environment" of a procedure invocation; only those objects in the LNS, or reachable from it via a path of capabilities (with appropriate rights), are accessible to the invocation.

Procedure. The analog of a *procedure*, or *subroutine*, in programming languages. As in programming languages, we make a distinction between a procedure and its invocation. A procedure is a static entity; the invocation of a procedure is an LNS and is the executable entity. Procedures are reentrant and may be recursive.

Templates. The analog of a *formal parameter specification* in a subroutine; the major difference is that all parameter checking is done during program execution, so a full description of the formal specification must be available at procedure call time. The *template* is this run-time specification.

The Call mechanism. The analog of a *subroutine call* in programming languages. The primary difference is that a Hydra *Call* operation involves a (complete) change in the protection domain. By contrast, programming languages typically provide only the protection enforced by their scope rules.

Although there are many similarities between these notions and their analogs in programming languages, and one can exploit that similarity to aid initial understanding, one must also be cautious. The concepts are *not* identical. Most of the differences arise from the fact that objects maintained by an operating system are likely to be "long-lived"; that is, they are likely to persist longer than the program that created them. By contrast, the variables of a single Algol- or Pascal-like program do not exist beyond its execution.[1] The implications of this difference are significant, since it means that other information, such as the type, must also persist and cannot be confused with other types from other programs.

4-1 OBJECTS

The abstraction of an instance of a resource, whether physical or virtual, is called an *object*. An object, for the present, may be thought of as a triple:

(unique-name, type, representation)

Every object has a *unique-name*, a name that differs from that of any

[1]Languages like LISP and APL also retain long-lived objects in their "work space" and thus are more like operating systems in this respect.

other extant object, any object that existed in the past, or will exist in the future.[2]

The *type* of an object defines the nature of the resource represented by the object. In general many objects will be of the same type, each being a specific instance of that kind of resource. Thus, the type attribute partitions the universe of extant objects into a set of equivalence classes. Some examples of types might be FILE, SEQUENTIALFILE, RANDOMFILE (various kinds of files), PAGE, CATALOGUE, PROCESS, SEMAPHORE, and so on.

The *representation* of an object contains its actual information content, e.g., a sequence of bytes in the case of an ASCII file. We shall have more to say about the representation of an object later.

4-2 TYPES

It should be clear from the preceding chapter that we do not know *a priori* what types of objects will exist. In fact we wish to permit, indeed encourage, the dynamic creation of new types. In addition, we do not know how long a particular object will exist and, hence, how long objects of its type will exist. Therefore, this section describes how an object's type attribute is represented, and, in particular, how this representation caters to the potential for long-lived objects and types.

The type attribute of an object is in fact the unique-name of another object—this latter object serves as a distinguished representative of the equivalence class of objects with the same type. Of course this representative object must itself have a type attribute; we demand that this be the special unique-name which we will call *$Type$*.[3] Initially the system requires a single distinguished object whose name and type are both *$Type$*. Figure 4-1 illustrates a situation in which three types (FILE, PAGE, and SEMAPHORE) have been defined.

To create a new object, a user invokes a kernel-defined operation, *Create*, and specifies the object's type. (The precise mechanism will be discussed in the next chapter.) A user may create a whole new type of object by invoking *Create* and specifying that the type of the new object is to be *$Type$*. The object returned will serve as the representative of the new class of resources.

As mentioned above, this particular mechanism for representing types is

[2]The unique-name of an object is a 64-bit value, obtained from the master clock (see Chapter 2).

[3]We are using this dollar-sign notation to emphasize that the *type names* are actually unique 64-bit integers. Each type also has a readable *print name*, but this is in all cases for convenience only. Two types may have the same print name, but they can never have the same type name. In most cases we will use the print names of object types, in small capitals for emphasis, such as TYPE or SEMAPHORE, when no confusion will result. The reader is urged to keep this important distinction in mind.

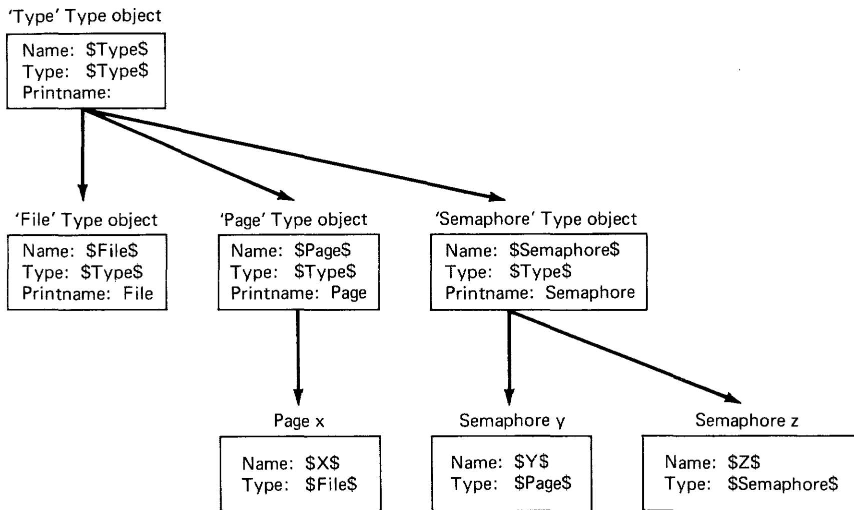

Figure 4-1 Type hierarchy

only one of several which might have been used. Its details are far less important than the property that new types may be created at will. An additional desirable consequence of this technique will, however, be discussed later (see *TypeCall* in Section 5-3.4).

4-3 CAPABILITIES

A *capability* is a pair

(unique-name, allowed-rights)

Intuitively, a capability consists of a reference to an object together with a list of *access rights* (allowed rights) to that object. For now, access rights may be considered to be a list of the operations that the possessor of the capability may legally apply to the object named by the capability.

Hydra departs from other capability-based protection systems by dividing the set of access rights into two disjoint subsets: the *kernel rights* and the *auxiliary rights.* The kernel rights apply to the *generic* (type-independent) operations (e.g., *Create*) provided by the kernel. The auxiliary rights apply to the operations defined on a particular object type by a user-level "subsystem."

It should be noted that the representations of capabilities are manipulated *only* by the kernel, and all objects must be accessed through capabilities. It is impossible to "forge" a capability, or to gain access to an object without

having a capability for it. (In particular, knowing the unique-name of an object will not help.)

4-4 REPRESENTATION OF OBJECTS

The concept of an object should be powerful enough that users may define new types of objects for new kinds of resources. To do this, one must be able to store various information in the object. (For example, a file object may contain the disk address of the contents of the file.)

In many capability-based operating systems a capability is an attribute of executors only (e.g., processes). In such systems the set of capabilities possessed by an executor defines its protection domain. While this is also true in Hydra, we have generalized the notion of objects and capabilities in an important direction. Capabilities are not attributes of executors alone; any object may contain capabilities for other objects. (Among other things this permits us to close the circle and define executors as merely a particular type of object.) The most important practical implication of this generalization is that new object types (new kinds of resources) may be defined in terms of existing types.

By analogy with programming languages, a Hydra object is a *record*. It is a heterogeneous collection of simple variables (data) and pointers (capabilities for other objects). For implementation reasons, the representation of an object is divided into two parts: a *data-part* and a *C-list* (i.e., a capability list). The data-part of an object is merely a block of storage that can hold subsystem-specified data. The kernel capability mechanism places no interpretation on this data, although presumably the subsystem that defines the object type does.[4] The C-list of an object contains an ordered set of capabilities as defined above. Thus any object may reference other objects. Either the data-part or C-list of an object may be empty.

It should be noted that the C-list of an object allows one to construct a general directed graph structure. The objects themselves are the nodes in this graph. The capabilities are the arcs; each arc is labeled with the access rights permitted to an object when it is referenced via that particular arc. We will often exploit this analogy with graph structures to draw diagrams representing a collection of objects. As an example, consider Figure 4-2, in which objects are shown as rectangular boxes, the top of which denotes the data-part and the bottom of which denotes the C-list; the objects pointed to by the capabilities of the C-list are indicated by the directed arrows coming from the C-list.

[4]A few object types are defined by the kernel; most of these will be discussed later. For these, the defining subsystem is a part of the kernel, and the kernel *does* place an interpretation on the contents of the data-part.

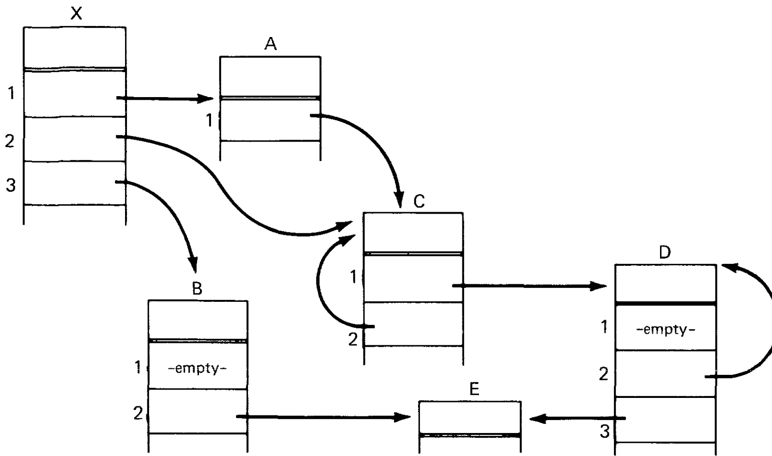

Figure 4-2 Example object graph

Example Let's consider a simple example of how the representation of an object can be used. The basic Hydra system does not provide the notion of a "directory" or "catalogue"; suppose that we wished to introduce this notion. (Alternatively, suppose that the notion had been defined by someone else, but their definition was unsatisfactory for our purposes and we therefore wished to create an alternative definition.) The function of a directory is to map external character string names, e.g., "XYZ," onto specific objects. More accurately in Hydra, a directory maps external names into *references* to objects, that is, to capabilities.

A natural representation of directories suggests itself. First we create a new object type, DIRECTORY. Then we build a number of user-level programs to provide the usual kinds of directory manipulation operations, e.g., "search," "insert," "delete," "rename," and so on. (We must defer discussing how the programs are created. For the present the important thing is that these programs must share certain assumptions about the structure of DIRECTORY objects.)

We might choose any of several representations for directories, but many of them will involve storing the external names in the data-part of the directory object and the associated capabilities in its C-list. A few possibilities are listed below.

1. If we are willing to restrict external names to a fixed length, say n bytes, then the simplest representational scheme is to store the capabilities in C-list slots 1,2,3,... and store the name corresponding to the ith capability in bytes $(i\text{-}1)\cdot n+1$ through $i\,n$ of the data-part.
2. The previous scheme seems to imply a linear scan of the data-part in order to implement the "search" operation; it could be easily modified to use a hash table or a discrimination net in the data-part.

3. If the restriction to fixed-length names is considered unacceptable, a more elaborate data-part structure is necessary. Specifically, the one-to-one correspondence between C-list slots and displacements in the data-part no longer holds and information must be recorded with each name which specifies the associated C-list slot number.
4. If a "delete" operation is provided, some scheme for allocation of free storage is needed, both for the data-part and for slots in the C-list. If there is a 1-1 correspondence between the C-list slots and displacements in the data-part, a single mechanism will suffice. If not, separate space management schemes must be implemented.

Before leaving the example, we would like to point out a property of these schemes that might not be obvious from this brief description. The directory systems of most operating systems map external names into references to files. Indeed, often the coupling is so tight that no distinction is made between the "file system" and the directory system—there is no opportunity for the user to name anything other than files. That restriction is *not* true here. This directory system maps external names to capabilities, which may reference files, but may also reference pages, semaphores, or any other type of object. Specifically then, it may map to capabilities for other directories. Hence, the familiar "tree-structured directory" is naturally accommodated by this structure.

4-5 THE LOCAL NAME SPACE

Up to this point we have described a relatively static view of Hydra and avoided a precise definition of the execution environment of programs. In this and the following sections we will begin to discuss the execution environment, or *domain*, of a program and how it changes dynamically.

LNS (for "local name space") is one of the object types recognized and maintained by Hydra. An LNS object defines the *instantaneous protection domain* of a program. That is, the C-list of an LNS contains capabilities for objects that a program may reference. In fact, *all* the objects referenced by a program must be referenced through its LNS. However, since the objects referenced by the capabilities in an LNS may themselves contain capabilities for objects which contain capabilities, etc., the set of objects available to a program is the transitive closure of the capabilities found in its LNS. As will be discussed later, the formation of this closure is restricted by the access rights in the relevant capabilities, but it should be clear that any object not in the closure is inaccessible.

In any capability-based protection scheme it is of paramount importance that the capabilities provide the only mechanism for gaining access to objects. In particular, knowledge of the unique-name for an object does not grant access to the object. To enforce this essential property within Hydra, all

objects are named by a *path* rooted in the current LNS of a program.

To illustrate this point, refer to Figure 4-3, in which we have added two LNSs to the object structure of Figure 4-2. Each object in the figure can be named in various ways, depending on the LNS which originates the reference. Table 4-1 lists the possible reference forms. Note that because the capability graph is not strictly tree-structured there can be several path names for the same object from the same LNS.

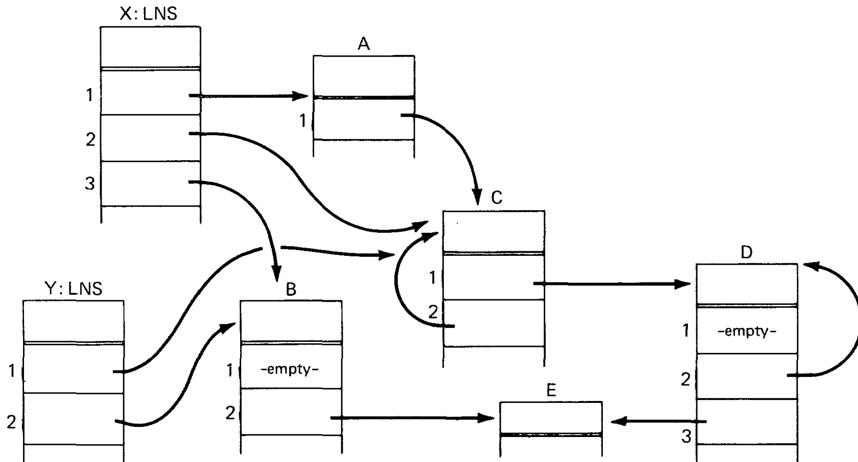

Figure 4-3 Objects in the domain of two LNSs

Table 4-1 Naming objects from two LNSs

Object	Path name from X	Path name from Y
A	Path(1) (or just '1')	Cannot be named
B	Path(3) (or just '3')	Path(2) (or just '2')
C	Path(2) or Path(2,2,...) or Path(1,1) or Path(1,1,2,2,...)	Path(1) or Path(1,2) or Path(1,2,2,...)
D	Path(1,1,1) or Path(2,1) or Path(2,1,2,2,...)	Path(1,1) or Path(1,1,2,2,...)
E	Path(3,2)	Path(2,2)

Paths are also used to name capabilities as well as objects, depending on the context in which they are used. Thus, while Path(1,1) and Path(2) both

name object *C* from LNS *X*, the paths name two different capabilities for *C*. This is important because the capabilities may have different sets of access rights for *C*.

Note that the unique-name of an object is never used to name it. The term "LNS" was originally chosen to emphasize its function as a mapping from local names in a single program to globally unique names.

4-6 PROCEDURES

PROCEDURE is another type of object defined by the kernel; it serves as a schematic from which an LNS is formed when the procedure is "called." The procedure is simply an abstraction of the intuitive notion of procedure or subroutine; that is, a procedure has some "code" and some "data" associated with it. It may be called and may accept parameters. It is reentrant and potentially recursive. Hydra's procedures go beyond this simple model by including protection facilities, as we shall see shortly.[5] To simplify for a moment, the *Hydra procedure call* (*Call*) is a kernel function that accepts a capability for a procedure, creates an LNS object, copies the C-list of the procedure into the C-list of the LNS, binds parameters, and transfers control to the code of the LNS (procedure). The "old," or "calling," LNS is stacked so that the complementary kernel function (*Return*) can destroy the new LNS and return control to the old one.

Thus procedures provide the initial state of an LNS, or equivalently, an initial protection domain. Note that a complete change in execution environment may occur when a procedure is called; in particular, an LNS does not automatically inherit access to any of its caller's environment. This fact is the basis of the observation that Hydra does not provide a hierarchical protection structure; although one may implement a hierarchy if one chooses, the system does not force subsystems to be more privileged than their callers.

The distinction between a procedure and an LNS is an important one, even though it is frequently convenient to blur the difference (thus we may speak of the "currently executing procedure" when we really mean an LNS created from that procedure). An LNS may change during the course of its execution, for example by creating new objects and storing capabilities for them into its C-list. The procedure from which the LNS was created is not affected by the execution; thus procedures are potentially reentrant and

[5]The term "procedure" in this context has unfortunately misled some people, suggesting to them a small unit of computation. Since the software implementation imposes a considerable overhead, a Hydra procedure is impractical for implementing simple subroutines such as sine or cosine. Although we can imagine hardware/firmware support that would make it practical for even the smallest subroutines to be protected procedures, with the present implementation a Hydra procedure is used only when a change in protection domain is desirable or required.

recursive.[6]

4-7 PROCESSES

PROCESS is another kernel-defined object type. As in most systems, a process is the smallest independent unit which may be scheduled for asynchronous execution. Technically, a process is simply a stack of LNS objects, the top one of which defines the current protection domain of the process. The stack is altered by calls and returns, as described above.

A PROCESS object also contains information which controls various policy decisions (e.g., scheduling), but we shall defer explanation of this aspect of processes until Chapter 12.

4-8 PROCEDURES AND ACCESS RIGHTS

Up to this point we have been intentionally vague about the precise meaning of the words "access right"; in fact we have not defined what it means to "access" an object. The remainder of the chapter will make these notions more precise, but we will motivate that presentation here.

Procedures might be used for any of several reasons, some of which are quite pedestrian. The natural way, for example, to implement a compiler in the Hydra context is as a procedure. Such a procedure would need to be called with, for example, a capability for FILE object containing the source text to be compiled. Although there may be many useful procedures such as compilers, loaders, etc., they are not the kind with which we are presently concerned.

A more important use of the procedure is in implementing the notion of an *abstract type definition*, which was introduced at the beginning of this chapter. Recalling that discussion, we recognize that two things are required in such a type definition: a definition of the representation of the new type and a definition of the operations on instances of the type. The data-part and C-list of objects provide the primitive tools for representing new virtual resources. Procedures are used to define the operations.

In Hydra, the things to be protected are objects, and they are to be protected against the unauthorized application of operations. Using the current terminology, protection is enforced at procedure invocation time. The protection mechanism validates that it is legal to call this particular procedure with the given actual parameter capabilities.

A "subsystem" is, in fact, nothing more than a collection of procedures,

[6]To ensure reentrancy, however, the author of a procedure must exercise some care. The LNS may, through inherited capabilities (explained later), alter objects named in the procedure, which would thus alter all LNSs subsequently created from the procedure.

each of which implements some operation on a specified object type. A file subsystem, for example, might consist of a set of procedures such as *Read*, *Write*, *Append*, *Rewind*, etc. If a user happens to have a capability for a specific file object, he may attempt to apply one of these procedures to it. The protection mechanism is embedded in the procedure call mechanism; it must verify that the procedure that the user is attempting to invoke is among those permitted by the access right part of the file capability.

It is not the case, however, that the caller of a procedure must supply a capability which has all the access rights needed by the procedure. In general a user who possesses a capability for an object, e.g., a file, will *not* have the right to access the representation of the object. It would be most unwise, for example, for the possessor of a file capability to manipulate the physical disk addresses stored in the corresponding object. On the other hand, the procedures in the file subsystem must have access to this information when they are invoked to perform some operation on behalf of the user.

This is one example of a case in which a procedure, in order to do its job, needs *more* access rights than its caller. The situation is a common one for procedures that, as part of a subsystem, implement a resource. To accommodate this common circumstance, the kernel allows *rights amplification*, an increase in access rights, when a capability is passed as a parameter. Although the amplification of rights can be allowed only under tightly controlled circumstances, it is an extremely important attribute of the Hydra protection mechanism and crucial to the goal of allowing operating system facilities to be defined at the user level.

4-9 TEMPLATES AND THE MERGE OPERATION

In order to describe the procedure call mechanism, we must first introduce another concept—that of a *template*. There are, in fact, three kinds of templates recognized by the kernel:

Creation templates
Simple templates
Amplification templates

A creation template is used exclusively to create a new instance of objects of a given type; we will discuss creation templates in the next chapter. A simple template is just a pair:

(type, required-rights)

An amplification template, on the other hand, is a triple:

(type, required-rights, new-rights)

where *type* specifies an object type, and *required-rights* and *new-rights* are sets of access rights.

The relation between simple and amplification templates parallels the two uses of Hydra procedures. Procedures implementing subsystems need amplification templates to expand the access rights of parameter capabilities passed to them. Other procedures (e.g., compilers) can make do with simple templates because they do not need the rights amplification facilities.

Since, in the present discussion, we shall not be concerned with creation templates, we will use the word *template* to mean either of the other kinds and hence something of the form shown above. In some cases, we shall also refer to these two forms as *parameter templates* where we wish to emphasize that they are not creation templates.

A template may occupy a slot in the C-list of an object, but it is *not* a capability. It may be thought of as a specification for a capability that will eventually occupy that slot. Templates are important because of a kernel-supplied operation called *Merge.*

Merge takes two parameters, a template and a capability, and if successful, returns a capability. Specifically, it does the following sequence of operations:

1. It verifies that the type specified in the template matches the type of the object named by the capability. If they disagree, the operation fails.
2. It verifies that the access rights specified by the capability are a superset of the required-rights as specified in the template. If this is not the case, the operation fails.
3. It forms a new capability and returns it. This capability will name the same object as the parameter capability. Its allowed-rights field is constructed as follows:

   a. If the template is a simple template, then the resulting allowed-rights field contains the allowed rights of the parameter capability.
   b. If the template is an amplification template, then the resulting allowed-rights field contains the new-rights field of the template. As will be discussed in the next chapter, however, certain rights can never be gained through amplification; these permit a user to protect himself from certain subsystem behaviors.

Thus the *Merge* operation performs a dynamic type and rights check and potentially creates a new capability for some object. In addition, however, it sets the allowed-rights field of the new capability to either that of the original capability or to that of the new-rights field of the template, depending on whether the template was a simple or amplification template.

4-10 THE CALL MECHANISM AND RIGHTS CHECKING

The *Call* operation, as explained previously, creates an LNS object from a procedure. The major omission from that discussion was the handling of parameters. We can now present the full definition, relying on the previous discussion of *Merge*.

Call first creates an empty LNS object and then copies the information from the parent procedure into the LNS object. In the process of copying the C-list of the procedure, it may encounter either capabilities or templates. In the former case the capability is merely copied; these are known as *inherited* capabilities. In the latter case a *Merge* is performed between the template from the procedure and a parameter capability supplied during the *Call* operation. The capability resulting from the *Merge* is stored into the C-list of the LNS object. Should any of these *Merge* operations fail, the entire *Call* will fail.

As can be seen from this discussion, templates serve somewhat the same role as formal-parameter specifications in a programming language. They allow the designer of a procedure to specify both the type and the rights that a parameter must have. In addition, in the case of a subsystem procedure, the "new-rights" may be used to specify rights amplification.

The *Call* mechanism (including the associated merging of templates and capabilities) is essential to the Hydra protection mechanism; it is the major point at which protection is checked. Thus it is worth reviewing the action of *Call*:

> A procedure may contain *templates* in addition to the usual collection of caller-independent capabilities. Templates characterize the actual parameters expected by the procedure. When the procedure is called, a new LNS is created. The slots in this LNS that correspond to templates in the procedure's C-list are filled with "normal" capabilities derived from the actual parameters supplied by the caller. This "derivation" is, in fact, a *Merge* operation; the template defines the checking to be performed. If the caller's rights are adequate, a capability is constructed in the (new) LNS referencing the object passed by the caller and which contains rights formed by merging the caller's rights with the rights specified in the template.

4-11 A NOTE ON IMPLEMENTATION

It might seem incongruous to inject a note on implementation at this point; in this case, however, the nature of the implementation has a significant impact on the conceptual issues we are discussing. The point at issue is the representation of the allowed-rights field in a capability, and the required- and new-rights fields of templates.

We mentioned before that all access rights (in all these fields) are broken into two subfields: *kernel rights* and *auxiliary rights*. The kernel rights refer to the generic operations provided by the kernel (e.g., *GetCapa*). The auxiliary

rights refer to the type-specific operations.

The relevant point is that both these subfields are simple bit vectors. (In the current implementation the kernel-rights field is 16 bits and the auxiliary rights field is 8 bits, although these sizes are arbitrary.)

Thus, let

C be the capability parameter to *Merge*
T be the template parameter to *Merge*
NC be the new capability returned by *Merge*

and let

C.type be the type of *C* (similarly for *T.type*, *NC.type*)
C.allowed be the allowed-rights field of *C* (similarly for *NC.allowed*)
T.required be the required-rights field of *T*
T.new be the new-rights field of *T*
amptemp(T) be a function that is *true* if and only if *T* is an amplification template

The *Merge* operation is:[7]

```
if C.type ≠ T.type then ERROR else
   begin
      NC.type := C.type;
      if BitVectorAnd(C.allowed, T.required) = T.required
         then NC.allowed := (if amptemp(T)
                  then T.new
                  else C.allowed)
         else ERROR;
   end;
```

As can be seen, this is a simple, fast operation. However, the important point from a conceptual viewpoint is that

- The *Merge* operation does not place an interpretation on the meaning of the auxiliary rights bits.
- There does not need to be a one-to-one correspondence between the rights bits and the operations on the type (though there can be one if the subsystem designer so chooses).

Since the rights check is performed only after the type check has been made, a subsystem designer is free to choose the interpretation of these bits as is appropriate for the particular resource he is defining. In particular, there may

[7]This definition is incomplete. The full explanation is given in Chapter 7. The only simplification here is that, in reality, certain rights cannot be gained through amplification.

be one bit for each possible operation, or certain operations may simply require a specified combination of other rights.

Example Suppose one is defining a file system and that three of the operations to be provided are "read," "write," and "update" (i.e., both "read" and "write" during a single "open" period). These three operations are to be implemented as procedures Read, Write, and Update; each procedure will require (at least) a capability for the file to be passed as a parameter.

The file (sub)system designer must choose which bits of the (auxiliary) rights field are to have what meaning. We will consider two schemes, either of which may be appropriate:

Scheme 1		Scheme 2	
Rights bit	Meaning	Rights bit	Meaning
C.allowed[0]	read access allowed	C.allowed[0]	read access allowed
C.allowed[1]	write access allowed	C.allowed[1]	write access allowed
C.allowed[2]	update access allowed		

Under the first scheme each of the three procedures will need a template for the parameter capability, and the type field of all these templates will specify that the parameter must be of type FILE. The required-rights field of each template, however, will be different:

Scheme 1	
Operation	Required-rights
Read	T.required[0] = 1, all others zero
Write	T.required[1] = 1, all others zero
Update	T.required[2] = 1, all others zero

In the second scheme everything will be the same except for the required-rights field of the template for Update:

Scheme 2	
Operation	Required-rights
Read	T.required[0] = 1, all others zero
Write	T.required[1] = 1, all others zero
Update	T.required[0] = T.required[1] = 1, all others zero

In short, under the second scheme we do not have a separate right associated with updating—we merely require that the caller have the right to both read and write the file.

4-12 PROTECTION VS. FLEXIBILITY

Flexibility and protection are closely, but not inversely, related; that is, more protection does not necessarily imply less flexibility, or conversely. We believe that protection is not merely a restrictive device imposed by "the system" to ensure the integrity of user operations, but is a key tool in the proper design of operating systems. It is essential for protection to exist in a uniform manner throughout the system and not to be applied to only specific entities (e.g., files). The idea of capabilities is most important in the Hydra design; the kernel provides a protection facility for all entities in the system. Protection includes not only the traditional "read," "write," and "execute" distinctions, but arbitrary protection conditions whose meaning is determined by higher-level software.

It is important in any discussion of protection to distinguish carefully between "protection" and "security." In our view, protection is a mechanism; security is a policy. A system utilizing a protection mechanism may be more or less secure, depending upon policies governing the use of the mechanism (for example, passwords and the like are policy issues) and upon the reliability of programs that manipulate the protected entities. Thus the design of the Hydra protection mechanism provides a set of concepts and facilities on which a highly secure system may be built, but does *not* provide that security inherently.

The particular extensible, capability-based protection system chosen for Hydra was picked because of its ability to allow the construction of user-visible operating system facilities as normal user programs. It also happens that a broad spectrum of security policies can be implemented in terms of the Hydra mechanisms (see [Jon75] and Chapter 7 of this book). Thus the Hydra mechanisms are interesting in their own right; however, the ability to extend the system at the user level is, in our minds, its greatest virtue.

4-13 RETROSPECTIVE

It may be that one of Hydra's most important contributions will be the philosophies and concepts presented in the last two chapters. Certainly, these are the things we believe should be emulated, albeit in evolved forms, in future systems. Even with the perspective of several years, the model is still both elegant and practical.

We will discuss the use of the Hydra mechanisms at length in subsequent chapters. It is worth a small peek ahead, however, to note here that the Hydra mechanisms have indeed fulfilled their goals. One *can* add new abstract types with ordinary user code. The facilities added in this way *do* assume an equal status with pre-existing ones. Hydra *does* routinely run with

several coexisting subsystems defining essentially the same facility—several directory systems, for example. These multiple subsystems *do not* interfere with each other, and no special provision is made to ensure this. Moreover, we have found construction of these systems to be remarkably easy. The construction of subsystems is substantially simpler and less error prone than the construction of similar facilities in the conventional way.

To those familiar with modern notions of data abstraction in programming languages, these assertions may not seem so remarkable—they are, after all, current doctrine. However, the Hydra model was developed contemporaneously with the notion of data abstraction in programming languages, and its implementation substantially predates everything except Simula. The model suffers only slightly from not having the benefit of previous models to build on. In the light of subsequent developments we might have adopted a stronger grouping of the procedures that implement the operations on an abstraction, for example, as is done in the CAL sytem [Stu74] or in Modula [Wir76]. We might also have associated mutual exclusion with the operations on an object as is done with monitors [Bri78].

The full generality of amplification templates provides fine-grain control; subsystem procedures need to gain only those rights that they require to perform a given operation. In practice, however, it is common for most subsystem procedures to grant themselves all rights. While there is no logical need for this, it is simpler than thinking about what is needed. Moreover, it is not clear that this "over-amplification" is bad; the user must trust the subsystem to perform as specified in any case.[8] Although the *Merge* operation is not especially complicated, it would be even simpler if amplification implicitly granted all rights. We could, in that case, simply elide the notion of "new rights" from amplification templates. Other systems, such as STAROS [Jon79], do this and we would probably do the same in a future system.

In retrospect, the *principle* of policy/mechanism separation seems unassailably sound. At least in some cases, it also works well in practice; the Policy Modules in Hydra do, in fact, control medium-term scheduling. Several PMs have been built and can run simultaneously. Significant differences in performance result from the use of the different PMs. Policies for establishing what is meant by a "user," for handling user-authentication, for controlling the resources of a "job," and so on, have also been successfully factored out of the kernel. On the other hand, we have never been able to find a clean model for separating mechanism and policy in those cases where a resource is shared. Secondary and primary memory are two clearly essential resources for which separation was not cleanly achieved; in the case of the

[8]However, we will show in Chapter 7 how the mutual suspicion of subsystem and user *can* be accommodated.

disks, in fact, policy control of shared disks was retained in the kernel.[9]

Perhaps one of our greater failings as a project is closely tied to the philosophy of the system. There are a number of manifestations of this failing, but they all relate to the fact that we all thought that the goals were *just right* and the mechanism *so elegant* that we became preoccupied with them to the exclusion of other important aspects of the system. The user interface, and predominantly the command language interpreter, was never well thought out, for example. Because the command language was not an integral part of the system—and "any user can build his own easily," or so we told ourselves—it never seemed worth the effort. To this day, we are absolutely convinced that the most friendly, elegant interface imaginable can be "easily" constructed for Hydra, but it was not done!

A corollary to the previous problem was our choice of general-purpose, time-shared computing as the target use of the system. There was simply too much additional software required for the general user—debuggers, editors, loaders, compilers, and so on. Although much of this eventually got built, it was too late. We would have been better off, and probably would have learned more about multiprocessors as well as operating systems, if we had focused on a narrower application domain. This *does not* mean that either C.mmp or Hydra was unsuited to the larger domain—only that we frail designer/programmers could not produce all the software it needed.

Finally, on reflection, we find it strange that processes did not play a more major role in our design of the kernel. We were, after all, constructing a system for a multiprocessor. Moreover, process-structured operating systems were very popular at the time. Nonetheless, although we supported them at the user level, they were given only passing consideration as a structuring tool for the operating system itself. With only a few exceptions that will be noted later, this has turned out to be a workable, if occasionally regretted, decision.

4-14 FURTHER READINGS

The concepts of protection, sharing, and information flow in computer systems have been the subjects of active research for over a decade. As a result, many of the fundamental concepts described in this chapter have appeared in other forms in systems before and after Hydra. Dennis and Van Horn [Den66] are generally credited with the first description of capabilities as a mechanism for controlling access. The other canonical method of representing protection information, access control lists, is best known from its use in Multics [Sal74]. Although the formal equivalence of these two representations with respect to information content is well-known [Lam74],

[9] User-level programs can acquire an entire disk, in which case they can also control policy.

the architectures built on top of them have rather different specific behavior— [Sal75] contains an extensive survey. Many conventional operating systems have offered authority-based protection [Bob72], [Amd64] (generally applied to files), and protection systems based on capabilities have been extensively developed [Jon73, Fab74, Fer74]. These different approaches have prompted considerable investigation into the practical use of protection mechanisms in programming languages [Mor73] and systems (see [Eng74] and other contributions to the 1974 IRIA Workshop). The construction of systems intended for everyday use has exposed strengths and weaknesses of capability-based approaches [Lam76, Wil79]. Protection systems have also been evaluated on their ability to provide a secure and reliable computing environment; [Lin76] is a good survey.

CHAPTER

FIVE

KERNEL FACILITIES

In this chapter we will discuss the basic facilities provided by Hydra for the manipulation of the abstractions introduced in the last chapter: objects, capabilities, and so on. It is neither practical nor particularly enlightening to discuss *all* the facilities provided, but we will cover:

1. The *generic facilities* of the kernel. These include:

   a. The set of operations that can be applied to any object or capability, regardless of its type.
   b. The set of kernel rights. These rights must be present in a capability for the various generic operations to be applied to it (or, in some cases, to the object it names).

2. The *kernel-defined object types*, along with their type-specific operations and auxiliary rights. A few of these (e.g., LNS and PROCEDURE) were mentioned in the last chapter; these types are critical concepts in Hydra. Other kernel-defined types (e.g., PORT) could have been implemented by user-level software but were included in the kernel for efficiency reasons; these types are not central Hydra concepts, and discussion of them is deferred until later chapters.

5-1 NOTATION

In the last chapter we use diagrams such as that shown in Figure 5-1 to illustrate a collection of objects. Each object is represented by a rectangular box. The upper portion of a box (above the double horizontal line) denotes the data-part of the object; the lower portion denotes the C-list. The C-list is divided into a number of rectangles (often called "slots"), each of which may hold a capability; arrows from these C-list slots to other objects show which objects are named by the capabilities. The rights in a given capability, or at least those of interest, are listed in the box representing the slot; for emphasis, we sometimes show a right crossed out to emphasize that it is not present in a given capability. Occasionally, as in *Me:LNS*, we place a mnemonic name (*Me*) and type (*LNS*) just above the rectangle representing

a specific object; this is merely a convenience for talking about the objects. As we noted in the last chapter, an execution environment consists of an LNS together with the set of objects referenced (possibly indirectly) by the LNS. Thus, for example, *A* and *B* are part of the execution environment of *Me*, but *C* is not.

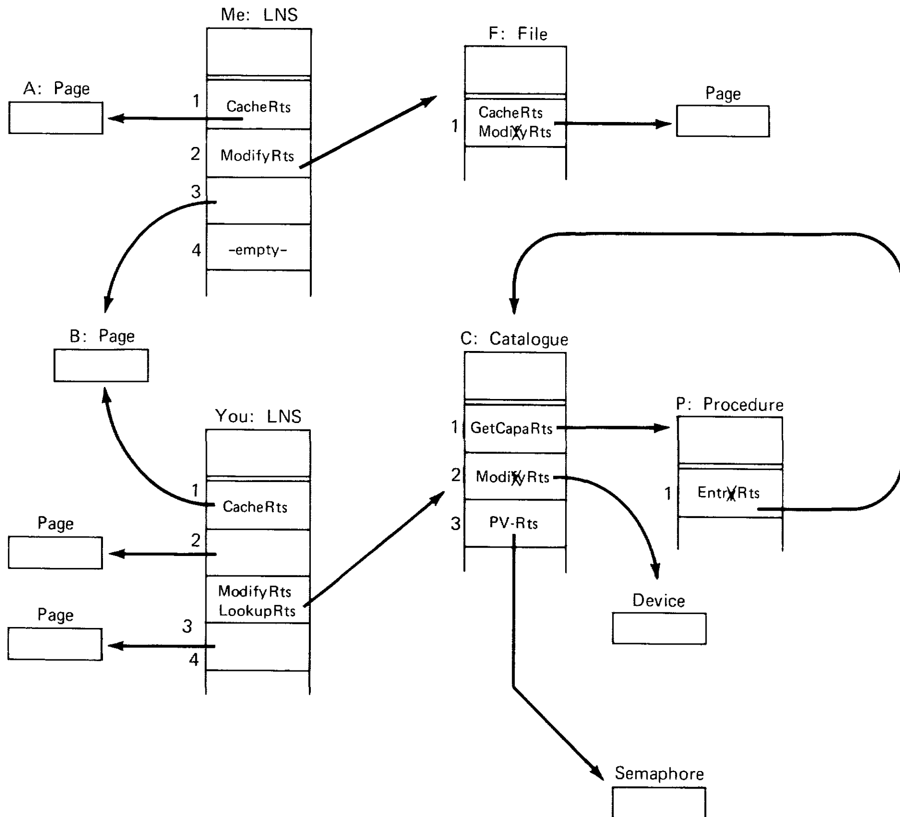

Figure 5-1 A Collection of Objects

Note that we have specifically shown the capability "slots" in the figure, and indeed have numbered them beginning with 1. In practical programming, both the data-part and C-list of an object are extremely important. In order to simplify discussions we will introduce some special notation.

Definition 1 If X is an object, then

X^d will denote its data-part
X^c will denote its C-list

$X^d[i]$ will denote its *i*th word of its data-part (origin 1)
$X^c[i]$ will denote its *i*th capability (origin 1)

Thus, in Figure 5-1, $Me^c[2]$ is the capability that names the file *F*.

In order for a program to name capabilities or objects in its execution environment, it specifies a *path* to the object or capability, where the path must be rooted in the program's LNS. More formally, a path specification is of the form

$$Path(a,b,\ldots,y,z)$$

and is defined recursively as follows:

Definition 2 *Path(a)* is the capability in the *a*th slot of the LNS and is equivalent to the simple index *a*. *Path(a,b,...,y,z)* is the capability in the *z*th slot of the C-list of the object referenced by the capability *Path(a,b,...,y)*.

Because we must be able to talk about the rights possessed by objects along a path, we will call the capabilities *Path(a)*, *Path(a,b)*,..., *Path(a,b,...,y)* the *steps* and the capability *Path(a,b,...,y)* the *pretarget*. (The pretarget is also considered one of the steps.) The final capability *Path(a,b,...,y,z)* is the *target*. Thus, for example, in Figure 5-1, Path(3,1,1) when interpreted in the context of the LNS named *You*, names the first capability of the object *P* (alternatively, it names the object *C* that is referenced by this capability). In this example $P^c[1]$ is the target, $C^c[1]$ is the pretarget, and $You^c[3]$ is a step.

5-2 KERNEL RIGHTS

Every capability contains a set of access rights that determine which operations may legally be applied to itself and to the object it names. The rights in a capability are divided into two groups:

Kernel rights. Those rights that control the application of the generic kernel operations (such as moving capabilities from one place to another).
Auxiliary rights. Those rights that are type-specific and are inspected only when a type-specific operation is attempted.

In this section we shall define the kernel rights. The auxiliary rights will be discussed in those sections that deal with specific object types. The kernel rights may be further subdivided into three groups:

Capability rights. Rights that apply to the capability. If these rights are missing, certain kernel operations cannot be applied to the target capability.

Object rights. Those rights that apply to the object named (referenced) by the capability. These may be further subdivided:

Data-part rights. Those rights that, if absent, will not permit operations on the data-part of the object named by the capability.

C-list rights. Those rights that, if absent, will not permit operations on the C-list of the object named by the capability.

Restriction rights. Those rights that, if absent, will prevent certain operations on any object that can be named, possibly indirectly, through this capability.

A full and complete definition of the kernel rights requires a bit of mental recursion. Strictly speaking, the meaning of each right is defined by the set of kernel operations that are allowed (disallowed) by the presence (absence) of the right. Alas, the definition of each of the operations, of course, also depends upon the rights that are present in its parameters. To break this circularity, Table 5-1 presents an intuitive description of the intent and function of each of the kernel rights. After reading this and the following section that defines most of the basic kernel operations, the reader would be well advised to review this section and compare the intuitive definitions of the rights with their use.

In Chapter 7 we will discuss how these access rights are used to solve various protection problems. In this chapter we will illustrate their use to support the construction of abstract data types as discussed in Chapter 3.

In any data abstraction facility, it is essential that only the operations of a given type can modify objects of that type. In order to modify the representation of an object, its user would need a capability with one or more of *PutDataRts*, *AppendDataRts*, *PutCapaRts*, *AppendCapaRts*, or *KillRts* (as well as *ModifyRts*). By removing all these rights from the capability, the subsystem can ensure that *no* modification can be performed by the users—and hence that only the subsystem procedures can effect changes. Alternatively, a subsystem may choose to permit the user to make certain modifications by setting the rights accordingly.

Recall another basic tenet of data abstraction—one should be able to hide the representation of the type being defined. Simply preventing modification of the representation is not enough; the user of the abstract type should know about its abstract behavior—its operations—but not about its representation. By hiding a representation we ensure that the user cannot develop dependencies on the representation, and thus we ensure that the representation can be changed if that is deemed a desirable thing to do.

Table 5-1 Kernel-defined access rights

Capability rights	
DeleteRts	Permits the capability to be deleted from the C-list that contains it.
EnvRts	("Environment rights.") Permits the capability to be stored outside the immediate environment, i.e., outside the LNS.
Data-part rights	
GetDataRts	Permits data to be copied out of the data-part of the object named by the capability.
PutDataRts	Allows data to be stored into the object named by the capability.
AppendDataRts	Allows data to be appended to the end of the data-part of the object named by the capability.
C-list rights	
GetCapaRts	Allows capabilities to be copied out of the C-list of the object named by the capability. In addition, *GetCapaRts* are required on all steps of a path.
PutCapaRts	Allows capabilities to be stored into the C-list of the object named by the capability.
AppendCapaRts	Allows capabilities to be appended to the end of the C-list of the object named by the capability.
KillRts	Permits capabilities to be deleted from the object named by the capability. Each capability to be deleted must, in addition, possess *DeleteRts.*
Restriction rights	
ModifyRts	Allows modification of the objects named by any path through the capability. For example, then, both *ModifyRts* and *PutCapaRts* must be present in the pretarget in order to store a capability into its C-list.
UncfRts	("Unconfined rights.") In capabilities for PROCEDURE objects, allows the procedure to be called "unconfined." Any call of a procedure lacking this right is termed a "confined call" and results in a "confined LNS." Confined LNSs cannot store information in such a manner that it can be accessed by another domain.
CopyRts	Allows a copy of the object named by the capability to be made.
CreateRts	Allows an object of the type described by a template to be created. This right has meaning only in "creation templates."

Notice that *GetCapaRts* are required along all steps in a path, including the pretarget. Further, *GetDataRts* are required in order to access the data-part of an object. Without these rights, it is impossible to name the components of an object. Therefore, the representation of an abstract type can be hidden from its user by the simple expedient of removing *GetCapaRts* and *GetDataRts* from capabilities for objects of the type. The subsystem that defines the abstract type must, of course, access the representation; it can do this, however, because it can amplify the rights in parameter capabilities to include these rights (see the discussion of templates and the *Merge* operation in Chapter 4).

Both of the previous examples (prohibiting modification and hiding the representation) rely on the ability to remove certain rights from all capabilities for objects of a given type. Actually this is easily done, and the mechanism for doing so accomplishes another objective of a data abstraction facility as well—namely, controlled initialization. Typically when a new abstract type is defined, the definer will not make "creation templates" (discussed below) publicly available; instead, he will provide a subsystem procedure that will create, and return a capability for, an object of the new type. This procedure will usually remove all C-list and data-part rights from the returned capability because these rights are inappropriate for the user to have. In addition, this "creation procedure" can initialize the data-part and C-list of the new object in type-specific ways. The ability to do this kind of initialization upon object creation is another essential property of a data abstraction facility.

5-3 KERNEL OPERATIONS

Hydra may be viewed as consisting of two parts—a kernel that implements the run-time support for subsystems (abstract data type implementations), and a collection of initial subsystems, without which it would be impossible, or grossly inefficient, to implement further subsystems. In this section we will define the generic operations that support subsystem construction; these operations are referred to as *Kalls* to emphasize both their similarity to, and their difference from, the procedure calls that invoke subsystem operations. The bulk of the section is simply a terse definition of the various Kalls, each presented in the following format:

SomeKall(D:slot, A:capa, B:capa(index,GetCapaRts), R:rights) returns X:integer

Here we will define the effect of the Kall, including any side effects, error conditions, etc. In addition, we will occasionally discuss special properties of the parameters.

The header line, "*SomeKall(...)*," provides the name of the operation and information about its parameters (information that doesn't fit the format is defined below the header line). The specification of each parameter is of the form

name: parameter-type

where *parameter-type* may be

integer — The corresponding actual parameter must be an integer.

mem [size] — The corresponding actual parameter must specify the address of a contiguous block of at least *size* words in primary memory.

rights — The corresponding actual parameter must specify a set of access rights; generally this set is used to restrict the rights in some capability.

slot (info) — The corresponding actual parameter must specify a path to an empty C-list slot. Wherever this form is used, the operation being described will store a capability into the slot.

capa (info) — The corresponding actual parameter must specify a path to a capability.

object (info) — The corresponding actual parameter must specify a path to a capability. The object named by this capability is involved in the operation, not the capability.

template (info) — The corresponding actual parameter must specify a path to a template.

In *SomeKall*, for example, the actual parameter that corresponds to *D* must specify a "slot," those that correspond to *A* and *B* must be capabilities, and *R* must specify a set of access rights. If a *returns* clause is present, as in this example, the Kall returns a positive integer value in addition to whatever effects it may have.[1]

The *info* is the most interesting (and complex) part of the specification of the parameters of most operations. Syntactically it is simply a list of informational items separated by commas and enclosed in parentheses. Generally the information will consist of the access rights in the target capability. There are some additional special cases, however:

1. In most cases an actual parameter capability can be named by an arbitrary path; in a few cases, however, the capability (slot) must be in the user's LNS—that is, only a single-step path is permitted. In these cases the special word *index* appears in the *info* (as in parameter *B* in *SomeKall* above).

[1]Any Kall may "fail" for a variety of reasons, particularly if a capability lacks sufficient rights. Kalls indicate failure by returning a negative integer value. We will ignore this detail and use the *returns* clause to indicate the data that the Kall will return if successful.

2. In some cases the rights in the pretarget as well as those in the target capability matter. In these cases we will either include "*pretarget(...)*" in *info* or discuss the situation following the header line. Similarly, if some right is required on all steps of a path, we will denote this by "*steps(...)*." Recall that the pretarget is considered a step; thus, if a right is mentioned in "*steps(...)*" it will not necessarily be repeated in "*pretarget(...)*."

There are also some conventions that we observe in the description.

1. Wherever a capability is named by a path, all steps along the path must possess *GetCapaRts*. Since this is a universal requirement, we will not repeat it in each operation specification.
2. Whenever an object is modified, *ModifyRts* are required in the capability for the object (the target) and in all steps of the path leading to that capability. Likewise, when a capability or capability slot is modified, *ModifyRts* is needed in the pretarget and in all steps of the path leading to the pretarget. Because this is a universal requirement, we will omit listing it in the Kalls.
3. Parameters are generally arranged so that, if the operation has any effect on its parameters, (only) the leftmost is altered. This convention is chosen by analogy with assignment statements, where the left-hand side is altered. Where appropriate, we have also named operands as *D*, for destination, and *S*, for source, in order to emphasize which operands are modified.

5-3.1 Informational Kalls

A few of the generic operations of the kernel simply provide information about objects and capabilities. The following five are typical examples:

LNSLength()

Returns the length of the C-list of the executing LNS.

CLength(X:object(GetCapaRts)) returns L:integer

Returns the length of the C-list of the object named by path *X*. (In the current implementation, a C-list may contain up to 256 capabilities.)

DLength(X:object(GetDataRts)) returns L:integer

Returns the length of the data-part of the object named by path *X*. (In the current implementation, a data-part may contain up to 2,000 words of data.)

ObjectInfo(M:mem[16], X:capa)

Stores into *M* 16 words of information about the capability *X* and the object it names; this information includes the access rights in the capability, the 64-bit unique-name of the object, and the 64-bit unique-name of the object's type. Note that the user cannot create a (possibly modified) capability from this information, so there is no reason not to make it available.

Compare(A:capa, B:capa(index)) returns B:bits

Returns a word containing bits reflecting the relations between the two capabilities. These relations include: whether *A* and *B* name the same object, whether they name objects of the same type, and whether the access rights of *A* are a subset of those of *B* and vice versa.

5-3.2 Generic Kalls

The following Kalls can be used to manipulate the contents of an object; as such, they are among the primary tools used in the coding of new abstract data type operations:

GetData(D:mem, S:object(GetDataRts), Disp, Count:integer)

Copies *Count* words from the data-part of the object *S* into the memory area beginning at *D*; the copy begins at the *Disp*th word of the data-part.

PutData(D:object(PutDataRts), S:mem, Disp, Count:integer)

Copies *Count* words from memory into the data-part of *D*; the copy begins at memory location *S* and the *Disp*th word of *D*.

AppendData(D:object(AppendDataRts), S:mem, Count:integer)

Copies *Count* words from memory, starting at location *S*, and appends them to the end of the data-part of *D*.

GetCapa(D:slot(index), S:capa)

Copies a capability from *S* into *D*; *DeleteRts* are always added in the new capability. If any capability in the path to *S* lacked *ModifyRts*, *ModifyRts* will be removed from the copy in *D*; similarly, if any capability in the path to *S* lacks *EnvRts*, *EnvRts* will also be deleted from the copy in *D*.

PutCapa(D:slot(pretarget(PutCapaRts)), S:capa(index,EnvRts), R:rights)

Copies the capability from *S*, a slot in the LNS, to *D*. *DeleteRts* are set, then all rights are restricted according to *R* (which may therefore remove *DeleteRts* again) before the copy is stored in *D*.

AppendCapa(D:object(AppendCapaRts), S:capa(index,EnvRts), R:rights)

The effect is similar to *PutCapa*, except that the (restricted) copy of *S* is appended to the end of the C-list of object *D*.

Restrict(D:capa(pretarget(PutCapaRts,KillRts),DeleteRts), R:rights)

Restricts the rights of *D* as specified by *R*—that is, the existing rights of *D* are "anded" (intersected) with *R*. Thus, the resulting rights can be no greater than those of either *R* or the original ones of *D*.

Delete(D:capa(pretarget(KillRts),DeleteRts))

Deletes the capability *D*.

As the reader may have noted, there is a weak analogy between the structure of the operations above and those of a "general register" computer: the LNS is rather like the bank of registers, the "get" operations are analogous to those instructions that load the registers, and the "put" operations are analogous to those that store them into memory.

In addition to the Kalls above, there is a moderately large collection of composite operations. These operations, except for the fact that they are indivisible, are each equivalent to a sequence of the operations listed above. A few of the composites are listed below.

TakeCapa(D:slot(index), S:capa(pretarget(KillRts),DeleteRts))

Similar to *GetCapa*, except that the source capability is then deleted; that is, except for indivisibility it is the same as

GetCapa(D,S); Delete(S)

PassCapa(D:slot(EnvRts,DeleteRts), S:capa(index), R:rights)

Similar to *PutCapa*, except that the source capability is then deleted; that is, except for indivisibility it is the same as

PutCapa(D,S,R); Delete(S)

InterchangeCapa(A:capa(DeleteRts), B:capa(index,EnvRts), R:rights)

Similar to

GetCapa(N,A); PutCapa(A,B,R); TakeCapa(B,N)

except, of course that no LNS slot (*N*) is actually used.

These last three Kalls, and especially *InterchangeCapa*, turn out to be quite useful for synchronization. We will see an example in Section 5-5.7.

5-3.3 Kalls for Creating Objects

There are several ways to create an object; the simplest of these is to copy an existing object:

> *Copy(D:slot(index), S:object(index,CopyRts))*
>
> Creates a new object of the same type as *S*; the initial contents (both data-part and C-list) will be identical to those of *S*. A capability for this new object will be stored in *D*, the initial rights in this capability will be identical to those in *S*, except that *DeleteRts* will be added. (For some kernel-defined types certain other rights may also be added.)

The second method of creating an object is to use the *Create* Kall, defined below. This method is only slightly more difficult than copying an existing object—but it is more difficult to explain. First, consider the *Create* operation itself:

> *Create(D:slot(index), T:template(CreateRts))*
>
> *T* must be a creation template (see below); an object of this type will be created and a capability for it will be stored in *D*. The capability will have those access rights present in the creation template.

Recall from Section 4-2 that every object has a type which is specified as the unique-name of another object whose type is TYPE. If, for example, there is an object whose type is FILE, then there must be somewhere another object na ed FILE whose type is TYPE. TYPE objects serve as representatives of the class of objects of a given type, and to create an object of the type one must possess a creation template for the class. To obtain a creation template, one uses the Kall:

> *MakeCreationTemplate(D:slot(pretarget(PutCapaRts)),*
> *S:capa(index,TYPE,TemplateRts))*
>
> *S* must be a capability for an object whose type is TYPE. (Note that this capability must have *TemplateRts*, which is an auxiliary right specific to type TYPE.) A creation template for the type named by *S* is placed in *D*.[2]

In the usual case the implementor of a subsystem will want to force all object creations to be done by one of the subsystem procedures. Therefore, he would never distribute either a capability for the TYPE object that has *TemplateRts* or a creation template. The complication of having creation templates is logically unnecessary; the *Create* operation could have used a

[2]This treatment of template creation, and that which follows, has been somewhat simplified from what is actually implemented; the authors feel that the subject was already complicated enough without introducing additional implementation details.

capability for the TYPE object instead of a template. However, in the case that the subsystem wants to allow users to create objects of its type, the subsystem can distribute creation templates without having to distribute capabilities for the TYPE object. (Such capabilities would give a user access to the *MakeAmplificationTemplate* Kall, discussed below.)

For kernel-defined types, Hydra provides Kalls which create objects directly and supply the maximum permitted rights, e.g., *MakeUniversal*, *MakePort*, etc. These Kalls all accept a single argument, a path to an empty C-list slot.

5-3.4 The Call Mechanism

The *Call* mechanism is the heart of both the protection and the data abstraction facilities provided by Hydra. First we have the merge operation described in Section 4-9:

> *Merge(D:slot(index), T:template(index), S:capa)*
>
> If no errors are discovered, *Merge* leaves a capability for the object named by *S* in *D*, this capability will have its access rights set as follows:
>
> - *DeleteRts* is always granted.
> - If *T* is a simple parameter template, the rights will be identical to those of the required rights of *T*. If *T* is an amplification template, the rights will be identical to the new rights of *T*, except that certain rights cannot be amplified, notably *ModifyRts*, *UncfRts*, and *EnvRts*.
> - If any capability in the path to *S* lacked *EnvRts*, *UncfRts*, or *ModifyRts*, the corresponding right will be removed from *D* as well.
>
> An error condition will be raised if either the type of *T* and *S* do not agree or the rights in *S* are not a superset of the required rights of *T*.
>
> *Call(D:slot(index), P:object(PROCEDURE,CallRts), argument-list)*
>
> Creates an LNS from procedure *P* and transfers control to it. Except for capabilities passed as parameters (as items in *argument-list*), the new LNS is a completely new environment that cannot access the objects named by the calling LNS. If *P* returns a capability, it will be stored in slot *D*.

The details of *Call* are important. It works as follows:

- Each item in *argument-list* specifies (a path to) a capability.[3]

[3]In reality, a number of facilities are provided for conveniently restricting the rights of such capabilities, creating temporary objects that simply hold data, and so on. Thus, *Call* may be viewed as a highly parametrized composite operation. None of these more advanced facilities are logically necessary, so they are omitted from this discussion. The Hydra Reference Manual [New77] has more details.

- An LNS object is created; its C-list is initialized from the C-list of the procedure object, *P*, and *argument-list* as follows:
  - Each capability (and creation template) in the C-list of *P* is simply copied to the corresponding C-list position of the new LNS. If the capability for *P* lacks *UncfRts*, both *UncfRts* and *ModifyRts* will be deleted from each copied capability. Similarly, if *P* lacks *EnvRts*, the copied capabilities will have *EnvRts* deleted as well.[4]
  - Each parameter template is merged with a capability from *argument-list*; if any of these *Merge* operations fail, the entire *Call* will fail.
- The data-part of the LNS is initialized by copying the data-part of the procedure *P*. The data-part contains information such as the starting address of the procedure's code, and is not of importance to this presentation.

When the new LNS returns (see below) it may return two "values"—a simple integer and a capability. The integer value appears as the value of *Call*; that is, when calling the procedure named by the capability *P* with parameters *X*, *Y*, and *Z*, one may write

if *Call(D,P,X,Y,Z)* < *0* **then** ...

to test the result value and store the capability result in *D*.

Return(V:integer, S:capa(index,EnvRts), R:rights)

Causes control to return from the currently executing LNS to its caller. Two "values" are returned: a simple integer, *V*, and a capability, *S*. The rights of the returned capability are restricted by *R*, except that *DeleteRts* are added.

In order to create a procedure containing parameter templates, one must first have the templates. They can be created with the following two Kalls.

MakeSimpleTemplate(D:slot(pretarget(PutCapaRts)),
S:object(index,TYPE,TemplateRts),
R:rights)

S must be a capability for a TYPE object. A simple parameter template will be placed in *D*; the type field of this template will be the same as that of *S* and its required rights field will be *R*.

[4]The motivation for these somewhat obscure manipulations with *UncfRts* and *EnvRts* is given in Chapter 7, as is a complete description of the effect of removing them. The reader may ignore these details for now.

MakeAmplificationTemplate(D:slot(pretarget(PutCapaRts)),
S:object(index,TYPE,TemplateRts),
RR, NR:rights)

S must be a capability for a TYPE object. An amplification template will be placed in D, the type field of this template will be the type of S, and its required and new rights fields will be RR and NR, respectively.

Note that a subsystem will generally make simple templates widely available for the type that it implements, *but will not distribute amplification templates.*

There is another Kall that facilitates calling procedures and is of enormous practical importance: *TypeCall.*

TypeCall(D:slot(index), S:capa, P:object(PROCEDURE,CallRts), argument-list)

Suppose that S has type $t(S)$ and, further, that $t(S)^P$ denotes the capability that one would name by a path P rooted in the $t(S)$ TYPE object. (This is the only case in which a path is not rooted in an LNS.) Then,

TypeCall(D,S,P, argument-list)

is equivalent to

Call$(D,t(S)^P$, *argument-list)*

That is, the kernel calls a procedure stored in the TYPE object—the user need not have a capability for either the procedure or the TYPE object. The capability S is called the *type representative* for the *TypeCall* operation. It may be a template or an object capability, and it need not be (but often is) passed as one of the arguments to the procedure.

TypeCall is important for two reasons. First, we can use it to achieve an important additional level of abstraction. In particular, by adopting conventions about the way in which certain operations are named in *TypeCall*, we can abstract away from the specific type of the objects involved. Second, it provides a useful convenience in that the user need not have access to capabilities for all the procedures he might need. The subsystem implementors can simply provide access to them via *TypeCall.* Since the first of these reasons is by far the more important, let's consider three examples of it:

1. Certain operations, such as "What is your status?," make sense on nearly all objects. By adopting a convention that all subsystems will provide such an operation, and will store a capability for this operation in slot N of the TYPE objects, we can perform

TypeCall(D,X,N,X)

to obtain the status of any object *X*.[5] If *X* names a file, for example, this might provide its type, its creation date, date of last access, etc. If *X* names a semaphore, on the other hand, the result might simply indicate the number of processes blocked on it. For a rigorously secure type, the subsystem designer's "status" procedure may choose not to return any information. Thus, even though the operation may be sensible and subsystem implementors *should* provide it, there is nothing, other than convention, that enforces particular semantics on the operation.[6]

2. Operations such as "Print yourself" make sense for most, but not all object types. Note that the "print yourself" operation may be quite different for each of the types to which it applies: printing a text file may simply dump the ASCII characters; printing a program may cause it to be automatically "pretty printed" and could be language-specific; printing a binary file could provide octal, hex, decimal, instruction, and character interpretations for each word.
3. Certain operations such as "Lookup entry" make sense only on a class of types that implement a common abstract concept, such as "Catalogue" or "Directory." In fact, two "directory" subsystems exist on Hydra/C.mmp, and many more could be implemented. Each provides a compatible set of operations for mapping string names to capabilities, inserting new (string, capability) pairs into the map, changing the string names in various ways, and so on. A command language interpreter needs access to these facilities in order to interpret the names that a user types at the terminal, but would like to remain impartial to the subsystem used. By convention, all subsystems that implement the abstraction of a "directory" have common operations and common paths to these operations from their TYPE objects. Hence the command interpreter, or any other program that wishes to use a specific directory, can use *TypeCall* to manipulate the directory and need not know which specific directory type is involved.

5-3.5 GST Kalls

Hydra attempts to present the user with the image of a "one-level store." In conventional systems, the user is generally conscious of two levels: primary memory in which his program and data reside, and secondary memory in which he has "files." For the Hydra user, however, there are simply objects that can be named by paths rooted in his LNS; he is not (usually) conscious of whether the object is in primary memory or secondary memory—that is an

[5]We assume that the status information will be returned in an object to LNS slot *D*. Note that *X* appears as both the type representative and an argument to the procedure.

[6]Each subsystem would have to decide also what auxiliary rights to use to control the application of the operations. It is unlikely that a subsystem would devote one of its rights for each operation of this kind.

implementation detail that is left to the system.

In reality, of course, the implementation keeps the representation of some objects in primary memory while others are on disk (Chapter 11 discusses the implementation). While the user can usually ignore this fact, on some occasions he would like to ensure that an updated version of an object is written to disk. Once on disk, of course, an object will survive a system crash; those in primary memory are more vulnerable. Therefore, the system provides the following Kall:

Update(D:object)

The object *D* is "updated"; that is, a permanent copy of the object is made on secondary storage.

The *Update* operation is typically used by cautious subsystems after they have modified an object.[7] The file system example in Chapter 8 is an excellent example of this.

5-4 KERNEL SUBSYSTEMS

As noted in the introduction to this chapter, the kernel may be visualized as consisting of two parts: the generic operations plus the *Call* mechanism, and a set of subsystems that define some basic object types. In this section we will discuss these object types briefly. In most cases a more complete treatment appears in a subsequent chapter. The types defined by the kernel are listed in Table 5-2.

DATA and UNIVERSAL objects are basic structures which can be used when data and/or capabilities need to be encapsulated for a purpose that doesn't justify the creation of a new object type. They are often used, for example, to pass a collection of related data and capabilities to a procedure. There are no type-specific operations or auxiliary rights for either of these types; the generic operations for manipulating the data-part and C-list are sufficient.[8]

TYPE objects have already been discussed. They have two auxiliary rights: *TemplateRts* and *ChangeTypeRts.* The first of these permits creation of templates, as discussed earlier. *ChangeTypeRts* permits one to alter certain information in the data-part of the TYPE object; this information specifies, for example, the initial and maximum size of instances of the type.

The remaining kernel-defined typesare discussed in later chapters.

[7]The *Update* Kall is a partial solution to the problem of performing atomic updates on secondary storage. See [Stu74] and [Lam80] for additional discussion of this interesting and complex problem.

[8]Data objects can be created automatically during *Call* to hold parameters, using mechanisms that we have chosen to pass over in this presentation.

Table 5-2 Kernel-defined object types

Data	DATA objects contain only a data-part; they are used to encapsulate short data segments. (See below for more details.)
Universal	A UNIVERSAL object has both a data-part and a C-list; it is used to encapsulate data and capabilities. (See below for more details.)
Type	TYPE objects define and represent classes of objects. (They have been discussed in Sections 4-2 and 5-3.3 and elsewhere.)
Process	Hydra's abstraction of an independent, schedulable unit of computation. It is a separate abstraction from procedures and LNSs. (See Chapter 12.)
Semaphore	POLICYSEMAPHORE and KERNELSEMAPHORE objects are Hydra's abstractions of counting semaphores. These objects may be used to achieve either exclusion or synchronization. (See Chapter 12.)
Policy	POLICY objects provides the abstraction that allows user-level schedulers to communicate with the kernel scheduling mechanism. (See Chapter 12.)
Page	A segment of information that can be made directly addressable. It is an abstraction of C.mmp's 8K-byte page frames. (See Chapter 13.)
CPS	A CPS represents a process' "working set" of pages—those pages which are resident in primary memory whenever the process is executing. (See Chapter 13.)
RPS	An RPS represents that subset of the process' working set that is directly addressable at a given instant. (See Chapter 13.)
Port	A port is an asynchronous message-passing facility (See Chapter 6.)
Device	DEVICE objects represent C.mmp's physical I/O devices. (Chapter 14.)

5-5 A COMPLETE EXAMPLE

To consolidate the ideas of the last three chapters, we will now show how a user would go about creating a real Hydra subsystem. Up to this point we have tried to hide many details to give the reader a better view of the important concepts in Hydra. In this section, however, we will reverse this trend and try to give the reader a feel for what it is actually like to program under Hydra.

The subsystem we will create will be called the *Box* subsystem. A Box may be thought of as something which holds a single capability; we will provide two Hydra procedures, *Deposit* and *Withdraw*, to place a capability into a Box and remove a capability from one, respectively. These operations are destructive: a capability deposited in the box replaces any capability already present, and withdrawing a capability removes it entirely. We will

implement the Box subsystem in such a way that only the *Deposit* and *Withdraw* procedures can access the Box's representation.

5-5.1 The Programming Environment

Let us first consider the environment in which the Box subsystem will be developed. A Hydra subsystem, like the Hydra kernel itself, is written in Bliss/11 on a PDP-10, cross-compiled and linked for the PDP-11, and then transferred to Hydra/C.mmp over the ARPANET. A simple program receives the subsystem's compiled and linked code and encapsulates it as a universal object containing one or more page objects. To turn these pages into complete procedures, the user must create procedure objects, install in them the code pages and any necessary parameter templates or inherited capabilities, and establish the procedures' initial addressability. When creating new subsystems, the user must also create a new object type, install the procedures in the new type object (for *TypeCall*), and define auxiliary rights for the subsystem.

The Hydra user may perform all these operations from the Hydra command language or he may write an additional Hydra procedure that builds the subsystem for him. This additional procedure, as we will see, is much simpler, and so can be assembled directly in the command language (see Section 10-1.3) using various utilities. We will illustrate the initialization of our example subsystem by writing a procedure (*CreateBoxSubsystem*) to create the Box subsystem. We will not discuss the details of creating this procedure using the command languge, but the reader should satisfy himself (after reading Chapter 10) that this is straightforward.

5-5.2 Programming Subsystems

Bliss/11 [Wul71] is the programming language normally used by Hydra subsystem implementors. However, the syntax and semantics of Pascal [Jen76] are more widely known and better suited to an introductory example. In the remainder of this section, we will use Pascal (occasionally with obvious extensions) for clarity, but attempt to retain the flavor of Bliss programming by faithfully adhering to the data structures that are actually used.

Kalls appear as predefined external functions in the programming language; C-list slot arguments are represented as integers, paths are represented by variable-length integer vectors, and access rights are two-component records of bit vectors:

```
type
    Slot = integer;
    Path = array 1..* of Slot;
    KernelRights = array 1..16 of boolean;
    AuxiliaryRights = array 1..8 of boolean;
    Rights = record
        KernRts: KernelRights;
        AuxRts: AuxiliaryRights;
        end;

function PutCapa(D:Path; S:Slot; R:Rights): integer;
(etc.)
```

The kernel rights are predefined as constant boolean vectors with only one element having the value "true":

```
const
    PutCapaRts: KernelRights := [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
    GetCapaRts: KernelRights := [0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
    (etc.)
```

As should be evident, we are using boolean vectors to represent sets of rights. The "or" operation (boolean "addition") is just set union, e.g.,

```
var PutAndGetRts: KernelRights;

PutAndGetRts := PutCapaRts + GetCapaRts;
```

We will assume the subsystem creator has compiled and linked together three Pascal procedures: *CreateBoxSubsystem*, *Deposit*, and *Withdraw*, which implement the three corresponding Hydra procedures. It is these three routines we will be examining. Because the subsystem is small, the code for all three routines will fit into a single Hydra page and we assume that this page is given to the user by the PDP-10-based compiler and linker.

The following declarations will generally be needed:

```
const
    AllRights: Rights := [[1,1,1,...,1], [1,1,1,1,1,1,1,1]];
    NoAuxRights: AuxiliaryRights := [0,0,0,0,0,0,0,0];
    NoRestrictions: Rights := AllRights;
```

5-5.3 CreateBoxSubsystem

The *CreateBoxSubsystem* procedure has the following duties:

1. It must create the new object type (BOX).
2. It must create the Hydra procedures *Deposit* and *Withdraw* and place them in the Box type object (so they may be invoked with *TypeCall*).
3. It must return any capabilities the subsystem builder wants to have. In our case we will return:

    a. A capability for the Box TYPE object, so the creator may modify it later (perhaps to replace the procedure capabilities).
    b. A capability for a Box parameter template, which the creator may distribute publicly so that users may write procedures which take Box objects as parameters.
    c. A Box creation template, also distributed publicly, so that users may create new Boxes. (This is a policy decision; it is safe here because Box objects happen to require no subsystem-specific initialization.)

As subsystem designers, we must establish the protection requirements for Box objects. We define two auxiliary rights bits corresponding to the two subsystem procedures:

```
const
    BoxDepositRts: AuxiliaryRights := [1,0,0,0,0,0,0,0];
    BoxWithdrawRts: AuxiliaryRights := [0,1,0,0,0,0,0,0];
```

Now we must identify the capabilities that *CreateBoxSubsystem* will need to do its job and assign LNS slots for them. The first slots to be assigned are those holding parameters or inherited capabilities; the assignment is arbitrary but must be known to the person who builds the *CreateBoxSubsystem* procedure, since he must install the initial capabilities or parameter templates.

In our case, we need only two inherited capabilities: the capability for the page object containing the linked code for *Deposit*, *Withdraw*, and *CreateBoxSubsystem*, and a creation template for type TYPE, which we will need to create a new Box TYPE object. We decide to use the first two C-list slots for these capabilities:

```
const
    TypeCreationTemplate = 1;
    PageCapability = 2;
```

CreateBoxSubsystem will also need some temporary LNS slots to hold capabilities during its execution, so we will assign them also. This assignment is completely arbitrary as long as it doesn't conflict with the earlier ones.

```
const
    BoxType = 3;
    ReturnObject = 4;
    BoxCreationTemplate = 5;
    BoxAmplificationTemplate = 6;
    BoxParameterTemplate = 7;
    DepositProcedure = 8;
    WithdrawProcedure = 9;
```

Now we consider the Kalls needed to perform the duties listed above. First, we create a universal object to hold the capabilities we will return to the builder:

```
procedure CreateBoxSubsystem;
begin
MakeUniversal(ReturnObject);
```

We create the new type object, giving it the print-name "Box." The capability is also placed in the object to be returned (the slot used is arbitrary):

```
Create(BoxType, TypeCreationTemplate, "Box");
PutCapa(Path(ReturnObject,1), BoxType, NoRestrictions);
```
[9]

We create all three kinds of templates for the new type:

```
MakeCreationTemplate(BoxCreationTemplate, BoxType);
MakeParameterTemplate(BoxParameterTemplate, BoxType);
MakeAmplificationTemplate(BoxAmplificationTemplate, BoxType);
```

We put the creation and parameter templates in *ReturnObject.* We restrict the rights in the creation template to eliminate all but the auxiliary rights and the "safe" kernel rights.

```
PutCapa(Path(ReturnObject,2), BoxCreationTemplate,
    [CreateRts+DeleteRts+ModifyRts+EnvRts+UncfRts;
        BoxDepositRts+BoxWithdrawRts]);
PutCapa(Path(ReturnObject,3), BoxParameterTemplate, AllRights);
```

We now create the two procedure objects. We give each a capability for the same code page, but each gets a different starting address.[10] The assignment of C-list slots in these procedures is also arbitrary and independent of the slots assigned in *CreateBoxSubsystem.*

[9]In a more suitable language environment, the argument *NoRestrictions* could be defaulted. Likewise, the parameter *Path(ReturnObject,1)* is really representing **ref** *ReturnObject[1].*

[10]*SetStartingAddress* is one Kall necessary to establish initial addressability; others (not shown) specify the initial CPS and RPS (see Chapter 13). The actual values of the procedure starting address for *Deposit* and *Withdraw* were assigned by the linkage editor.

```
const
    DepositPage = 1;
    WithdrawPage = 1;

MakeProcedure(DepositProcedure);
MakeProcedure(WithdrawProcedure);
PutCapa(Path(DepositProcedure,DepositPage), PageCapability,
    NoRestrictions);
SetStartingAddress(DepositProcedure, Deposit);
PutCapa(Path(WithdrawProcedure, WithdrawPage), PageCapability,
    NoRestrictions);
SetStartingAddress(WithdrawProcedure, Withdraw);
```

Both the *Deposit* and *Withdraw* procedures will accept a Box as a parameter. We therefore define a parameter slot for these two procedures and place the appropriate amplification template in them. Because the *PutCapa* Kalls below specify no rights restrictions, all rights will be amplified; the *SetRequiredRights* Kall is used to insure that the caller has the appropriate auxiliary right and all the non-amplifiable kernel rights.

```
const
    DepositBoxParameter = 2;
    WithdrawBoxParameter = 2;

PutCapa(Path(DepositProcedure,DepositBoxParameter),
    BoxAmplificationTemplate, NoRestrictions);
SetRequiredRights(Path(DepositProcedure,BoxParameter),
    [ModifyRts+EnvRts+UncfRts; BoxDepositRts] );
PutCapa(Path(WithdrawProcedure, WithdrawBoxParameter),
    BoxAmplificationTemplate, NoRestrictions);
SetRequiredRights(Path(WithdrawProcedure,BoxParameter),
    [ModifyRts+EnvRts+UncfRts; BoxWithdrawRts] );
```

The Deposit procedure will also accept a second parameter: a capability of any type. The *MakeNullParameterTemplate* will create a parameter template which matches any object type. No rights are required except *EnvRts*.

```
const
    AnyObjectParameter = 4;
```

```
MakeNullParameterTemplate(Path(DepositProcedure, AnyObjectParameter));
SetRequiredRights(Path(DepositProcedure, AnyObjectParameter),
    [EnvRts; NoAuxRights] );
```

We now store capabilities for the two procedures in the Box type object. The rights in these capabilities do not need to be restricted because general users will never have a capability for the type object. (The builder will keep it in his private directory.) Again, the assignment of slots in the type object is arbitrary.

```
const
    DepositIndex = 1;
    WithdrawIndex = 2;

PutCapa(Path(BoxType, DepositIndex), DepositProcedure, NoRestrictions);
PutCapa(Path(BoxType, WithdrawIndex), WithdrawProcedure,
    NoRestrictions);
```

Finally, we return to the subsystem builder:

```
Return(0, ReturnObject, NoRestrictions);
```

5-5.4 Deposit

The *Deposit* routine below is quite simple; it just stores the user's capability in the Box and returns. If a capability is already there, it is replaced by the new capability and the old capability is lost. (Section 5-5.7 will discuss some issues behind this implementation.)

```
procedure Deposit;
    begin
    InterchangeCapa(Path(DepositBoxParameter,1), AnyObjectParameter,
        NoRestrictions);
    Update(DepositBoxParameter);
    Return(0, 0, 0)
    end;
```

By judiciously creating the Box amplification template in *CreateBoxSubsystem*, we are assured in *Deposit* that the correct type of argument has been passed, that the caller has *BoxDepositRts* for the Box, and that all necessary kernel rights are present, whether through inheritance (e.g., *ModifyRts*) or amplification (e.g., *PutCapaRts*).

5-5.5 Withdraw

Withdraw is equally simple for the same reasons. Note that the *AnyObjectParameter* slot in *Withdraw* is initially empty because *CreateBoxSubsystem* stored no template there.

```
procedure Withdraw;
    begin
    TakeCapa(AnyObjectParameter, Path(WithdrawBoxParameter,1));
    Update(WithdrawBoxParameter);
    Return(0, AnyObjectParameter, AllRights)
    end;
```

5-5.6 Using the Subsystem

After invoking *CreateBoxSubsystem* the subsystem creator may place the Box parameter and creation templates in a public directory. A user wishing to create a Box in LNS slot *BoxObject* will retrieve the creation template, put it in slot *BoxTemplate* of his LNS, and invoke *Create*:

Create(BoxObject, BoxTemplate);

Because of the way *CreateBoxSubsystem* restricted the rights in the creation template, the capability returned in *BoxObject* will have no C-list or data-part rights, effectively making it impossible to do anything with the box but invoke *Deposit* and *Withdraw*. To invoke *Deposit* and store a capability from slot *Capa* of his LNS in the box, the user would invoke

TypeCall(0, BoxObject, DepositIndex, NewBoxObjectSlot, Capa);

The user could then retrieve the capability (into slot *NewCapa*, say) by invoking *Withdraw:*

TypeCall(NewCapa, BoxObject, WithdrawIndex, BoxObject);

The subsystem creator would probably supply Bliss/11 macros to make the *TypeCall* look something like, say,

Deposit(BoxObject, Capa);
Withdraw(BoxObject, NewCapa);

5-5.7 Some Design and Implementation Issues

The implementation of *Withdraw* and *Deposit*, although apparently simple, involve some subtle points that we should mention.

Synchronization In *Deposit* and *Withdraw*, the *InterchangeCapa* and *TakeCapa* Kalls were carefully chosen to make explicit synchronization unnecessary. The subsystem builder must be conscious of the possibility that his procedures will be called simultaneously by different users of the same subsystem object. In this case, if we had replaced *InterchangeCapa* with the sequence *Delete(...); PutCapa(...)* there would be an instant in which *Withdraw* would find no capability in the Box. Similarly, replacing *TakeCapa* with *GetCapa(...); Delete(...)* would permit two callers of *Withdraw* to get the same capability out of the Box. If the Box subsystem were more complex, explicit synchronization could be accomplished by placing a Policy Semaphore in each Box. (See Chapter 12.)

Fault tolerance The Box subsystem was defined so that if two users attempt to withdraw a capability from the same Box at the same time, only one user will get it. The *Update* Kalls in *Deposit* and *Withdraw* guard against unpredictable behavior should Hydra/C.mmp crash in the midst of these operations.

The *TakeCapa* and *InterchangeCapa* Kalls make modifications in the objects in the Active GST; the changes are not necessarily reflected in the Passive GST immediately.[11] If the *Update* Kalls are omitted, the following sequence of events is possible.

1. *Withdraw* is invoked on Box *B* by user 1, and it returns a capability for object *C*. The capability is placed in object *D*, and the user invokes *Update(D)*. *Box B is not updated.*
2. The system crashes. The Active GST is lost. The system comes up again.
3. *Withdraw* is invoked on Box *B* by user 2. A capability for *C* is returned to user 2, because when object *B* is retrieved from the Passive GST it is in the state it was in prior to the first call on *Withdraw*. User 1's object *D* will have survived the crash, and so it contains the original capability for *C*.

By invoking *Update* in *Withdraw* and *Deposit*, we ensure the Box is in a stable state (with respect to system crashes) before the procedures terminate.

Security Is the Box subsystem secure? Can a user get a capability for a Box object with rights sufficient to access its C-list or data-part directly? There are two questions to answer.

1. Can the user create a Box object with liberal rights? No, because the only creation template available had its rights restricted by *CreateBoxSubsystem*.

[11]The Active and Passive GST are discussed in Chapter 11. The reader may wish to return to this section after reading that chapter.

2. Can he obtain an amplification template which could amplify the rights on a restricted capability? If he could obtain a capability for the Box type object the user could breach the security of the system by either creating a new amplification template or by retrieving one from the subsystem procedures, which are accessible through the type object. The only capability for the type object was returned to the builder by *CreateBoxSubsystem.* The builder can keep the capability in his private catalogue (in which case he depends on the security of the Catalogue subsystem), or he can imbed the capability in a Hydra procedure which can implement any arbitrary authentication algorithm before returning it.

Subsystem maintenance All software systems undergo modifications from time to time. For Hydra subsystems, these modifications are usually accomplished by changing the subsystem procedures. The *CreateBoxSubsystem* is not a good vehicle for making such changes; every time it is executed it creates a *new* Box type object, i.e., a new subsystem as incompatible with the old one as Files and Directories.

This is a common engineering problem in Hydra and other operating systems. Additional utilities must be created to "replug" procedure capabilities in existing type objects and to maintain "versions" of subsystems so that a system can be "rolled back" when a bug is found in the current system.

5-6 RETROSPECTIVE

In general the operations and access rights provided by Hydra have proven adequate for the construction of the subsystems that have been built. These subsystems, in turn, span a sufficiently broad spectrum that we feel fairly confident of the adequacy of the kernel facilities. Given the relatively primitive level of the *operations* (Kalls), this is perhaps not too surprising; it is fairly simple to determine their adequacy by inspection. For the *rights,* the case is less obvious; this issue will be the main topic of Chapter 7.

Whether the Hydra operations and rights are the *best* set that could have been chosen is unclear. In retrospect we see many things that we would have done differently. For example,

1. There are too many rights. In practice, a subsystem either grants complete access to its representation or it grants no access. A single right could have controlled all seven of the C-list and data-part rights listed in Table 5-1.
2. Some of the more esoteric rights, notably *EnvRts* and *UncfRts* (discussed in detail in Chapter 7), are probably not worth the bother. Their effect in many cases can be achieved more simply, and they are not complete solutions to the problems they were intended to address. In another iteration of the system design we would probably try for a single right that

covered the most common cases that these were intended to address and let the concerned programmer handle the more subtle cases.

3. There are too many kernel-defined types; many could have been eliminated, especially if we had better hardware support that allowed direct addressing of the data-part.
4. The composite operations (e.g., *InterchangeCapa*) were partly a response to the need for more convenient primitives, but the main motivation was to ensure indivisibility. A better approach might have been to provide a general means for applying a sequence of more primitive operations indivisibly.

There are, of course, a number of operations that we would add or delete; none of these is fundamental, however.

In retrospect we see *TypeCall* as an essential abstraction; in a next-generation system we might not provide *Call* at all. In light of advancements in programming methodology, we recognize *TypeCall* to be a crucial part of an object-oriented programming style, providing an absolutely indispensable level of abstraction. Strangely, we originally viewed it as merely a convenient way to avoid a proliferation of capabilities for procedures.

5-7 FURTHER READINGS

The detailed design of capability-based systems offers a myriad of opportunities for variation. Hydra's design developed largely from the model in [Jon73]. Other designs stress other aspects of protection more heavily; [Red74] considers the problem of revocation of access, [Wil79] offers an alternative to rights amplification, [Fer74] concentrates on domain structure. The high cost of interpreting references through capabilities is well-understood [Stu74], prompting hardware architectures that support capabilities directly [Eng74, Wil79].

CHAPTER

SIX

THE MESSAGE SYSTEM

An operating system that encourages the use of cooperating sequential processes has a dual responsibility. On the one hand, it must provide protection mechanisms to insulate processes from one another so that erroneous or malicious behavior on the part of one cannot interfere with unrelated ones. On the other hand, it must also provide mechanisms for cooperation among the processes working on a common task. The last two chapters have dealt with some aspects of Hydra's response to the first of these responsibilities. In this chapter we shall deal with one aspect of the second.

Within the Hydra context, a wide range of interaction mechanisms are possible, from tightly coupled memory sharing to loosely coupled message communication. Moreover, the user is free to define application-specific mechanisms that lie anywhere along this spectrum. The Hydra Message System is a particular communication facility which we believe is convenient for many loosely coupled applications, and which can form the basis for many others.

6-1 OVERVIEW OF THE MESSAGE SYSTEM

The design of the Hydra Message System was motivated by several objectives. First, we wanted the Message System to be very flexible, in keeping with Hydra's own role as a general-purpose system. We wanted to be able to use the facility for communication between processes in a single "job" as well as between processes in different jobs. In particular, we felt it was inappropriate for the destination of a message to be a process, as is common in many systems, since it is difficult for dynamically created processes to know each other's names.

Second, we wanted to support a "user/server model," in which processes would provide abstract "services" to other processes via the Message System. The separation of user and server in this model is important; both should be able to hide details of their implementation from the other. In particular, we wanted to be able to vary the number of user (server) processes dynamically, without affecting the server (user).

Third, envisioning a system in which message communication would be used for many purposes, we felt it was imperative to provide for "multiple waiting," so that a process could wait for messages from a number of different sources. Indeed, we wanted to allow the process to specify the "nature" of message it was waiting for, in addition to the source.

Finally, because we did not think anyone knew the "best" design for a message system, we wanted our system to be efficient enough to support other user-level communication mechanisms which might be better suited for particular applications.

To meet these objectives, we designed the system around a number of concepts: *ports*, *connections*, *messages*, and *replies*.

Ports Messages are sent between objects of type PORT, not between processes. A port is in one sense the abstraction of a "service"; one sends a message to a port to request that service. Since ports are objects, they may be shared; in particular, several server processes can share a port in such a way as to make their number transparent to the users.

Channels and connections Messages leave ports on *output channels*; they arrive at ports on *input channels*. A topology of possible communication paths is established by defining a set of *connections* between output channels and input channels of different ports. After a connection is established, the destination of a message need not be specified—only the output channel is necessary. Input channels in a single port also form a unit upon which multiple waiting is possible.

By creating, sharing, and connecting ports appropriately, any communication graph can be established. Figure 6-1 shows a number of alternatives.[1] The simplest structure is the single server/single user shown in (a). As shown in (b) and (c), either side of the connection may be implemented by cooperating processes sharing the port. Non-cooperating users would use different ports, as in (d).

Messages In our system, a *message* is best thought of as a vehicle or container for transmitting data, rather than as the data itself. In particular, the operations of creating a message, putting information into it, and sending it are all distinct. Similarly, receiving a message is distinct from reading its contents. Since the identity of a message is distinct from its contents, it makes sense to read a message, modify its contents, forward it to another server, and still talk about it as being the *same* message.

[1]By historical convention, we draw ports as triangles rather than using the standard object notation. (It was thought that the triangular shape was reminiscent of a "hydraphone.")

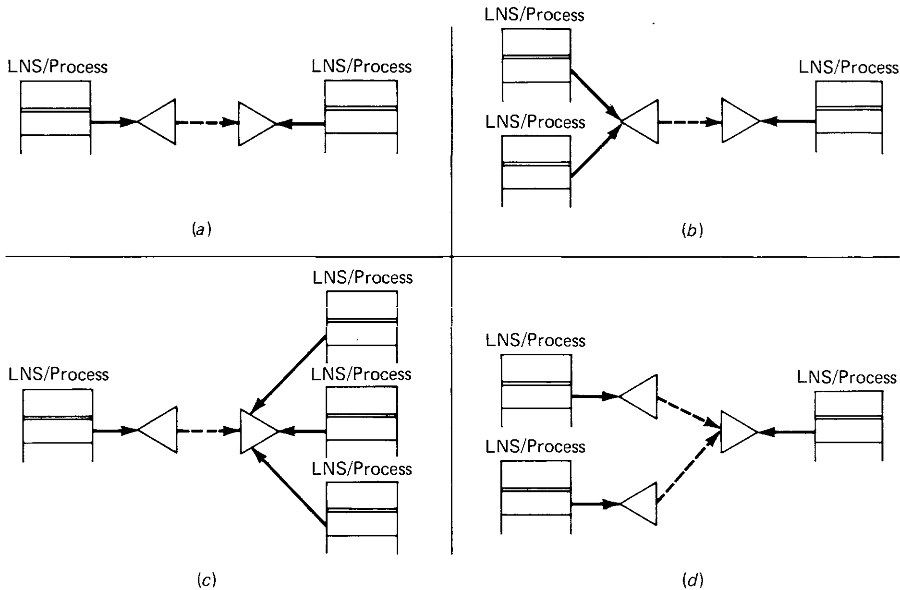

Figure 6-1 Simple interprocess communication structures

Replies Given our view of messages as "persistent," it is also reasonable to associate historical control information with a message. By remembering the identity of the originating port, we are able to develop a formal notion of a "reply" to a message—a means for returning a message to the sender. This allows a server process to respond to a user request without an elaborate protocol. The situation is analogous to subroutine invocations—the subroutine provides a service and returns to its caller without knowing the identity of that caller. Indeed, the analogy with subroutines is a strong one—Hydra in fact maintains a stack of "return points" in the message. Hence it is possible for a message to be forwarded many times, from one server to another, and still return to each point along the inverse route.

Input/output Input/output is an asynchronous activity (service) which is easily modeled on a message paradigm. Hydra does this explicitly by representing peripheral devices as objects of type DEVICE to which messages can be sent exactly as with ports.

6-2 AN EXAMPLE: DATA BASE MANAGEMENT

Before describing the Message System in detail, we can give an example of the kind of communication that is possible in Hydra. Suppose that one wishes to implement a data base system in which an arbitrary number of user

processes can make inquiries and updates. Further, suppose that the actual data base is distributed across many physical disk drives, and that the expected high degree of concurrent access to the disks means that head motion and latency optimizations would highly desirable. Figure 6-2 illustrates a possible communication structure for such a system.

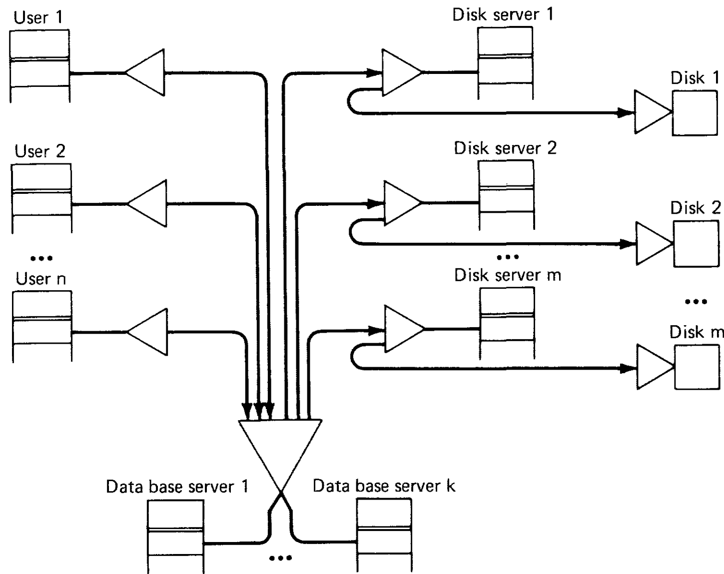

Figure 6-2 A data base communication structure

In the example, user processes each have ports with an output channel connected to a single "service port" of the data base system. An arbitrary number of server processes extract requests from this port and map the requests into operations on the disk devices.[2] Actually, what appear to be disk devices to the Data Base Servers are actually Disk Server processes which buffer and reorder the I/O requests to minimize head movement and latency on the physical devices.

If, for the moment, we make the simplifying assumption that each user's data base request maps to a single disk operation, we can trace such a request through the system.

1. The user process generates the requesting message, whose contents will reflect both the nature of the request (e.g., "read") and the logical entity to be operated upon (e.g., "Harbison's address"). When the user process

[2]We assume that any server can process any request.

sends the message it will be enqueued at the Data Base Servers' port. (It will implicitly contain the name of the originating user's port.)

2. One of the Data Base Servers will eventually receive the message and, on the basis of its contents, will convert the request into a physical I/O operation on one of the disks. The Data Base Server process will then alter the contents of the message to specify this physical operation and forward the message to the appropriate Disk Server port.
3. The Disk Server process will receive the message, read its contents, and decide how to schedule the operation in relation to the others it presently has. Eventually, the message will again be forwarded—this time to the disk DEVICE object, where the kernel will perform the actual I/O operation.
4. When the operation completes, the kernel will reply to the message. By virtue of being the last sender, the Disk Server process will receive the reply, giving it the chance to verify that the operation completed without error.
5. The Disk Server can now reply to the message again; this time the message will be enqueued at the Data Base Server port.
6. Finally, one of the Data Base Servers (not necessarily the same one that originally serviced the user's request) will read the reply and, if all is well and no further action is necessary, will reply it to the user process.[3]

With this overview in mind, we now turn to the details of the system.

6-3 PORTS

A PORT object consists of four parts:

1. A set of *output channels.* Each connected output channel contains a reference to an input channel in some port. Output channels are numbered consecutively from 0; there is no specific upper limit to the number of them in a port.
2. A set of *input channels,* each of which can receive and queue messages from any number of output channels. There are 16 input channels in a port, numbered from 0 to 15.
3. A set of *message slots.* Message slots provide buffers which hold messages and provide a local naming mechanism for them. There is no specific limit to the number of message slots a port may have. Message slots are used only while operating on messages; they in no way limit the number of messages which may be enqueued at a port or sent by it. Message slots are named by small positive integers.

[3]In reality there is a means of short-circuiting some of this replying mechanism, which will be covered later.

4. A *blocked process queue.* A process attempting to receive a message that has not yet arrived may be placed on this queue and suspended until a message does arrive.

A port has multiple input channels so that it may send and forward messages to a variety of destinations. As will be seen, multiple input channels allow some selectivity in waiting for arriving messages. Messages are at all times physically associated with some port—either enqueued at one of the input channels or residing in a message slot. Messages can move between ports only by traveling along a connection; it is not possible to receive a message at one port and forward it out through another port.

The kernel provides a Kall, *MakePort*, to create ports.

6-4 CONNECTIONS

An output channel of one port may be linked to an input channel of another port to form a *connection* along which messages may travel. A connection is established by the *Connect* Kall:

Connect(OutPort, InPort:object(PORT,PortConnectRts),
OutChannel, InChannel, ConnID:integer)

Output channel *OutChannel* of *OutPort* is connected to input channel *InChannel* of *InPort.* *ConnID* becomes the connection identifier of the connection; every message sent over the connection will be tagged with *ConnID.* *OutPort* and *InPort* may be the same port. (*InPort* may also be a DEVICE object.) *PortConnectRts* is an auxiliary right for PORT objects.

A single output channel cannot be connected to more than one input channel, but any number of different output channels (in any number of ports) may be connected to the same input channel. This "fan-in" at the input channel is invisible to a receiving process.

An output channel may be disconnected at any time (for possible later re-connection) by the Kall *Disconnect.*

Disconnect(Port:object(PORT,PortDisconnectRts), OutChannel:Integer)

As a side effect of *Disconnect*, a special *disconnect message* is sent along the output channel. It may be used by processes on the input side of the connection to recognize the fact that the connection is being broken.

Connect is the only Kall that requires a single LNS to have a capability for both the sending and receiving port, and it is not necessary for the connector to be either the sender or the receiver. Normally, when the communication channel is of the "user-to-server" variety, it is the server system that is given the privilege of connecting the ports. That is, instead of simply making a

capability for the port available to users, the server subsystem provides a procedure that takes the user's port and output channel number. The server procedure (which can inherit a capability for the server process' own port) then performs the connection. This method has several advantages:

1. It allows the server to allocate his input channels and connection identifiers in a controlled manner.
2. It allows the server to allocate table space or other resources for the new user.
3. It allows the server to perform arbitrary access restriction by requiring the user to present other capabilities for inspection. (The file system described in Chapter 8 is a good example of this.)
4. It further hides the implementation of the server from the user. (The server might have more than one port, for instance.)

6-5 MESSAGES

It is advantageous to think of a message as a real piece of storage which passes from port to port and which can be read and written. This storage is actually divided into several fields, the relevant ones being

- A *text/capability buffer*, holding the message text and/or a (single) capability.
- A *message type*, an integer in the range 0 to 15.
- A stack of *reply frames*, recording where the message should go when replied.

The message type is uninterpreted by Hydra; it is often used by applications to differentiate classes of messages and replies (e.g., "normal" vs. "error"). The message type, the input channel on which the message arrives, and the connection identifier all help a receiver to discriminate among messages.[4]

The reply stack is central to the message reply mechanism. Each reply frame on the stack specifies:

- A port and input channel to which the reply will return.
- A *reply mask* that specifies which replies are of interest to the sender.
- A *message identifier*, an uninterpreted token supplied by the sender and returned with the reply.

Whenever a message is sent, a new reply frame is pushed on the message's stack and filled with information about the sender. A "reply" operation pops

[4]The "type" of a message should not be confused with the "type" of an object in Hydra. Messages are not objects, and the two uses of the term "type" are unrelated.

off the top frame and uses its contents to determine the return destination of the message. Of course, the contents of the message's data buffer may have been (usually is) modified by the time the message returns.

Because messages are not Hydra objects, all rights-checking in the Kalls that follow occurs on the capability for the PORT object that is the first argument. There is an auxiliary right for each major operation on ports or messages, although some do double duty. For instance, the right allowing the *WriteMsg* Kall also governs the *PutMsgCapa* Kall.

6-6 OPERATIONS ON MESSAGES

The *CreateMsg* Kall creates a new message in a port.

CreateMsg(Port:object(PORT,MsgCreateRts),
StackSize, Length:integer) returns slot:integer

Creates a new message and stores it in a free message slot, whose number is the return value of the Kall. The message's maximum text length will be *Length* and its reply stack will have a possible depth of *StackSize* frames.

Several Kalls are provided to read and write the text/capability buffer of a message.

ReadMsg(Port:object(PORT,ReadRts), Slot:integer, Address:mem,
StartingByte, Length:integer)

WriteMsg(Port:object(PORT, WriteRts), Slot:integer, Address:mem,
StartingByte, Length:integer)

GetMsgCapa(Port:object(PORT,ReadRts), Slot:integer, D:slot(index))

PutMsgCapa(Port:object(PORT, WriteRts), Slot:integer, S:capa(EnvRts),
Rights:rights)

These Kalls are all similar. The message is specified by a (*Port*, *Slot*) pair and the text is specified by an address in the user's address space (*Address*), a starting byte in the message buffer (*StartingByte*), and a length in bytes (*Length*).[5] *GetMsgCapa* transfers a capability from the message to a slot (*D*) in the LNS. *PutMsgCapa* moves a capability (*S*) from the LNS to the message, possibly restricting rights as it does so. (*ReadRts* and *WriteRts* are other auxiliary rights for ports.)

[5]This mechanism permits a user to alter small portions of the message text while leaving the rest untouched.

To send a message, one uses the *RSVPMsg* Kall,

> *RSVPMsg(Port:object(PORT,SendRts), Slot:integer, Type:integer, OutChannel:integer, ReplyMask:mask, ReplyInChannel:integer, MsgID:integer)*
>
> The message is specified by the *(Port,Slot)* pair and the destination is specified by *OutChannel*. (The output channel must have previously been connected with *Connect*.) The argument *Type* sets the message type field of the message. The remaining three arguments affect the reply: *ReplyInChannel* is the input channel on which the reply will arrive, *ReplyMask* is a bit mask that can be used to selectively ignore some replies on the basis of their message type, and *MsgId*, the message identification, is an arbitrary token that will be returned with the reply so that the sender may identify the message even if the text buffer has been altered or the messages are not replied in FIFO order. These three parameters, along with *Port*, are stored in a reply frame which is pushed onto the message's reply stack as the message is sent.

The *ReplyMsg* Kall returns ("replies") a message to its sender.

> *ReplyMsg(Port:object(PORT,ReplyRts), Slot:integer, Type:integer)*
>
> Returns the message to the previous applicable sender, as specified in the message's reply stack. If no such sender exists, the message is deleted. *Type* will be the message type of the replied message.

ReplyMsg does not necessarily return the message to the most recent sender. After obtaining the reply frame from the message, the Message System compares the value of *ReplyMask* stored therein with the *Type* parameter in *ReplyMsg*. If the bit in *ReplyMask* corresponding to *Type* is 1, then the message is returned to the indicated port. However, if the bit in *ReplyMask* is 0, the reply frame is discarded and the next reply frame is popped and processed in the same fashion, thus causing the reply to *bypass* the first port.

When the last reply frame has been popped off a message, the action of *ReplyMsg* is to delete the message. Because this is the only way in which a message can be deleted, it is impossible to thwart the reply mechanism by prematurely destroying a message.

The *ReceiveMsg* Kall provides a very flexible mechanism for receiving messages and replies.

> *ReceiveMsg(Port:object(PORT,ReceiveRts), Filter:msgfilter, Address:mem(16)) returns Slot:integer*
>
> Returns the first applicable message waiting at the port, as specified by *Filter*. Relevant characteristics of the received message are stored in the

block of memory, *Address*, and the number of the message slot holding the returned message is the return value of the Kall.

Filter is actually a series of arguments with which it is possible to specify a subset of messages waiting at a port. The details of how *Filter* is encoded are not important; it suffices to know that one can specify three classes of messages:

1. Messages from any specified subset of the 16 input channels
2. Messages having any specified subset of the 16 message types
3. Messages having a (single) specified message identifier

When more than one message satisfies the filter condition, the messages are received in FIFO order. The message filter can also indicate whether the *ReceiveMsg* Kall should block if no message matches the filter, or simply return notifying the user that no suitable messages are present.

The message description provided by *ReceiveMsg* is designed to give the receiver a lot of information about the message before reading its contents. The information includes:

1. The message type (set by *RSVPMsg* or *ReplyMsg*)
2. The input channel on which the message arrived
3. The connection identifier set by *Connect* (only if this is not a reply) or the message identifier set by *RSVPMsg* (only if this is a reply)
4. The length of the text in the text buffer
5. Three bits indicating (a) whether this is a reply or an "original" message, (b) whether this is a "disconnect message," and (c) whether a capability is present in the message

The final message operation we shall describe is *ReQueueMsg*:

ReQueueMsg(Port:object(PORT,ReceiveRts), Slot:integer, Type:integer,
Channel:integer, MsgId:integer,
ConnID:integer, Replybit:integer)

This Kall allows the programmer to use the implicit queues of the message system for enqueuing tasks to be done (where each task can be represented by a message). It is also useful for holding messages that cannot be processed immediately. Although the details may seem complex, the net effect is to requeue a specified message on a specified input channel of the same port. Any parameter of the message can be changed at the same time, but the reply stack remains intact.

Although the Message System has a large number of Kalls, they are used in fairly regular ways. Below are typical sequences of operations for a user and a server.

User	Server
slot := CreateMsg(Port,...)	
WriteMsg(Port,Slot,...)	*(* Wait for request *)*
RSVPMsg(Port,Slot,...)	
	slot := ReceiveMsg(Port2,...)
	ReadMsg(Port2,slot,...)
(Wait for reply *)*	*WriteMsg(Port2,slot,...)*
	ReplyMsg(Port2,slot,...)
slot := ReceiveMsg(Port,...)	
ReadMsg(Port,Slot,...)	

6-7 A VIEW OF THE REPLY MECHANISM

An understanding of the reply mechanism is crucial to a proper understanding of the Hydra Message System, for it is the key to structured interprocess communication and exception handling in Hydra. An analogy to ordinary sequential control structures might be helpful.

With no reply stack at all in a message, there could be no *RSVPMsg* or *ReplyMsg* primitives. The only way to transmit a message would be with a one-way "*SendMsg*" Kall. *SendMsg* might be likened to a "GO TO" statement, in which control transfer is one-way, with no information as to where it came from.

When we add the reply stack to messages, in effect we add the call stack of the familiar sequential process. Ignoring the *ReplyMask* parameter for the moment, we see that *RSVPMsg* is analogous to a subroutine call and that *ReplyMsg* is analogous to subroutine return. Most message systems provide only what amounts to a one-level subroutine call mechanism.

With the *ReplyMask* parameter of *RSVPMsg* we add an exception-handling ability analogous to ON conditions in PL/I or ENABLE declarations in Bliss/11 [Wul71]. The *ReplyMsg* operation is now more like an interprocess version of the SIGNAL statements of PL/I or Bliss/11 than a simple subroutine return. When a *ReplyMsg* is done, both control and data are transferred to that environment (i.e., port) nearest the top of the reply stack that had been "enabled" for replies of the type generated.

The use of the call-signal mechanism for structured exception handling in sequential environments is not uncommon. However, such a mechanism is even more important in a multiprocessing context. In addition to providing better program structure in exception-prone applications (such as I/O), it saves a great deal—perhaps hundreds of milliseconds—of processor time, paging time, and queuing delay by avoiding unnecessary reply handling.

It is our contention (and experience) that programmers almost always

desire to treat message transactions abstractly as subroutine calls. Both the data structure and the control structure implied by the Hydra reply mechanism would nearly always be embedded in user processes anyway if the reply mechanism were not there. With this reply mechanism, programmers are liberated from the great deal of work needed to make effective use of a communication system.

6-8 RETROSPECTIVE

Users and implementors have three chronic complaints about the Message System: it's too slow, it's too "baroque," and it shouldn't be in the kernel.

The Message System was included in the kernel principally for efficiency reasons; PORTs could have been implemented by a user-level subsystem without loss of functionality. For similar efficiency reasons, it was decided to not implement messages as true Hydra objects. In retrospect, both these decisions seem questionable. The additional effort of coding subsystems in the kernel would have been better spent on making the basic *Call* mechanism faster. Moreover, because messages are not true objects, they do not migrate to secondary storage. Under heavy load this becomes a strain on primary memory. Finally, Hydra's protection structure does not mesh well with the kernel implementation of ports. A capability names a port, yet most often one is interested in a (port, channel) pair; to establish a connection, for example, a process will often pass a capability for one of its ports to a server's procedure. The process really only wishes to pass a channel, but has no way to do so. A user-level subsystem implementation would implement connections with capabilities, and the desired protection properties would be naturally available.

Even though many people lament the Message System's baroqueness—too many Kalls with too many arguments—there is no consensus that any features should be eliminated. In fact, most users agree that the general *RSVP-Reply* paradigm is very good. The most common objection is the *lack* of one particular feature—the so-called "timed receive."

A *TimedReceive* Kall, if it were implemented, would act exactly like *ReceiveMsg* except that it would take an additional argument, an elapsed time, which would cause the Kall to abort if no message arrived within the specified interval. This is an important facility for a message system that wants to allow for cooperating but mutually suspicious processes, which Hydra certainly does. In fact, we believe that this feature should have been provided in the earliest designs of the message system.[6]

The slowness of the message system is also a common objection. We tried to alleviate this situation in two ways:

[6] "Disconnect messages" were another feature we thought of only after we tried to build systems without them. Fortunately, they were easier to add.

1. We allow the Policy Module to specify, on a per-process basis, an interval during which a process will remain under control of KMPS after blocking on a port or Policy Semaphore. Only after this interval expires will the process be returned to the PM. (Recall Section 3-3 and see Chapter 12.)
2. We provided several "composite" Kalls in the message system, such as *ReceiveAndRead* and *WriteAndRSVP*. As with other composite Kalls, these Kalls are functionally just the concatenation of the constituent operations, but some overhead is eliminated.

Interestingly, although the message system is slower than we would like, message transmission is still faster than a procedure call. This tends to encourage programmers to use processes and messages rather than procedures.

The message system's abstractions are occasionally confusing because they try to address two different paradigms for message system interaction: the message-switching model and the levels-of-abstraction model, also called the File Server model.[7]

The message-switching model focuses on the store-and-forward facilities of the message system. Messages travel to a port, where they are sorted and retransmitted by an intermediate process to their final destination. Very often, the destination of the message can be determined by the type or connection identifier of the message, thus allowing the message text to remain untouched. This model is the basis for our decision to unbundle the allocation of a message buffer from the sending or receiving of that message. The ARPANET control program example in Chapter 9 is a good demonstration of this model.

In the File Server model, we picture a user connecting to a server process which implements the "file" abstraction seen by the user. The user's messages to the server contain file operations. The File Server subtly alters the user's message buffer so that it can be forwarded directly to a disk device; i.e., it transforms the message from a file request to a disk request. Replies from the disk are then sent back to the user, perhaps bypassing the Server except when an error occurs. This model is the basis for much of the reply mechanism.

The message system's treatment of a message buffer is a compromise between the ideals of these two paradigms. The advantages gained by viewing message transactions as asynchronous procedure calls are somewhat offset by the necessity of sharing a common text/capability buffer at all levels of the transaction. The decision to separate buffer allocation from send/receive was based on the premise that little alteration of the buffer contents would be necessary as the message moved from port to port. However, viewing each port as implementing a separate and independent

[7]This model does not correspond exactly to the actual implementation of the current file system (Chapter 8), but it is a possible implementation.

abstraction seems inconsistent with this premise.

Independent of its abstraction properties, the message buffer does have the advantage of keeping message text out of the address space of the communicating processes. Some important programs (notably the ARPANET control program) have completely finessed the small address space problem by exploiting the "auxiliary address space" implicit in message buffers. This is a major advantage given the large amount of physical memory on C.mmp and the small address space available to programs.

6-9 FURTHER READINGS

Operating systems have been offering message systems and message-style interprocess communication for some time, an early example being the RC 4000 system [Bri70]. More recent systems [Rit74, Che79] strongly encourage or enforce message-style communication as the sole means of exchanging information across process boundaries. The relative merits of procedure calls versus message passing have long been debated; [Lau79] presents a provocative view of the relationship between these communication paradigms.

PART THREE

THE SYSTEM IN USE

CHAPTER

SEVEN

USING THE PROTECTION MECHANISMS

Hydra's philosophy, presented in Chapter 3, is that protection must be an integral part of any general-purpose operating system. A set of protection mechanisms should be part of the lowest level of an operating system, and those mechanisms must be flexible enough to support the wide range of security and reliability policies needed by subsystems and application programs.

Hydra was designed with four interacting mechanisms that together provide a base for supporting a broad spectrum of such policies. These mechanisms have been described already, but we summarize them here.

Procedure invocation. To ensure that a procedure's execution environment is determined by only the capabilities in its own C-list plus those capabilities passed to it by its caller.

Rights amplification. To allow (only) designated procedures to access the representation of user-defined object types.

Rights checking. To restrict the set of operations that can be performed on an object accessed along a path.

Rights propagation and masking. To implement a set of rules for determining how rights are propagated when capabilities are copied.

In this chapter we will see in more detail how these mechanisms can be used to solve a number of common protection problems. The discussion will center on three important kernel rights: *ModifyRts*, *EnvRts*, and *UncfRts.* Before considering special mechanisms, however, we should note that Hydra procedures and the *Call* mechanism provide a means for implementing arbitrary security policies, even in the absence of the other mechanisms involving capability rights. A procedure can act as a *gatekeeper* for an object by holding in its C-list the only capability for the object; where a capability for the object would ordinarily be made available to some user, a capability for the gatekeeper can be made available instead. The user is unable to access the object directly; he can invoke only the gatekeeper, presenting whatever additional capabilities are required for authentication.

7-1 KERNEL RIGHTS

It would be unreasonable to force every user to program every security policy explicitly, as would be required if gatekeepers were the only protection mechanism. The most common protection problems are solved in Hydra through the mechanism of rights checking and amplification, which is applicable to all capabilities. The specific set of kernel rights was designed for Hydra with several goals in mind.

Selective access. A number of operations, such as *GetCapa*, are applicable to any object, and such operations have an associated right (e.g., *GetCapaRts*). The operation is permitted on the object only if the capability for the object has the appropriate right.

Reliability. Even a trusted procedure may not operate correctly due to software or hardware failures. As a result a procedure may mistakenly delete or modify an object. The rights *DeleteRts* and *ModifyRts* help caller and callee guard against such accidents.

Limiting propagation of access. It is possible for a capability to be copied and placed, perhaps improperly, in objects where it would be widely accessible. *EnvRts* is designed to prevent such propagation where it is inappropriate.

Limiting propagation of information. Even when a capability for an object is not made widely accessible, data contained in the object can be copied to a new object which in turn can be made accessible. *UncfRts* is used in conjunction with procedure invocation to prevent this.

We wish to examine the protection mechanism in some detail, and to do this we choose to look at several classical protection problems. The attainment of the goals above allow Hydra to directly solve these problems:

- *The Mutual Suspicion Problem*, which motivates the basic Hydra *Call* mechanism and selective access to objects
- *The Modification Problem*, which addresses the problem of reconciling reliability with privilege
- *The Conservation Problem*, which is the problem of limiting the propagation of capabilities
- *The Confinement Problem*, which is the (harder) problem of limiting the propagation of information
- *The Initialization Problem*, a particularly difficult problem that combines aspects of all the previous problems

7-2 THE MUTUAL SUSPICION PROBLEM

In most operating systems, a user takes a risk whenever he invokes a system utility or a program belonging to another user. He has no way of being sure that the program he calls will not do something disastrous, such as request that the operating system delete all his files. Most users simply take such low-probability risks for granted and rely on backup systems to aid recovery in the unlikely event that disaster should occur.[1] But in a system in which security is important, faith is not enough. Further, in Hydra-like systems in which most functionality is provided at the user level, the probability of errors may be higher. The user needs some way to limit or circumscribe the amount of damage a procedure that he calls can do.

Similar problems are faced by the author of a (possibly proprietary) utility program intended to be called by many different users. The program probably makes assumptions about the format of its private files and therefore wishes to have exclusive access to the files. The programmer needs some guarantee that, except through execution of his program, users cannot access his sensitive data structures or his program code.

This situation is known as the Mutual Suspicion Problem [Sch72]. Restated in the language of Hydra, the problem is this: the caller of a procedure needs a guarantee that the procedure will not be able to gain access to any of the caller's objects, except those explicitly passed as parameters. The procedure (i.e., the owner or maintainer of the procedure) likewise needs a guarantee that the caller cannot gain access to any objects private to that procedure, except when the procedure explicitly allows it. The *Call* mechanism was designed to solve both problems directly.

Execution environments and access privileges in Hydra are not hierarchical. When a procedure is called, the execution environment of the instantiated LNS is the union of two sets of capabilities: those passed to the LNS from its caller, and those inherited from the procedure object. Because the caller cannot access the procedure's capabilities,[2] he cannot tamper with the LNS's inherited environment. Because the called LNS cannot reach the caller's environment, the new LNS's ability to tamper with the caller's capabilities is absolutely limited to those passed as arguments.

The *Call* mechanism actually provides finer control than required to solve the Mutual Suspicion Problem. Not only can the caller control the set of objects that he must allow the callee to access, but by restricting the rights in the capabilities he passes, he can actually control the *kinds* of accesses he risks. He thus has extremely tight access control of his objects.

Technically, there is one exception to this tight access control: if the procedure in question has an amplification template for some type, then it

[1] Of course, trust in the backup systems is still necessary.

[2] Assuming the capability for the procedure lacks *GetCapaRts*, which is typical.

may be able to acquire more rights to an object than the caller passed. However, it should be recognized that the possession of an amplification template for some object type is, more or less by definition, the distinguishing characteristic of the subsystem procedures that implement that type. Subsystem procedures must be trusted to some degree; otherwise the subsystem should not be used. Having said this, we now consider various ways in which a user *can* protect himself from subsystems and restrict even the kinds of operations the subsystem can perform on its own objects.

7-3 THE MODIFICATION PROBLEM

Users often want guarantees that an object passed as an argument to a procedure will not be modified as a result of the call. Ordinarily, it is sufficient to restrict those rights that allow modification (i.e., *PutCapaRts* and *PutDataRts*) before passing the capability for the object as an argument. When a procedure that belongs to the subsystem for the object type in question is called, however, rights amplification may reinstate those rights.

In general, of course, users must trust that a subsystem fulfills its specifications, just as they trust that the subsystem maintainers do not distribute amplification templates indiscriminately. Unfortunately, programs are changed, and trustworthy subsystems occasionally develop bugs. Ideally, both the user and the subsystem want a way to *ensure* correct operation. We cannot do this in general, but we can provide a solution to the Modification Problem, i.e., the problem of ensuring that an object is not modified in any way.

The generic capability right, *ModifyRts*, implements the solution. Each Hydra Kall that modifies an object *in any way* requires a capability with not only the right that allows the specific operation but *ModifyRts* as well. Thus, to store a capability in an object, one must have a capability for the object with both *PutCapaRts* and *ModifyRts*. To put data in the data-part of an object, one needs a capability for the object with both *PutDataRts* and *ModifyRts*.[3]

ModifyRts can never be gained through amplification! A capability lacking *ModifyRts* represents an intention to prevent all modification to the object through that capability. A capability produced by *Merge* (Section 5-3.4) will contain *ModifyRts* only if *both* the amplification template and the original capability have *ModifyRts*.

Because what the user thinks of as a single "object" (e.g., a file) may actually be implemented with many different object types (e.g., pages and semaphores), *ModifyRts* must prevent the modification of the representation of an object as well as the object itself. Hydra therefore ensures that loading

[3]A Kall that modifies the internal structure of a kernel-supported object also requires *ModifyRts*. Thus, *P* and *V* operations on semaphore objects require a capability with *ModifyRts*.

a capability into one's LNS through an intermediate capability that lacks *ModifyRts* masks out *ModifyRts* in the loaded capability (recall the operation of *GetCapa*, section 5-3.2).

In Figure 7-1, the left-hand diagram indicates the environment of LNS *A* at the time it loads a file capability into its C-list via the operation

GetCapa(2,Path(1,1))

As seen in the right-hand diagram, the new capability for the file lacks *ModifyRts* because the capability for the UNIVERSAL object lacked them. This masking means that if a capability for a file lacking *ModifyRts* is passed to a procedure, the procedure will not be able to modify either the file or the page and semaphore objects that make up its representation.

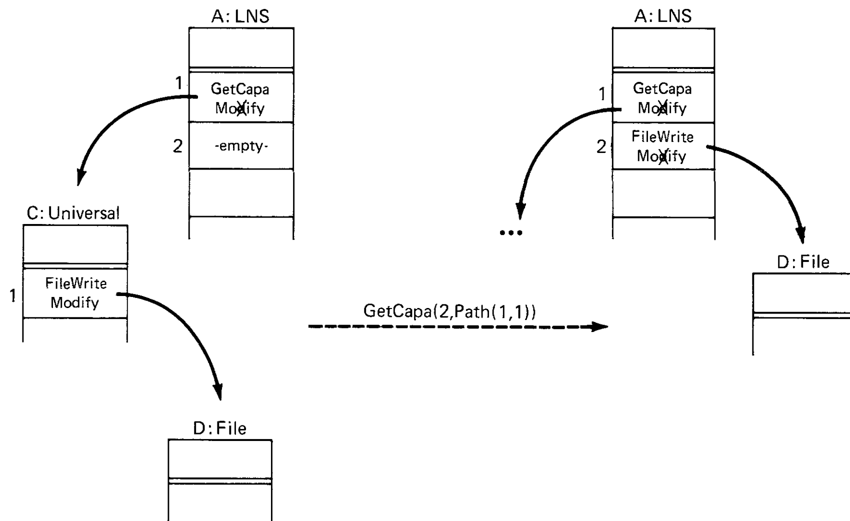

Figure 7-1 Masking *ModifyRts* along paths

The practicality of this solution to the Modification Problem depends on the cooperation of authors of procedures. A procedure can *allow* users to protect themselves only if the procedure can operate without *ModifyRts*, and this does pose some restrictions on the procedure. For instance, a file system's Read procedure may wish to update a "date-of-last-access" field in the data-part of the file. In this case, Read, while conceptually not altering the file, will need *ModifyRts*, and thus users of Read cannot restrict *ModifyRts*. (The author of Read should confirm this by including *ModifyRts* in the required-rights field of the file parameter template.)

7-4 THE CONSERVATION PROBLEM

Assuming that a user accepts whatever risk of modification is inherent in passing a capability to a procedure, there is still a problem of determining when that risk is over. Ideally, the risk should extend only over the lifetime of the procedure invocation; when the procedure returns, the user would like to be able to assess any damage done and, finding none, be assured that he is once again safe. Unfortunately, it is possible for procedures to *retain* capabilities passed to them, and even to pass those capabilities to other processes which in turn could monitor or modify the object at arbitrary times in the future. Thus we have the Conservation Problem, a special case of the problem of limiting the propagation of capabilities.

To solve this problem, Hydra implements another kernel right, *EnvRts*. *EnvRts* must be present in any capability to be stored outside the LNS. Thus, an LNS with a capability lacking *EnvRts* could not invoke the *PutCapa* Kall on that capability. Like *ModifyRts*, *EnvRts* is propagated along paths and cannot be gained by amplification.

The lack of *EnvRts* does not prevent a capability from being passed to a procedure as an argument (though perhaps it should), or returned to a calling procedure as a result, but it does effectively prevent any sharing of the capability with any other user since LNSs are never shared[4] and no capability lacking *EnvRts* can escape from the LNS into a shared object. When the LNS returns to its caller, Hydra deletes all capabilities in the LNS (including the ones lacking *EnvRts*).

Figure 7-2 shows the canonical dangerous situation. LNS *A* is about to call the DIRECTORY subsystem procedure *X*, passing it an object of type DIRECTORY. Unknown to *A*, *X* shares a UNIVERSAL object with another procedure (*Y*), and plans to store a capability for *A*'s directory in that object so that *Y* can access the directory later. To protect himself, *A* restricts *EnvRts* in the directory passed to *X*, thus resulting in the situation depicted in Figure 7-3. LNS *X'* has been instantiated from procedure *X*, but if *X'* now attempts to store the directory in the UNIVERSAL object, the Kall

PutCapa(Path(2,1),1)

will fail because the directory capability in C-list slot 1 of *X'* lacks *EnvRts*. When *X'* returns, *A* is assured that no capabilities for the directory are retained.

As in the case of *ModifyRts*, procedure writers must anticipate that their callers may pass them capabilities lacking *EnvRts* and thus design their algorithms such that the right is not needed.

[4]In fact, it would be nice for debugging processes to get hold of a capability for an LNS, and this is a real problem in Hydra. We do allow an LNS to pass a capability for *itself* to another procedure, but *EnvRts* is always removed from the capability.

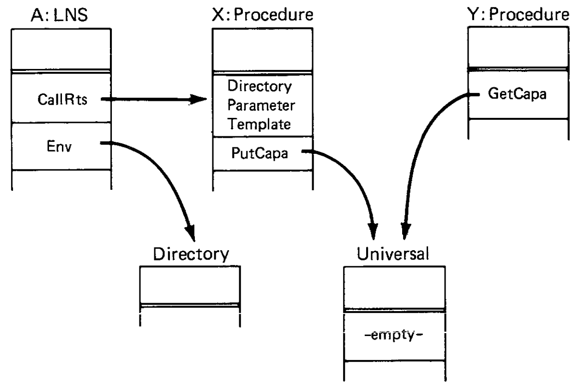

Figure 7-2 The Conservation Problem

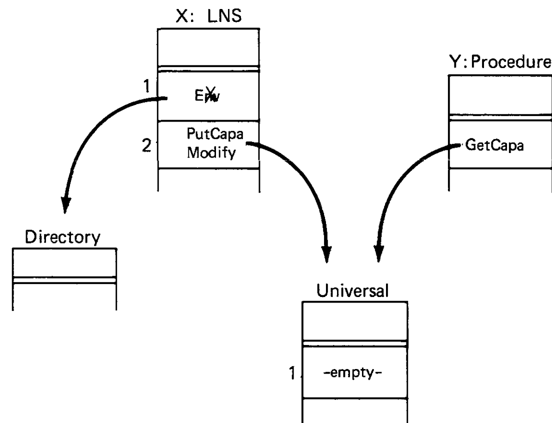

Figure 7-3 The Conservation Problem solved

7-5 THE CONFINEMENT PROBLEM

While *EnvRts* is useful in preventing propagation of capabilities, it is of limited usefulness in preventing propagation (disclosure) of *information* (the Confinement Problem). Even though a capability lacking *EnvRts* may not escape outside of its execution environment, nothing prevents a user from copying the data from the old object to an existing object.

Hydra addresses the Confinement Problem by permitting a procedure to be *confined* at the time it is invoked. A procedure is confined whenever the capability used to access the procedure at the time of invocation lacks *UncfRts*.[5] When a confined procedure is called, all *inherited* capabilities (from

[5]A procedure invoked by *TypeCall* is confined if the type representative in *TypeCall* lacks *UncfRts.*

the procedure) in the new LNS have *ModifyRts* and *UncfRts* removed. Capabilities passed as arguments by the caller are not affected.

Figure 7-4 shows a situation in which LNS *A* wishes to call procedure *X*, passing it file *B*.

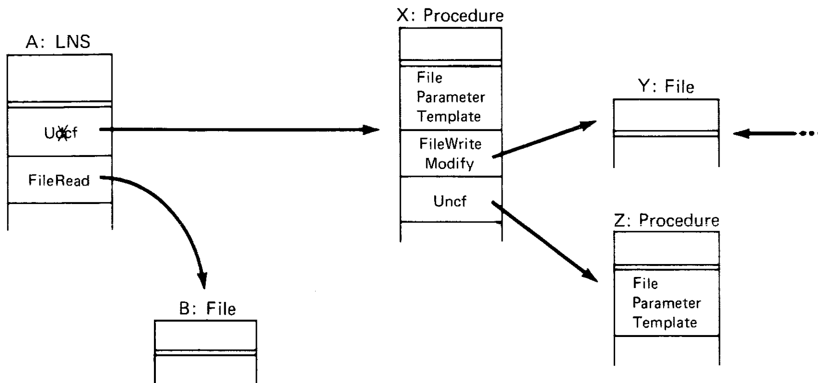

Figure 7-4 The Confinement Problem

Although LNS *X'* must be able to read the file in order to perform its functions, *A* wishes to ensure that the information in file *B* will not leak away, as might happen if *X* were allowed to copy information from *B* into shared file *Y*. Therefore, *A* invokes *X* through a capability lacking *UncfRts*, yielding the situation in Figure 7-5. Although *X'* has all rights to file *B*, it cannot copy the information into file *Y* because it has lost *ModifyRts* to that file. (For the same reason, it could not store a capability for *B* in any shared object.) Furthermore, *X'* cannot invoke another procedure (*Z*) to do the leaking, for it has lost *UncfRts* to *Z*, which must therefore be called confined.

This mechanism solves the Confinement Problem because no information can be copied into any inherited (and potentially shared) objects. New objects may be created and modified, but they themselves cannot be stored into the inherited objects, and so cannot be shared or saved after the return. The only other modifiable objects in a confined procedure are (possibly) the objects passed as parameters. These, being theoretically part of the caller's environment, or at least under his explicit control, should not be dangerous.

Because *UncfRts* is masked out along paths, all procedures the new LNS might call are also confined. Capabilities (with *UncfRts*) for procedures may be passed as parameters and thus called unconfined, because they represent operations that we assume the caller has deemed safe.

In practice, it may be hard to write some procedures so that they can operate confined. The next chapter discusses some of the problems in connection with the design of a particular confinable subsystem.

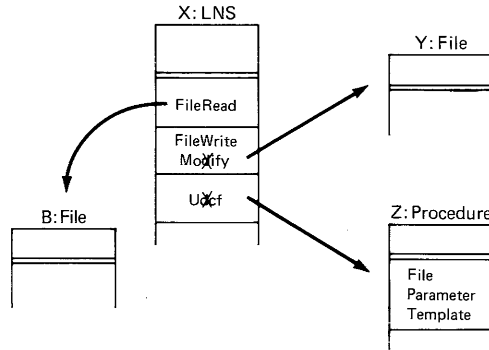

Figure 7-5 The Confinement Problem solved

7-6 THE INITIALIZATION PROBLEM

In conjunction with the Conservation Problem, we showed how a procedure could be prevented from storing away or sharing a capability for an object passed to it. This solution depended upon an assurance that the procedure did not inherit unrestricted access (i.e., with *EnvRts*) to the object. This expectation may especially be violated in the initialization of the object.

Initializing a newly created object entails the generation of its representation. Suppose that procedure FileInit initialized a file passed to it by creating a DATA object and storing it in the file. We want to prevent FileInit from making either the file or the newly created DATA object available to another user.

Restricting *EnvRts* when passing the file to FileInit will not suffice, since a capability for the newly created DATA object could be shared with the unwanted user at the same time it is used to initialize the file. This can be prevented by confining FileInit (calling the procedure without *UncfRts*). In that way, no capability for either the file or the newly created DATA object can be propagated beyond FileInit's environment. Unfortunately, confinement alone is not enough. Instead of initializing the file with a newly created DATA object, FileInit might use a DATA object that it *already* shares with another user.

Hydra solves this tricky problem by removing *EnvRts* at procedure invocation in a way similar to the removal of *UncfRts*. When a procedure is called through a capability lacking *EnvRts*, all capabilities in the incarnated LNS inherited from the procedure have *EnvRts* removed. In the example above, no object already available to FileInit could be stored in the file; only newly created objects (or capabilities passed to FileInit with *EnvRts*) may be stored in it.

To initialize an object safely it is necessary to call the procedure via a capability containing neither *EnvRts* nor *UncfRts*. In that way, we guarantee that any new capabilities placed in the object will be for newly created objects,

and that the entire representation of the object will be unavailable to any other environment.

7-7 RETROSPECTIVE

In a research environment such as ours, restrictive security policies find few active practitioners, but the basic mechanisms of rights checking and amplification are used regularly to restrict access to the representation of user-defined object types. The reason for this is the users' desire for good program structure, rather than security. Subsystems created by users likewise define auxiliary rights and enforce their use, and careful programmers try to design their procedures to work without requiring *ModifyRts* or *EnvRts* in the parameters.

The more advanced mechanisms (e.g., procedure confinement) are used very seldom, and even then mostly as experiments, such as is described in Chapter 8. One problem is that our solutions to many of these protection problems are overly restrictive. For instance, in solving the Conservation Problem, we prevent a capability from being stored in *any* object, not just one that is shared. Likewise, to solve the Confinement Problem, we prevent information from leaving an LNS, whereas it might be more appropriate to prevent information from leaving a subsystem. Unfortunately, there is no way in Hydra of determining when an object is shared, or when a procedure "belongs" to a subsystem.

Another example of the lack of flexibility in protection is in the way *ModifyRts* is removed along a capability path. In directory-like objects, one frequently wishes to ensure that the directory is not modified while allowing the modification of any (or some) objects retrieved from that directory. In Hydra, this policy cannot be expressed using *ModifyRts*.[6]

These protection mechanisms also have a subtle influence on overall system performance. Although the simple checking and amplification of rights has negligible overhead, the more complicated features that require rights restriction on inherited capabilities force these capabilities to be copied from the procedure to the LNS. LNS creation *is* expensive and common, and some obvious optimizations (such as creating an LNS containing just the actual parameters and a capability for original procedure) do not mesh well with these protection mechanisms.

In summary, the protection facilities represent an ambitious attempt to solve a number of complex and subtle problems in the cooperative use of information. The mechanisms we developed offer solutions to several

[6]Originally, Hydra had two rights affecting modification: *ModifyRts* controlled modification of the top-level object only, and another right controlled the modification of the object's representation (and was propagated along paths). We found that this division was not suitable either and folded the two rights into one.

important problems, though perhaps not in the most elegant and efficient way. Nevertheless, the problems they solve are ignored by nearly all other real systems, and the mechanisms they supply to do so are actually used, in practice, by some of the more important Hydra subsystems (including the file system presented in Chapter 8). This confirms the utility and adequacy of these facilities.

7-8 FURTHER READINGS

Hydra's protection mechanisms have the ability to solve a number of protection "problems," as illustrated in this chapter. [Red74] discusses the problem of revocation of access, which Hydra's mechanisms do not address. The confinement problem [Lam73] has been examined by many; its complete solution is known to be very difficult [Lip75]. Other protection problems are considered in [Rot73] and [Coh75]. The relationship of protection mechanisms to security policies and the implementation of specific security policies has been extensively studied; [Lin76] and [Sal75] contain good surveys and substantial bibliographies.

CHAPTER

EIGHT

A FILE SYSTEM

This is the first of three chapters that will examine actual subsystems built on top of the Hydra kernel. In this chapter we will consider a file system which directly addresses some of the protection issues presented in the preceding chapter and which offers a good example of the way that Hydra procedures can be combined with the Message System. The next chapter will consider an application which particularly stresses the Message System, and Chapter 10 will examine some subsystems which implement basic operating system abstractions like "user" and "job."

The Hydra file system was constructed by a team of users relatively late in Hydra's life, after we had begun to understand some of the subtler points of designing subsystems. The basic outline of the facility was fairly obvious from our previous experience with Hydra and other existing file systems. In particular,

1. A "file" would have to be represented by a new type of object so that it could be protected in the canonical Hydra fashion. Certain operations on files (discussed later) would be implemented with procedures and *TypeCall.*
2. File I/O would have to occur via the Message System because the overhead of *Call* would be too high to impose on each read or write. (To give the user direct access to the file's representation would be an unacceptable violation of protection principles.) By using an asynchronous server process to manage file representations, we could both exploit C.mmp's parallelism and make file I/O resemble I/O to peripheral devices. (The server's port simulates a DEVICE object—see Chapter 14.)
3. We would have to allow for different methods of data representation within files because we could not predict in advance what representations would be appropriate for future applications. Although the traditional Hydra response to this problem would be to construct several independent file systems, we felt there were advantages to be gained by accommodating different representations within the same subsystem.

At the time the file system was designed, we wanted to demonstrate clearly the power of Hydra's protection mechanisms. For that reason a principal goal of the file system was to solve two of the classical protection

problems presented in the Chapter 7: the Modification Problem and the Confinement Problem. We can restate these problems using file terminology as follows:

The Modification Problem. Suppose a user wants to grant "read-only" access to a file. How can he prevent the file system from modifying the file in any way? Hydra allows a capability for such a file to be passed to the file system procedures without *ModifyRts.*

The Confinement Problem. Conversely, suppose a user has a file containing sensitive data. How can he ensure that the file system will not "leak" information from the file to the outside world? The file system described below allows successful exploitation of Hydra's notion of "confinement" to solve this problem.

The Confinement Problem was first posed by Lampson [Lam73]; very few systems even attempt solutions to it. Often such systems assign a sensitivity-level, from a partially ordered set of levels, to each datum and ensure that information flows only in ways determined by this partial ordering [Lip75]. This scheme models the military security system and thus represents a special case of the general problem stated by Lampson. We are able to do somewhat better in Hydra.

8-1 FILES AND SUBFILES

Responsibility for handling files is distributed between two subsystems: the FILE subsystem and any one of several SUBFILE subsystems. The FILE subsystem handles protection and synchronization issues not related to data representation. The SUBFILE systems are concerned only with data representation and I/O with the user.

This division has several benefits. By allowing for several independent subfile systems, we leave open the difficult representational issues. By having a common top-level interface (type FILE), we promote uniformity and allow the choice of representation to be ignored in higher-level software.

To get an overview of the file system, we can examine it at several levels. When a user wishes to establish communication with a file, he passes his file object and a capability for a port to a FILE system "open" procedure. Using mechanisms invisible to the user, the file system will connect the user's port to a server process which will process I/O requests from the user.

If we examine the design of the FILE subsystem, we see that it acts as a kind of intermediary between the user who accesses the file and the subfile system that manages the data in it. A FILE object actually contains only a semaphore and a single subfile. (See Figure 8-1.) When the user passes his file to the file system, the file system extracts the subfile capability from the file and passes it to a subfile system "open" procedure. This subfile

procedure will return a capability for the server process' port, and by connecting the user's and server's ports the file system establishes the communication path.

At the bottom level, the subfile system has complete freedom in the establishment of server processes. It could create a process and port for each file, or it could pass the subfile to a single process which managed requests between several users and subfiles. Likewise, there is no restriction on the data representation used by the subfile system. A common implementation is shown in Figure 8-1; the subfile simply stores the data in a list of page objects stored in the subfile. Alternatively, a subfile system could store data directly on a disk or tape, using the subfile to hold disk addresses or tape volume numbers.

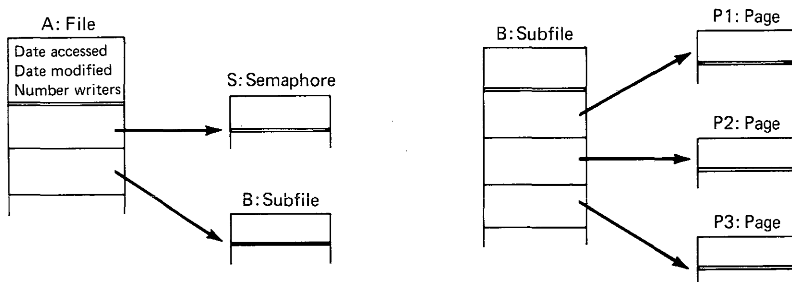

Figure 8-1 The representation of files and subfiles

There are a few implementation issues surrounding this system which should be explained.

Synchronization One of the crucial questions in the design of any file system is the atomicity of transactions to the file. Can readers and writers share the same file? Do changes made by writers appear instantly to readers? Our design for Hydra's file system closely paralleled the semantics of the TOPS-10 file system with which we were most familiar; to wit:

1. Any number of readers and at most one writer may simultaneously access a file.
2. Readers and writers must explicitly initiate and terminate their transactions; i.e., they must *open* and *close* files. (The *open* operation, in addition, must specify "for reading" or "for writing.")
3. *Open* and *close* operations on a file are atomic. When a reader successfully opens a file, he will be unaffected by any modifications by a writer. A writer's modifications take effect only when he closes the file; subsequent openers get the new version.

The file system stores in the data-part of files the number of current readers and writers. (This data is protected during concurrent operations on the file by the semaphore associated with each file.) When a user opens a file, this data is used to decide whether to allow the user his desired access.

To provide isolation between readers and writers, the file system enforces the convention that subfiles are never modified once the file system is given a capability for them.[1] This convention has two important consequences:

1. Once a reader extracts a subfile capability from a file object he can be sure the subfile will not be modified.
2. The subfile system must create a new subfile when a writer attempts to modify the subfile.

Closing files Because a file may be simultaneously read and written by several users, the file system must have a way to determine which transaction is involved in a "close" operation. Therefore the file system implements a second object type, OPENFILE. (Likewise, subfile systems implement OPENSUBFILES.) This object encapsulates any necessary information needed during *Close*, typically the original file and the user's port.

8-2 OPERATIONS ON FILES

To summarize the file system structure, we present the detailed specifications of the file system operations. We will use the same format we used earlier for kernel Kalls; in fact they are invoked with *TypeCall*.

8-2.1 File Operations

OpenForWriting(D:slot(index), F:object(FILE,FileOpenRts),
P:object(PORT,PortConnectRts), C:integer)

Connects file F to output channel C of port P. Returns in slot D a capability for an OPENFILE object.

OpenForReading(D:slot(index), F:object(FILE,FileOpenRts),
P:object(PORT,PortConnectRts), C:integer)

Same as *OpenForWriting* except for reader/writer synchronization.

[1]The file system enforces this by not including *ModifyRts* in any subfile capability presented to the subfile system.

Close(F:object(OPENFILE,FileCloseRts))

Close disconnects the user's port from the server's port and updates the reader/writer information in the file. Additionally, if the file had been open for writing, the current subfile capability in the file will be replaced with the new subfile returned from the subfile system's *SubfileClose* operation.

8-2.2 Subfile Operations

Each subfile system must implement the following operations. The file system invokes these operations "blindly," with *TypeCall*.

SubfileWrite(D:slot(index), P:object(index,PORT),
S:object(SUBFILE)) returns InputChannel: integer

Invoked only by the file system. The subfile system performs any necessary actions to locate or create a server process, passes to the process the subfile *S*, and returns a capability for the server's port in slot *P*. A capability for an open subfile object is returned in slot *D*.

SubfileRead(...)

Like *SubfileWrite*, except the subfile system may take advantage of the additional information that no write operations will occur.

SubfileClose(D:slot(index), S:object(OPENSUBFILE,CloseRts))

Invoked only by the file system. Informs the server process that I/O will cease. Returns in slot *D* a capability for a new subfile if the user had been modifying the subfile.

8-2.3 File I/O

Different data representations could demand quite complex message formats for file I/O to take advantage of specialized structures (e.g., "return the object code of routine Test in module ListPackage" might be an appropriate operation for a subfile which implemented a format to be used by program libraries). To try to encourage standardization without restricting subfile creators, we agreed on a few standard operations ("read/write the next group of characters," for instance) and left new subfiles the freedom to implement other operations with other message formats. We will not discuss them further.

8-3 IMPLEMENTED SUBFILES

To date, there are three subfile systems in active use:

The *SOS Subfile* system defines a representation convenient for the line-number-oriented *SOS* editor on C.mmp.
The *Line Printer Subfile* system implements an output file type. Closing one of these subfiles causes the data to be passed to a line printer spooler process which will print the file on the system line printer.
The *Random Subfile* system implements a random-access byte representation especially suited for some types of intermediate files used by various programs.[2]

The particularly flexible implementation of file I/O as interprocess communication makes possible some other subfile types which have been proposed but not implemented, including

The *Terminal Subfile* system, which would translate file I/O to terminal I/O, thus providing a uniform I/O protocol among files and terminals.
The *ARPANET Remote Subfile* system would implement files whose representations were actually on different computers. Opening one of these subfiles would establish the appropriate network communication, and I/O operations would move data over the network.

8-4 PROTECTION

There are two ways in which the design of the file system is influenced by the desire to solve the Modification Problem.[3] First, the file system must alter its operation when given a capability for a file lacking *ModifyRts*. Whenever a file system operation has no intrinsic need to modify the file, it detects when the passed file parameter lacks *ModifyRts* and functions correctly without modifying the file in any way. Thus the *OpenForRead* operation refrains from modifying the date-of-last-access field in the file object when the capability lacks *ModifyRts*.

A second, and more subtle, influence of the Modification Problem involves the mutual exclusion semaphore that controls access to the file object during *Open* and *Close*. (*P* and *V* operations on semaphores require *ModifyRts*.) An analysis of *OpenForRead* reveals that the only necessarily indivisible operation on the file object is the copying of a capability for the

[2]The Random Subfile also prompted the addition of *OpenForUpdate* operations on files and subfiles.

[3]This section and the following retrospective are based on previously published evaluations by the File System authors [Alm77].

current subfile object. This operation (the *GetCapa* Kall) is already indivisible, so no explicit locking is necessary and *ModifyRts* is not needed. Were it necessary to determine the current subfile by means of access to a multi-word data structure in the file object, on the other hand, some external synchronization, and thus *ModifyRts*, would be needed.

The solution of the Confinement Problem was both more difficult and more interesting. Although it was not difficult to implement a FILE subsystem whose procedures could be called confined (i.e., without *UncfRts*), it was difficult to construct confinable subfile systems. In the case of the Line Printer Subfile system, there is the intrinsic need to modify a particular inherited object, the line printer device. In other cases, the need to modify is not intrinsic, but technological, and stems from the customary technique of multiplexing a single server process among all the open subfiles of a particular type. Given this efficiency-oriented shared-server concept, confinement is impossible.

One particular subfile system was constructed explicitly to explore the feasibility of a confinable subfile system. The *SubfileOpen* procedure of this so-called "Confinable Subfile" system creates a new server process for each open subfile. Due to the confinement constraint, the initial LNS of this server process will be confined and may modify only its parameters (which in turn must have been parameters or locals of *SubfileOpen*). The Policy Module procedures which create new processes cannot be called confined, and so the caller of *Open* must pass in an additional argument—a type representative for the Policy Module. This parameter is propagated down to *SubfileOpen*, which uses it to call the *MakePMProcess* procedure (see Section 10-1.3) *un*confined. This technique is acceptable, since passing the Policy Module type representative is equivalent to the user explicitly "certifying" his trust that the Policy Module will not leak information.

A point can now be made about the importance of the Modification Problem. The most obvious motivation, that given in Chapter 7, is that a user should be able to attempt a read-only access to a file without any risk of its corruption, as in the case when a user suspects a file system bug. This is not a forceful motivation, however, because such a situation would occur very rarely in such a critical subsystem. A more convincing motivation stems from the desire to make a solution to the Confinement Problem a practical reality. A confined call to a subsystem will fail unless the subsystem can effectively get its work done without *ModifyRts* and *UncfRts* in its inherited capabilities. If, for example, a subsystem needs to read a file (or look up a read-only item in a directory) and if the file (or directory) system did not solve the Modification Problem (it might fail by insisting on being able to update a date-of-last-access field), then it would be impractical or impossible for the subsystem to function confined. Thus, any subsystem that intends to solve the Confinement Problem in a practical way must also solve the Modification Problem.

8-5 RETROSPECTIVE

The distinction between file and subfile objects has made the file system highly extensible. This kind of extensibility is very important in experimental computing environments, like Hydra, where representational issues are open-ended. We viewed the successful implementation of the file system as welcome evidence that the underlying mechanisms were sufficient for real systems whose designs we might not be able to foresee. After all, we had not anticipated the file/subfile distinction which arose for the file system. We had always expected that each separate file representation would give rise to a separate and wholly independent file system.

The file system and the three subfile systems mentioned earlier have been operational since early fall of 1976. The design of these systems took place during May and June of 1976 and involved six people for a total of about 14 man-days. The implementation of each of these systems took less than one man-month. Each was written in Bliss/11, and the code for all procedures in each system totals about 3,500 words (about 2,300 instructions). Coding, compiling, and most of the testing of the file system and various subfile systems were done independently. The only testing that required coordination between two implementors occurred when communication between the file system and a subfile system was in question. This amounted to only a few man-days in each case.

While the Hydra file system design is functionally pleasing, its implementation is slower than we would like it to be. This is a natural result of using protection mechanisms as general and powerful as those provided by Hydra. Most of the overhead in the system is due to the *Call* mechanism during *Open* and *Close.* During a measured series of calls to the file system, the average amount of computation by user code within the file system was 49 ms and the average cost of *Call* was 89 ms (i.e., 65% of the total computation time).[4] Thus, while the domain crossings did make the protection and software engineering results possible, they are expensive. File I/O, which uses the Message System, is much faster.

Finally, it is interesting to note that the file system does not occupy as important a position in Hydra as does, say, the Catalogue subsystem, which provides named access to arbitrary capabilities. This is partially due to the fact that there are few language processors on C.mmp, and hence there is little need for program source files or the compiler's intermediate files. An even more important reason, however, is that files are not really a natural abstraction for all long-lived objects. In Hydra, where any object may persist, the user is more likely to use a direct implementation of the abstraction he desires. For instance, when the need arises to store a small amount of data, one usually tries first to encapsulate the data in the data-part of some object.

[4]These times were subsequently improved somewhat; see Chapter 16 for more details.

If that is too small or cumbersome, the data can be stored in a page. If even more data is required, pages can be strung together in a universal object. By doing this, one is able to address his data directly (in 8K-byte chunks) and doesn't have to interface with the Message System. These alternatives pose interesting tradeoffs. Access is more efficient, but each programmer generally has to reinvent the whole system. It is also difficult to share such data collections in the relatively rare cases where sharing is desired, because no synchronization is provided. Thus, because Hydra offers several data storage mechanisms whose functionality and performance vary significantly with the amount of data, the file system is less heavily exploited than in systems where the options are more limited.

CHAPTER

NINE

A NETWORK CONTROL PROGRAM

In Chapter 6 we presented the facilities of the Hydra Message System and motivated its structure with a small example. In practice, however, the Message System has considerable functionality that is not needed in simple situations. Only when one attempts to coordinate a substantial collection of asynchronous activities does one encounter the problems that motivated the more complex features of the Message System. In Hydra's case, the ARPANET control program was the application that inspired most of those features. This chapter examines the problems inherent in ARPANET communication and shows how the message system is used to advantage in solving them.

9-1 THE PROBLEM

Before we consider the internal structure of the ARPANET control program (henceforth called the NCP), let us examine the requirements it must satisfy. To do so, we need to understand the abstraction supplied by the ARPANET interface and the abstractions a user-level program that wishes to use the network would like to see.[1]

The ARPANET [Hea75] is a collection of message processing computers (called *IMPs*) interconnected by dedicated communications lines. Each IMP serves as an interface for one or more *host computers* to the communications network. Information is transmitted between host computers in the form of *messages*, which are variable-length blocks of bits. The content and sequencing of messages is determined by a number of *protocols*, most of which are unknown to (i.e., uninterpreted by) the IMPs. The goal of the network control program (NCP) is to implement the lowest level of host-to-host communication (the *Host-Host Protocol*), which defines a standard method of communication through the network for heterogeneous computers. At the

[1]The subsequent description of the ARPANET and its functional capabilities is necessarily simplified. Wizards and cognoscenti will recognize many places where subtle problems are glossed over or ignored. It is our purpose to present the network structure and function at a sufficient level of detail that the reader can appreciate the problems facing the implementor of an NCP.

level of the host-host protocol, the details of interfacing to the IMP are hidden. Thus, the NCP also must assume complete responsibility for communicating with the IMP. In fact, the NCP implements all the host-host protocol as well.

Most protocols rely on the notion of a *connection.* A connection is a unidirectional "virtual circuit" between two *sockets.* A socket is thus the entity that a program uses to communicate with another program in the network. In principle, a host supports an arbitrary number of simultaneous connections.[2] Since communication is usually bidirectional, sockets are often in grouped in pairs, one each for input and output.

Although users of the ARPANET tend to think of sockets as the primitive connection mechanism, there is, in fact, a lower level. The IMPs implement the notion of a *link number,* which is used to multiplex simultaneous logical "conversations" (e.g., socket-level connections) over the single physical IMP-host channel. The *IMP-Host protocol* defines a small, fixed-length *header* on every message that contains a source and destination "address" for the message. These addresses are pairs (host number, link number). Each host's NCP must manage the set of link numbers for its host, except that link 0 has a special meaning. For the host-host protocol, a connected socket within a host has a unique link number assigned by that host's NCP.[3]

It is worth emphasizing the distinction between link numbers and sockets. A link number is used exclusively within the NCP, since it is a part of the IMP-Host protocol. With the exception of link 0, a link number has no implied meaning; it is simply a local identifier for a conversation in progress. On the other hand, a socket is, at least in principle, a long-lived entity. Specific socket numbers may be advertised as offering particular services (e.g., file transfer). In fact, higher-level protocols tend to treat a socket (pair) much like a telephone; that is, it can be connected to some other "phone" in the network for a time, then disconnected and reconnected somewhere else. It can "talk" only to one place at a time. There are protocols (e.g., ICP, the *initial connection protocol*) that establish these connections and define analogs to the familiar telephonic notions of dialing, "busy" signal, no answer, hung up, and call queuing. We will discuss these in more detail later.

Thus, an NCP serves, in essence, as a connection manager. It provides mechanisms for: establishing and breaking connections, creating and destroying the "telephones" (sockets) that define the participants in a conversation, and transmitting data over connections. In doing so, it provides a consistent abstraction for communication between host computers while masking the details and most of the complexity of the underlying communication network. This implies that the NCP must perform a substantial amount of error

[2]In practice, the IMP imposes an upper bound because of its limited internal resources.

[3]By limiting the number of permissible link numbers, the IMP effectively limits its number of simultaneous connections, as suggested earlier.

processing, much of which is rather complicated. In the following sections, we will, on the whole, ignore the problems of robust communications, except where they have a noticeable effect on the structure of the NCP.

9-2 THE HYDRA NCP

The Hydra NCP provides a comprehensive example of the use of the Hydra message system. In fact, the design principles underlying the NCP implementation tend to emphasize the importance of the message system facilities. Specifically, the NCP's organization follows from two primary assumptions:

1. The "boundary of trust" is the socket-level interface. That is, all the mechanisms that implement the facilities needed to support sockets and connections are mutually trusting. This doesn't mean that the various NCP components have intimate knowledge of each other's data structures; on the contrary, they are rather well isolated from each other, and validate scrupulously the data that passes between them. They do not, however, rely explicitly on the Hydra protection mechanisms for that isolation. In contrast, the socket interface is a Hydra subsystem (i.e., SOCKET is an object type), and user programs manipulate sockets by invoking Hydra procedures in the socket subsystem.
2. Within the NCP, all communication among components is performed using Hydra messages. Each logically distinct data structure of any significance is managed by a single process, and only in response to requests submitted in messages. Thus, no synchronization mechanisms are employed other than the one implicit in the sequential reception of messages.

With these principles in mind, let us proceed to examine the organization of the Hydra NCP. The reader may find it helpful to refer to Figure 9-1 during the following discussion. In the figure, the major communication paths are shown as solid lines, representing port connections. The solid lines between a user and the IMP interface are exercised whenever data is exchanged in a conversation with a remote socket (pair). Two independent socket (pairs) are shown. The dotted lines represent port connections used for connection control and are less heavily exercised. The following sections discuss these paths in detail.

9-2.1 IMP-Host Communication

One of the C.mmp processors has a peripheral device that provides a full-duplex, DMA connection to the IMP. We can think of this interface as two independent bit-stream channels, one for input and one for output. Hydra provides access to this hardware interface in the usual way, by

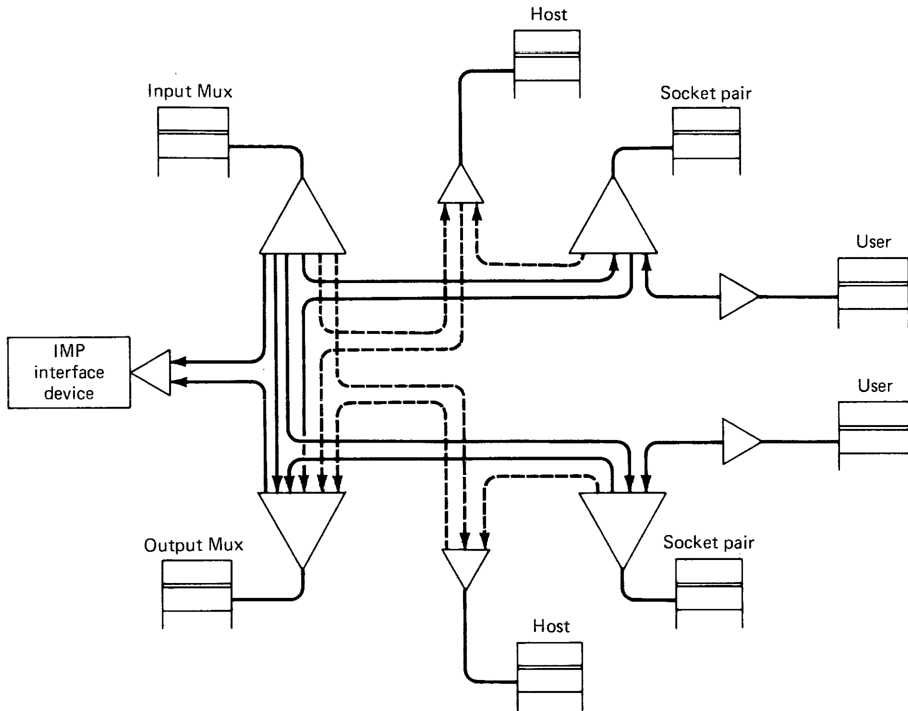

Figure 9-1 The NCP communication structure

predefining a device object to which messages may be sent requesting physical I/O. Since Hydra messages transmitted through this port represent direct communication between the host (C.mmp) and the IMP, they must observe the IMP-Host protocol. Thus, the network source and destination are represented in these messages as (host number, link number) pairs. Recall that link numbers are used to multiplex simultaneous conversations. Accordingly, the NCP has a process, labeled "Input Mux" in Figure 9-1, that accepts incoming messages from the IMP and demultiplexes them. As we will see shortly, this generally means mapping the destination link number to its corresponding socket. Similarly, the NCP has an output process, "Output Mux," that accepts outgoing messages from various sources (generally sockets), and forwards them to the IMP, inserting the source link number as required by the IMP-Host protocol.

Together, the input and output multiplexors provide a simple, message-oriented communication facility over a single physical channel. To see this, let us consider the operation of these processes in a bit more detail. Each one has a single Hydra port. The input process sends input requests to the IMP interface port, where they remain queued until input arrives. An incoming network message is placed in the buffer of one of these Hydra

messages and replied, causing it to return to the input multiplexor's port.[4] The input process examines the destination link number, maps it to an output channel number, and sends the message out on that channel. It then waits for subsequent input. When the recipient of the message eventually replies it, the message is destroyed.

The output process is slightly more complicated. From the viewpoint of a socket, the output multiplexor does not reply to a message requesting output until the data is successfully received at the destination host. Thus, when the output multiplexor receives an output request, it forwards it to the IMP interface. The message returns (is replied) to the output multiplexor when the IMP has accepted its contents. However, since the network may fail to transmit the message successfully, the output multiplexor cannot yet reply to the output request. Instead, it temporarily places the request in a local queue. Eventually, a notification will arrive from the IMP indicating whether the message was successfully received by the destination host.[5] Upon receiving the notification message, the output multiplexor removes the original request from its local queue and replies it to the requester. Thus, the originator is informed of the outcome, successful or not, of the attempt to transmit his message to its remote destination.

9-2.2 Host-Host Communication

Given the abstraction provided by the input and output multiplexors, communication between connected sockets is straightforward. (We will consider the protocol for establishing connections shortly.) Connected sockets use the Host-Host protocol and pass streams of data bytes (of a mutually acceptable size) between them. Recall that a connection between sockets is unidirectional, so that "conversations" typically involve a pair of sockets. The NCP typically creates a single process with a single port to manage a socket pair. The port is connected to the input and output multiplexors at the time that the socket (pair) is created. The socket process accepts requests from a user program and, after appropriate transformations and validity checks, communicates them to the multiplexors. Recall that these requests are beyond the "boundary of trust" and therefore must be treated with suspicion by the socket process. It must detect and report protocol violations and prevent runaway user programs from clogging up the NCP's internal communications.

[4]This is a simplification. Because of various buffer size constraints, a single network message may be fragmented into more than one Hydra message and subsequently reassembled. We will ignore this complexity here.

[5]This notification is, in fact, an input message, and so is naturally received by the input multiplexor. The input multiplexor recognizes the message as an acknowledgement of an output message, and forwards the notification to the output process. This connection from input to output multiplexor is shown in Figure 9-1 and is the only interaction between these otherwise independent processes.

The socket process actually handles the message fragmentation alluded to earlier. The interface it offers to its user, however, is purely stream-oriented.

9-2.3 Connection Management

Up to this point, we have been considering the major communication paths within the NCP. As we have seen, the interconnection of ports and processes naturally follows the requirements of the IMP-Host and Host-Host protocols. However, a certain amount of complexity is introduced by additional protocol requirements that deal with *connection management*. These include the mechanisms for establishing a socket-level connection, controlling the flow of information along the connection, and passing control signals that are asynchronous with the data flow through the connection. We will sketch enough of these requirements to illustrate their effect on the structure of the NCP.

Requests for connection (RFCs) arrive on link number 0. (Recall that this link is reserved for special purposes—this is one of them.) An RFC specifies the socket number to which the connection is to be made and, of course, includes the host and socket requesting the connection. The input multiplexor demultiplexes all link 0 messages by originating host. It expects to have an output channel for each such host, leading to a port managed by a *host process*. (If no such port and process exist, they are created dynamically.) The host process determines whether the socket number requested corresponds to an extant socket object. If so, the RFC is forwarded to the socket.[6] The socket may accept, reject, or queue the RFC. In the first two cases, the socket replies to the RFC message, causing it to return to the host process where appropriate link 0 messages are generated to either complete or terminate the connection protocol.[7] If no socket can be found by the host process, it rejects the RFC directly. Thus, socket objects are never implicitly created in response to RFCs. (A higher level of protocol, the Initial Connection Protocol, offers a mechanism for dynamic socket creation.) When a network connection is successfully established, a message system connection between the socket and the host corresponding to its remote socket is established as well. The purpose of this connection will become clear momentarily.

A socket-level connection is primarily intended to support a unidirectional, stream-like flow of data bytes. However, it is occasionally necessary to

[6]The socket, in turn, may not have an associated port and process. If necessary, these are created dynamically and appropriately initialized.

[7]For internal technical reasons, there is actually an "RFC" port between the host port and the socket port. This port and its associated process exist to simplify the implementation and because they manage a connection data base that is logically distinct from the host processes. For ease of exposition, however, we will not consider the RFC process to be a separate entity from the host processes.

pass control information as well, e.g., to control the amount of data transmitted or to interrupt the send or receiver. Logically, this information is asynchronous with the data flow and thus must be transmitted along a logically distinct path. There are many ways to implement this separate path; the Host-Host protocol chooses to use link 0. Thus, *all* control messages for all socket connections are transmitted on link 0. The purpose of the connection between a socket port and its corresponding host port is now evident; the host process passes control messages along this path.

9-3 RETROSPECTIVE

The NCP took a long time to build, in part because it was the first sophisticated user of the Message System facilities. In fact, several features were added to the Message System because they eliminated difficulties in the NCP. In the end, the resulting structure of the NCP was determined largely by informational requirements of the ARPANET protocols and only minimally affected by the quirks of ports and messages. It is appropriate, therefore, to summarize what we learned about interprocess communication by building the NCP.

The internal organization of each NCP process is extremely straightforward. Each has a single outer loop whose body receives a message, processes the message contents, and finally replies the message. Of course, each process has internal state information, which affects the precise parameters to the *ReceiveMsg* operation, the particular processing of the incoming message, and the final disposition of the message in hand. This overall structure is easy to understand and modify, so it is worth understanding the message system facilities that permit this organization.

First, the *RequeueMsg* operation (see Section 6-6) enables the NCP to do all its queue management using the Message System. The NCP simply treats an input channel as a queue and uses *RequeueMsg* to add messages to it. *ReceiveMsg*, with appropriate parameters, permits dequeuing either a selected element or in FIFO order. Thus, no explicit queue structures exist in the NCP.

Second, the arbitrary fan-in and fan-out allowed by the message system's port interconnection facilities permits the NCP to model easily the multiplexing requirements of the network protocols. (The implicit reply mechanism is also vital in this regard.) Fan-out (through output channels) enables a single process to direct messages to several distinct destinations, depending on their contents. This is convenient for the host processes, which receive a variety of unrelated requests on link 0. Fan-in (through an input channel) allows a single process to service requests from several distinct sources. The multiplexor processes do just that. Typically, the connections that "fan-in" to a particular input channel are labeled (using the message system's connection

identifiers) with convenient identifiers, such (host, link) pairs. Whenever a message arrives, the relevant connection identifier is supplied by the message system to the receiver of the message. This substantially simplifies the receiver by eliminating certain mapping tables in the NCP, since the message system is, in effect, storing the map implicitly.

Third, the ability to process and store messages in a port in an essentially arbitrary order permits the NCP to define three rather flexible styles of communication among its components. The simplest form of message is one that is received, immediately processed, and replied before another message is received. This simple sequential style of processing is used by the output portion of the IMP interface. In other cases the receiving process may have to wait for some other event before replying to the message. The input side of the IMP interface exemplifies this situation, since a read request cannot be replied until the data it requests has been received from the network. Even in this case, however, the simple loop paradigm is adequate, since such requests are still handled in strict sequence. However, it is occasionally useful to allow such requests to be kept "off to the side" until the information they request is available. This situation arises in handling asynchronous error conditions, where a message is sent requesting acknowledgement of an exceptional condition. The receiver holds the message in abeyance (in a local name of a port or by requeuing) and replies to it only when (and if) an exception arises.

Two message system properties, however, have adverse effect on the NCP's structure. First, the Message System imposes a resource limit on each port. This number is determined by the port's creator, but remains a fixed upper bound for the life of the port. This can be inconvenient for certain ports (in particular the input multiplexor) because they must be prepared to handle essentially arbitrary data arrival. Peculiar communication structures tend to arise from this. Input, for example, is generally reflected as *replies* to input *requests*, rather than original messages containing the input. In short, the Message System does not alleviate the well-known problem of buffering real-time input.

Second, message buffers have no implicit stacking mechanism analogous to the stacking of reply frames. This tends to force the NCP to transform the data in the buffers at each level of message transmission. Although in principle the amount of data manipulation can be substantial, careful design of the message formats eliminates most of these transformations.

In summary, we believe the NCP demonstrates that the functionality of the Hydra Message System is worth its occasional complexity. Several unpleasant details in the ARPANET protocols that few other NCPs implement were easily accommodated under Hydra. At the same time, the implementation seems to run a single connection about as well as a uniprocess implementation would, suggesting that the overhead of multiprocessing does not significantly affect the performance of the NCP's task. Further-

more, the NCP's multiprocess, multiport structure adapts comfortably to C.mmp and, in fact, is one of the few large programs that is unaffected by the 16-bit address space limitation. This, in itself, is an important affirmation of much of the Message System design.

CHAPTER

TEN

A USER-LEVEL OPERATING SYSTEM

The previous two chapters have given a detailed picture of two subsystems built over Hydra. In this chapter we will give a somewhat more high-level description of the subsystems which have evolved into the "operating system" seen by users of C.mmp. In so doing, we will try to substantiate one of our major claims for Hydra: that it is easy to construct operating system facilities which are non-preemptive, that is, which can be replaced or removed by any user dissatisfied with the facility as supplied, without affecting other users.

To describe these operating system subsystems, we will make use of a simple example. We suppose that a user wishes to log on to the Hydra/C.mmp system and print a file. First, let's simply observe a record of the terminal session; we will analyze it in more detail later. In the following scripts, the text typed by the user is italicized; all other text is typed by the operating system.

CMU CS Front End
Host: *c*
Welcome to Hydra/C.mmp
Job Monitor
V5.09 6-Nov-1978 18:54:25
Type "Help" If you need it
@*log*
Job 21 Hydra/C.mmp TTY13 On 7 Aug 79 12:33:47
Name: *harbison*
Password:
Command Language
V1.28.1 2-Aug-1978 10:53:55
>

Our user is now logged on to the system and can invoke various commands to do his work. ('>' is the Command Language prompt character.) He might, for instance, list his private catalogue with the "list directory" command:

```
>di()
Fortran      Procedure    (377,160002)     8-Mar-79
Letters      Directory    (377,160002)    14-May-78
Profile      Commands     (377,162102)     5-Jan-79
PrintFile    Procedure    (172,160002)    23-Feb-78
Public       Catalogue    (377,160002)    15-Jul-79
RootFinder   File         (377,160002)    11-Aug-78
>
```

A user's catalogue contains named capabilities for arbitrary objects. Here we see six: a file, two procedures, a "commands" object, a directory, and a catalogue. Listed for each entry are the capability's access rights and the date the capability was entered into the catalogue. *PrintFile* is a capability for a procedure which lists its single file argument on C.mmp's line printer, so the user may type:

```
>printfile(rootfinder)
```

to print the file. Note that the command language interprets "printfile" and "rootfinder" to be catalogue entries in this context. Procedures may also be invoked as separate processes; our user might, for instance, invoke the Fortran compiler asynchronously on the same file by typing:

```
>capa &process
>&process = $MakePMProcess( fortran, rootfinder )
>$StartPMProcess( &process )
>
>$StatusPMProcess( &process )
Running
>
>$StatusPMProcess( &process )
Stopped
>
```

To log off the system, the user types a "break" character (represented by the character "*"):

>*Back to Jmon, remember to Kjob
@*kjob*

CMU CS Front End
Host:

10-1 ANATOMY OF THE USER-LEVEL OPERATING SYSTEM

With the possible exception of the command syntax and the spawning of asynchronous processes, the terminal session above resembles those encountered on most timesharing systems. Hydra is distinguished more by the *way* the functions are provided than by the *kind* of functions. In fact, at no time in the above example was the user communicating with the Hydra kernel directly. All the services were provided by user-level subsystems. We now go over the same example again, this time explaining the underlying mechanisms.

10-1.1 Connecting to the System

The Hydra kernel has no concept of "user" or "terminal"; it simply provides a set of device objects for the terminals. When the system is initialized, the user-level *terminal multiplexor* procedure, TMUX, is invoked as a separate process and is passed capabilities for all the terminals. TMUX connects a port to all the terminals and waits for something to be typed at one. It is TMUX's main responsibility to multiplex a single terminal among multiple processes, a useful function in a multiprocessing environment such as this one.

TMUX has the additional responsibility to detect an initial connection to C.mmp and to invoke higher-level software. TMUX possesses a capability for the *job monitor* procedure, JMON. When the Front End processor connects a terminal to C.mmp, it sends a special "connect" character which causes TMUX to spawn a JMON process connected to the user's terminal. From this point, TMUX simply moves characters between JMON and the user's terminal; its special responsibilities are over. If another user connects to C.mmp, TMUX will spawn a new JMON process for him from the same JMON procedure. The connection structure is shown in Figure 10-1.

When the JMON process is started, it immediately types out a greeting and waits for a command to be typed. Thus we have reached the end of the first part of the terminal script presented above. We reproduce it again below, indicating from what source the various messages come.

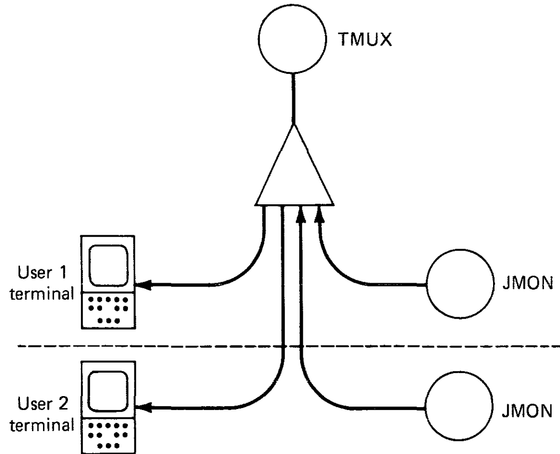

Figure 10-1 TMUX interposed between terminals and users

CMU CS Front End	Front End
Host: *c*	Front End
Welcome to Hydra/C.mmp	TMUX
Job Monitor	JMON
V5.09 6-Nov-1978 18:54:25	JMON
Type "Help" If you need it	JMON
@	JMON

10-1.2 Logging In

JMON understands a few commands and is principally responsible for getting the user logged in and talking to a command language. There are two reasons for interposing JMON at this point, rather than connecting the user directly to a command language:

Reliability. The command language process may stop because of hardware or software errors and it would be well to have a "backstop" process available which could recover from such a death.[1]

Decoupling abstractions. The notion of a "job" is (we believe) a different concept than "command language," and so we try to preserve this distinction in the implementation.

In fact, validating a user for the purpose of "logging in" is entrusted to still another subsystem procedure, *Authenticate.* When the "login" command is typed to JMON, *Authenticate* (for which JMON has a capability) is invoked

[1]Hydra cannot distinguish command language processes from any other and so takes no special action when the command language encounters an error.

and is passed a connection to the user's terminal. It is *Authenticate* which performs the familiar "name and password" dialog, and returns a special object of type USERTOKEN. (*Authenticate* is part of the USERTOKEN subsystem.) USERTOKEN objects contain (among other things) the user's official name and a capability for the user's private catalogue.

JMON can now start up a command language process for the user, using a capability for the standard command language procedure, CL. CL is passed the catalogue returned by *Authenticate* and is connected to the user's terminal by TMUX. JMON now becomes dormant; it is still waiting for commands, but TMUX is routing terminal traffic to the command language process. (See Figure 10-2.)

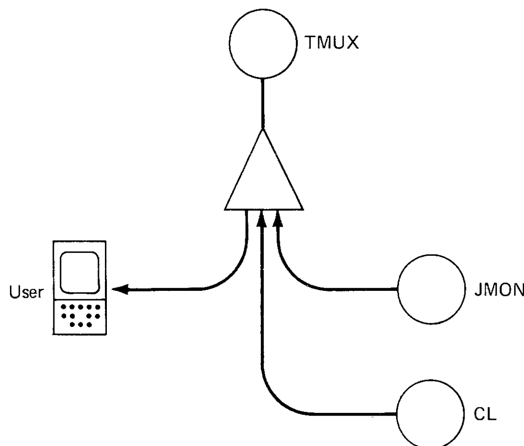

Figure 10-2 Processes associated with the user

Returning to the original script, we can see that what appeared to be a dialog with a single program actually involved two processes and three subsystems (not counting the PM that is scheduling all of this):

@*log*	
Job 21 Hydra/C.mmp TTY13 On 7 Aug 79 12:33:47	JMON
Name: *harbison*	Authenticate
Password:	Authenticate
Command Language	CL
V1.28.1 2-Aug-1978 10:53:55	CL
>	

10-1.3 The Command Language

Hydra's principal command language, CL, is quite interesting in itself.[2] It provides an interactive environment modeled after an Algol-like programming language, complete with iterative and conditional statements, macros, subroutines, and direct access to all the Hydra Kalls. At the same time, CL is effectively decoupled from many other facilities; it has no concept of "user" or "job," for instance. It is simply a procedure which takes two arguments: a terminal connection and a catalogue (which is interpreted to be the user's private catalogue).[3]

Some of the "flavor" of the command language can be seen in the portion of the user script involving the creation of a new process:

> *>capa &process*
> *>&process = $MakePMProcess(fortran, rootfinder)*
> *>$StartPMProcess(&process)*

The user is declaring a "capability variable," *&process*, and assigning to it the capability returned by the procedure *$MakePMProcess*. This returned capability, of type PMPROCESS (defined by the Policy Module), is then passed as an argument to *$StartPMProcess*, which starts the process.[4]

The CL is also an interesting example of the use of *TypeCall* to allow for many competing facilities. When CL sees a name which is not defined within itself, it assumes the name is an entry in the user's catalogue. To retrieve the associated capability, CL simply performs the "lookup" operation (a standard *TypeCall*) on the catalogue object. In fact, Hydra has two subsystems implementing two types of catalogues: CATALOGUE and DIRECTORY. The CL need not be aware of which object type it has because both subsystems define compatible *TypeCalls*.

10-1.4 Logging out

When the user logs off, he must first return to the JMON process, which understands such things. TMUX understands the break character to be a request to reconnect the terminal to the "previous" connection. Hence the sequence:

[2] Although CL is the standard, there are other such languages (e.g., see [Sno80]).

[3] We also note that the CL procedure has in its C-list, and hence all CL processes inherit, a capability for a shared "system catalogue" of useful capabilities.

[4] *$MakePMProcess* and *$StartPMProcess* are predefined CL macros which expand into *TypeCalls* on the (default) Policy Module. These operations are mapped by the Policy Module into the corresponding Kalls provided by KMPS to create and start processes. (See Chapter 12.)

>*Back to JMON, remember to Kjob	CL,JMON
@	JMON

The "kjob" command can now be typed to JMON, causing the CL process to be destroyed. Finally, JMON will inform TMUX of the termination, and TMUX will disconnect the terminal and destroy the user's JMON process:

@*kjob*	JMON
CMU CS Front End	Front End
Host:	Front End

10-1.5 Subsystem interactions

One of the major factors leading to the complexity of basic operating system software is the intricate web of dependencies often found among modules. With Hydra, however, the facilities described above are exceptionally well isolated. In almost all cases the interface consists of a single procedure call and no special privileges need be granted. Consider

TMUX. TMUX's only privilege consists of its being given capabilities for the terminal devices. It would be quite easy for several multiplexors to coexist: all that would be necessary is for them to be given disjoint subsets of the terminals. In fact, this is the way experimental versions of TMUX are debugged alongside the standard version.

JMON. TMUX's responsibility for higher-level software consists only of spawning a new process from a "canned" procedure. That the procedure is JMON is of no concern to TMUX. JMON's "privileges" consist of the (inherited) capabilities in its C-list.

CL. Again, this is only a canned procedure invoked by JMON. Different command languages could be associated with different users, perhaps via the USERTOKEN object returned by *Authenticate*.

We do not mean to suggest that there are no interactions in the above mechanisms. All the procedures have some knowledge of TMUX's operations because each must use TMUX to establish terminal connections. This means only that the procedures must acquire parameter templates for the TMUX object type which represents such connections, and that they must know what operations are available (via *TypeCall*) on those objects. This type of interaction between subsystems is common and easy to understand.

10-2 THE JOB SYSTEM

In discussing the functions of JMON, we glossed over another subsystem which is closely related: the Job System, which defines the abstraction of *job* in Hydra. This is another good example of how functionality is provided by user-level systems.

There are three important abstractions provided at the user level:

User is defined by *Authenticate* and provides a protected way of associating "user catalogues" with user names.

Terminal connection is defined by TMUX and provides the abstraction necessary to manage a hierarchy of connections between a person at a terminal and some collection of processes.

Job is defined by the Job System and provides the means to associate resource consumption with a logical task, such as a terminal session.

The motivation for the introduction of jobs into the system was the realization that there are a large number of housekeeping tasks which should be performed when a user "leaves." Files should be closed, devices deallocated, and in general all resources acquired by the user should be released. This is usually done during "logout" on other timesharing systems. Under Hydra, of course, the resources we wish to free are allocated by individual subsystems, and it is those subsystems which need to specify their own logout actions.

We therefore invented jobs so that we would have something to log out.[5] It works like this:

1. JMON invokes the Job System to create a new job (i.e., a new JOB object) when a user is accepted by *Authenticate.* A capability for the job is passed to CL and made available to the user.
2. Any subsystem that wants to be notified when the user logs off requires the user to pass a capability for his JOB object to one of the subsystem's procedures, such as *FileOpen* or *AllocateDevice.*
3. The subsystem can then invoke a Job System procedure, *JobInsert.* This procedure accepts a job and any other capability and enters the capability on the job's *kill list.* Typically, the capability entered on the kill list represents the resource to be reclaimed.
4. When the user logs out, JMON invokes the *LogOut* procedure on the JOB object. *LogOut* in turn will invoke the "destroy" *TypeCall* on every capability in the job's kill list. Subsystems implement this "destroy" operation for their own objects, and therefore they can perform arbitrary clean-up functions.

[5]Note that *login* is no problem. It is handled by *Authenticate* and involves a different set of issues.

This system works very well. For example, the File System puts its OPENFILE objects on the kill list as part of the *Open* procedures. The "destroy" *TypeCall* on such objects is just *Close*.

The Job System also allows users to create *subjobs* from a JOB object, allowing resource control to be structured hierarchically. Any procedure may consume arbitrary resources by spawning processes and invoking other subsystems, so passing a subjob to a procedure, and later logging out just that subjob, provides a means to recover those resources, even (especially) if the procedure is faulty and is unable to clean up after itself. (Subjobs are put on the kill list of their parent jobs. The "destroy" *TypeCall* on JOB objects is just *LogOut*, so the whole operation is recursive.)

10-3 RELIABILITY MECHANISMS

The subsystems which make up Hydra's operating system were contributed by many people with different programming styles and abilities. It was therefore always assumed that any procedure could be faulty and, in fact, might never return if called.[6] In some subsystems, such as the file and catalogue systems, this possibility was recognized, but ignored on the grounds that the users of those subsystems assumed the risk when they invoked the procedures. Some subsystems, however, like Job, *Authenticate*, TMUX, and CL, are critical because no one could access the system should they fail. For this reason they take extraordinary precautions when they invoke other subsystems.

As an example, consider the *LogOut* operation on jobs. The Job System must iterate through a kill list, invoking other subsystems. To guard against any of these invocations failing, the iteration is done in a separate process which is monitored by the Job system. Should any subsystem fail, the monitor will detect the stopped process and will start another process to complete *LogOut*.

Another need for explicit reliability mechanisms comes from the possibility that processes might be halted by Hydra due to hardware or software errors. Hydra cannot invoke any higher-level software to handle such cases beyond its normal interaction with the Policy Module. Therefore subsystems which use server processes must supply their own detection and recovery mechanisms. The Policy Module PM1 is a good example of typical mechanisms. Some of PM1's functions are implemented by demon processes which do *ReceivePolicy* operations and process KMPS' stop messages (see Section 12-2). There are several identical processes for two reasons: performance and reliability. By having several processes, PM1 achieves a high degree of parallelism since the processes need to synchronize only when accessing the

[6]It is for this reason that the Hydra kernel never invokes user-level software during any Kall except *Call* and *TypeCall*. The kernel supplies only type-independent operations.

shared scheduling tables. Also, should one of the processes encounter an error, another process would discover the corresponding stop message in the mailbox and could restart another process. Thus, each PM process constantly monitors all the others.

10-4 OTHER SUBSYSTEMS

We have discussed the most important subsystems in Hydra from the standpoint of providing operating system facilities: TMUX, JMON and the Job System, CL, Policy Module PM1, and Authenticate. Many other subsystems have arisen to cope with other practical problems encountered in a real operating system. Some of the major ones are listed here to round out this discussion.

10-4.1 Directory and Catalogue

These subsystems have been mentioned informally many times. Both of them implement the same fundamental abstraction; the directory subsystem was constructed first and was superseded by the improved catalogue subsystem some years later. A "directory" in Hydra essentially takes the place of a "file system" in most other systems. It provides "lookup," "enter," "create," and "delete" operations.

10-4.2 Device Allocation System

The Device Allocation System (DAS) manages the allocation of physical I/O devices. (The kernel enforces no policies at all with respect to DEVICE objects—for instance, it does not prevent several users from simultaneously connecting to, and using, the same device.) DAS provides a level of protection between users and the I/O system: it enforces mutual exclusion (when desired); it interfaces with the Job System to deallocate devices when a user logs off the system; it keeps publicly available lists of devices in use; and it provides special operations which can be used by an operator to forcibly disconnect a user from a device.

10-4.3 Fork

Hydra and the Policy Modules implement a very simple process creation mechanism. It is primitive in the sense that there is no way for a process to return a "value," nor is there a way to block waiting for a process to complete. The Fork System allows processes to be spawned in an environment which provides these functions.

10-4.4 Commands

COMMANDS objects are supported by the command language, CL; they are essentially "command language procedures." They can take arguments and inherit capabilities in much the same fashion as Hydra procedures, and are much easier to create—hence they are heavily used.

10-4.5 SYSMON

SYSMON is responsible for starting the user-level subsystems when Hydra is rebooted; it is the single procedure which is given control after the kernel has been initialized. It first starts the Policy Modules and then invokes initialization procedures for all the standard subsystems. These procedures may spawn processes (as is the case for several SUBFILE systems which spawn servers) or may simply initialize internal tables (as is the case with DAS). SYSMON normally operates automatically, but can receive explicit instructions from the operator's console.

10-5 RETROSPECTIVE

As we designed Hydra, our primary concern was *whether* we could build an operating system on top of it, rather than what that operating system would look like. It was therefore very satisfying when we saw an operating system develop which was not only functionally sufficient but better structured than many (we believe *most*) traditional ones. Unfortunately, most of this development was haphazard—our concentration on the kernel mechanisms caused us to pay very little attention to laying out this part of the system. Building an operating system as user-level programs does not necessarily reduce the amount of software necessary, and it should be planned with care.

One particular aspect of the user system that we never adequately addressed was the *environment* in which user programs could develop and operate. Hydra's protection mechanism is very good at ensuring that a procedure gets sharply defined privileges; unfortunately, many programs want to send messages to a "user" or print a message on the "console." Hydra works at a much lower level than the concepts of "user" or "console," and therefore every procedure must explicitly provide for appropriate capabilities to be passed to it, in addition to the parameters the procedure is really interested in. Many proposals for so-called "environment objects" (to be passed to all procedures by convention) have been offered, but each has been rejected because of its divergence from the Hydra philosophy and because we could not agree as to what should go in such objects. In the end, JOB objects assumed this role to a certain extent, but no acceptable general solution was ever designed. Most programs use a loose set of conventions supported by a standard User Library [Rei75, Gum78].

PART FOUR

THE SYSTEM IMPLEMENTATION

CHAPTER

ELEVEN

THE OBJECT STORE

The Hydra GST (for "Global Symbol Table") is the implementation of the object/capability abstraction on which all of Hydra rests. Virtually all the long-term data stored in the system is embodied in objects managed by the GST. For this single, universal storage mechanism to be usable, these objects must be managed efficiently and reliably. At the same time, the internal details required to achieve these goals should not be evident in the abstraction presented to users.

11-1 A VIRTUAL MEMORY SYSTEM

From the outside, the GST resembles a virtual memory system with a large, graph-structured address space. In fact, however, the GST subsystem proper implements only a large linear space of objects referenced by unique-names. It is actually the protection mechanism, with its knowledge of "paths," that provides higher-level structure.

Like other virtual memory systems, the GST uses several levels of the storage hierarchy (primary memory, drum, disk) to achieve an efficient implementation of the large address space. This is the rationale for dividing what appears to be a uniform structure into two parts:

Passive GST, which maintains objects and capabilities on secondary storage for long periods of time

Active GST, which maintains copies of objects and capabilities in primary memory for short periods while they are being referenced

Objects are referred to as being *active* or *passive*, depending on whether they are in the Active GST or not.

Initially, all objects are passive. Newly created objects are placed in the Active GST, and older objects are copied from the Passive GST to the Active GST if they are referenced. Active objects may be returned to the Passive GST if they cease being referenced, or they may be explicitly *updated*, i.e., copied back to the Passive GST in order to make the two GSTs consistent. Updating is important because at the time of a system crash only the Passive GST is preserved.

11-1.1 The Representation of Objects and Capabilities

The representation of objects differs in the Active and Passive GST, but in each case the representation is based on the division of an object into two parts:

Fixed-part, passive or active, which is a concise representative of the object
Representation, which is the union of the *C-list* and the *data-part*

The fixed-part is often the focus of the GST mechanisms because many references to objects can be satisfied by consulting only the fixed-part.

In the Passive GST, an object's (passive) fixed-part and representation are stored contiguously on a disk pack. In the Active GST, the object's (active) fixed-part is at a fixed address in primary memory; the representation may be in primary memory, but it may also remain passive if it is not needed. (See Figure 11-1.)

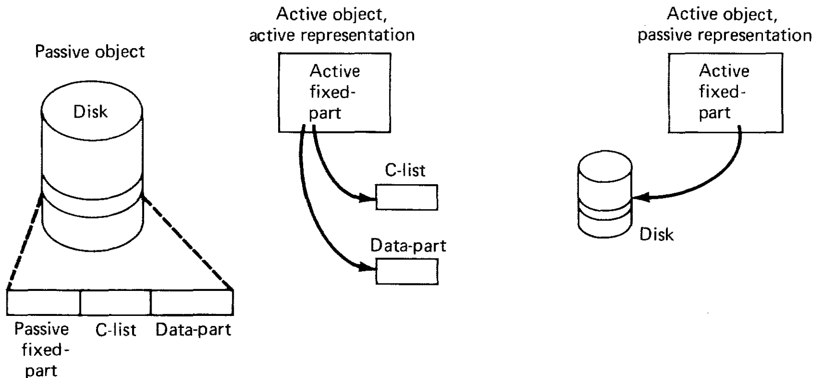

Figure 11-1 Various representations of objects

Table 11-1 lists the contents of the passive and active fixed-parts. Along with each field is a letter indicating whether the field is functionally necessary (F), is an optimization for efficiency (E), or is there to enhance reliability (R). Fields marked (FE) store functionally necessary information in an optimized format. The more important fields are described in Table 11-2.

The significance of these fields will become clearer as we discuss the mechanisms and policies for mapping capabilities to objects.

Capabilities also have passive and active forms. A passive capability contains only two things: the unique-name of the object it represents, and the set of kernel and auxiliary access rights. Passive capabilities may refer to objects in either the Passive or Active GST. In active capabilities, the object's unique-name is replaced by the memory address of the object's active fixed-part. Active capabilities thus refer only to objects in the Active GST.

Table 11-1 Composition of fixed-parts

Passive fixed-part		Active fixed-part	
Field name	Use	Field name	Use
UniqueName	F	UniqueName	F
CurVersion	F	CurVersion	F
PreVersion	R	PreVersion	R
TotRefCnt	F	TotRefCnt	F
TypeName	F	ActRefCnt	F
Flags	F	TypeIndex	FE
		PasDrm	E
		DrmRefCnt	R
		Checksum	R
		State	F
		CList	F
		DPart	F
		Semaphore	F
		TimeStamp	F

Table 11-2 Fields in the fixed-parts

Field name	Contents
UniqueName	The 64-bit name which uniquely names the objects
CurVersion	The disk address of the passive object
PreVersion	The disk address of the previous version of the passive object
TotRefCnt	The total reference count; the number of outstanding capabilities for the object
ActRefCnt	The active refence count; the number of outstanding references to the address of the active fixed-part
TypeName	The unique-name of the associated type object; i.e., the object's type
TypeIndex	An optimization of the type name (*TypeName*); an index into a table of addresses of the active fixed-part of TYPE objects
CList	The address of the C-list in primary memory
DPart	The address of the data-part in primary memory
Semaphore	A semaphore used to ensure mutual exclusion of access to the active fixed-part

Active capabilities are *active references* to objects, and as long as active references exist, an object may not be passivated.

In general, an active C-list may contain a mixture of active and passive capabilities, but when a C-list is passivated, all its capabilities are converted back to passive form. Thus the contents of the Passive GST does not depend on the state of the Active GST.

11-1.2 Mapping Capabilities to Objects

We now turn to the problem of dereferencing capabilities, a fundamental mapping operation in Hydra. Suppose we are given a capability and we wish to access the object to which it points. If the capability is in active form, we know that the object is also active, and in fact we have the address of the object's active fixed-part stored in the capability. The mapping is immediate. If the capability is in passive form, we have only the object's unique-name, and we do not know whether the object is active or passive. We must therefore first search the Active GST for the object, and then, should the search fail, proceed to the Passive GST, where we must find and activate the object.

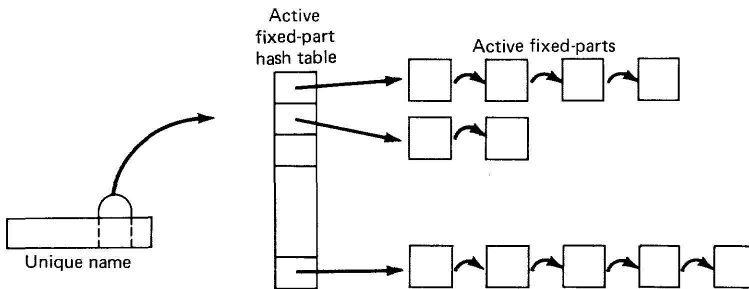

Figure 11-2 Mapping into the Active GST

All fixed-parts in the Active GST are stored in a 128-bin, chained overflow hash table. (See Figure 11-2.) Thus our first step is to hash the unique-name in the passive capability and determine in which bin the object should lie. We then search the list linearly for the active fixed-part. If it is found, we can immediately use the active fixed-part to change the capability to active form and increment the object's active reference count, and the mapping is complete.

If the object is not in the Active GST, we must find the object on disk. To avoid a linear search of the entire Passive GST, Hydra uses one of the fast paging disks to hold a copy of every passive fixed-part in the GST. We call this disk "the GST drum" to avoid confusion with the Passive GST disk. The drum is divided into 128 blocks of 256 fixed-parts,[1] and it is possible to

[1]The dimensions have been adjusted empirically over the years.

locate the block containing a specified object by hashing the object's unique-name. Given the proper drum block, we read it into a buffer and search it for the required object. (See Figure 11-3.) Once found, we can construct an active fixed-part from the passive one, set its active reference count to 1, link it onto the Active GST hash table, and change the original capability to active form. The object's representation is still on disk, and depending on the operation which originally caused the mapping, we may or may not bring it into core now.

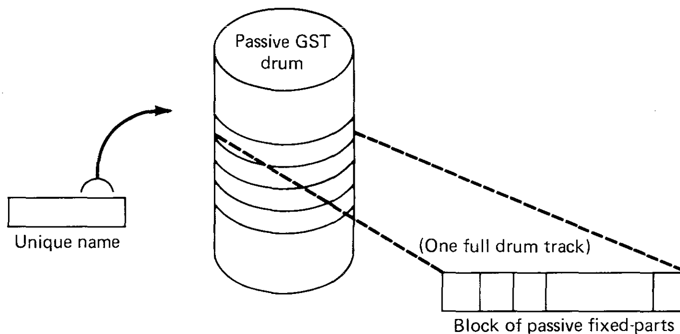

Figure 11-3 Mapping into the Passive GST

11-2 STORAGE MANAGEMENT IN THE GST

Efficient management of the Active and Passive GSTs is quite important for the proper functioning of Hydra. If objects remain in the Active GST too long, they consume a large amount of the available primary memory. Likewise, the elimination of garbage (i.e., unreachable objects) from the Passive GST is necessary lest the drum and disk fill up.

11-2.1 Active GST Maintenance

The Active GST uses a reference count scheme to determine when an active object may be *passivated*, i.e., sent back to the Passive GST. Each object's active fixed-part includes an active reference count which tallies the number of references to the active fixed-part's address. (Active capabilities are one source of such references; there are a few others.) If this count goes to zero, all remaining references must be by unique-name only, and hence the object may be returned to the Passive GST. The actual passivation process is handled by an asynchronous process called the *GST Demon*. This demon traverses the Active GST directories, looking for objects with no more active references. When it finds one, it either passivates it, or if the object's total

reference count is also 0, deletes it.[2] The GST demon independently looks for objects whose representation has not been accessed recently, regardless of the object's active reference count. (This is determinable from a time stamp held in the active fixed-part.) If the time of last access is greater than some threshold, the demon will passivate the representation of the object (leaving the fixed-part untouched). This can be done at almost any time because the only reference to the active representation is from the active fixed-part, and therefore synchronization is not difficult.

An interesting side effect of this passivation is the elimination of unreachable rings of objects from the Active GST. One would think that the active reference count of such objects would remain forever positive, but in fact when their representations become stale and are passivated, the capabilities in those representations are converted to passive form, thus decrementing the active reference counts in the referenced objects [Alm80].

11-2.2 Passive GST Maintenance

The situation with the Passive GST is more complicated. First, at the time of a system crash, all information in the Active GST is lost. This means that any reference counts maintained in only the Active GST will be lost, and it is practically impossible to keep the Passive and Active GSTs consistent at all times.[3] Second, the problem of circularity is ever present. Some form of garbage collection is necessary.

In fact, we have two Passive GST garbage collectors. The original one processes the GST off-line and eliminates unreachable objects. It is invoked primarily by system developers when they switch between the User and Experimental GSTs, since it has the side effect of rebuilding the passive fixed-part directory on the drum. (This so-called "cold start" of the GST is a uniprocessor algorithm which takes about 20 minutes for a 20,000-object GST.)

The more recent addition to Hydra is a multiprocess *parallel garbage collector* [Alm80], which runs automatically once a day and is able to clean the entire GST in parallel with other users. It has many pleasant characteristics, including insensitivity to system crashes. We will not discuss this

[2]The situation is bit more complicated than this. Certain object types, such as ports, are *never* passivated, because they reference dynamic storage areas outside the province of the GST. Other types, such as processes and LNSs, can be passivated but do not survive system crashes; capabilities for them are deleted upon reactivation after a crash. Users may cause object types they create to have this property also.

[3]Reference counts in the Active GST are safe. Hence when an object is created, we keep both an active reference count (ARC) and a total reference count (TRC). If the TRC goes to zero when the ARC does, we can delete the object rather than passivate it. Once an object has been passivated, the TRC is forever ignored and assumed to be infinite. This scheme works well under observed usage patterns in which most objects are never passivated anyway (see Chapter 16).

interesting algorithm here, except to note that in many ways the garbage collection problem in the GST is easier than similar problems in, for instance, LISP systems. In our environment, the objects to be collected are fairly large and all have integral synchronization mechanisms (semaphores).

11-3 MECHANISMS FOR RELIABILITY

In a highly interconnected structure, it is essential to limit the spread of damage caused by the appearance of incorrect data somewhere in the data base. The GST machinery is designed to detect inconsistencies as soon as possible and prevent the proliferation of incorrect data throughout the graph structure. The reliability mechanisms concentrate on detection of inconsistencies and restoration of erroneous structures to a consistent state. In this section we will examine the mechanisms used for error detection.

Let us first consider the representation of objects on disk. As noted earlier, the fixed-part, C-list, and data-part are written as a single contiguous unit. (The disk fixed-part is redundant anticipating the loss of the drum.) Whenever an object is returned to passive form, its three components are collected in a single buffer and written to disk, then read back and verified. Although costly in space and time resources, this approach practically guarantees a consistent version of the object on disk. The internal structure of a passive object has considerable redundancy as well. The fixed-part, data-part, and each capability have independent checksums. These ensure that a unique-name stored anywhere in the object cannot accidentally be changed to reference a different object. All interconnections on disk are represented as unique-names; no disk addresses are stored in passive objects. (The fixed-parts on the drum do contain disk addresses. These are constructed by scanning the disk and locating all objects after off-line garbage collection.)

In the Active GST structure, the data changes rapidly, and checksums are an inefficient error detection mechanism. Key status information is coded redundantly and legitimate values of all zeros or all ones are avoided.[4] Data that change infrequently or not at all (e.g., the unique-name) are checksummed and validated when the cost of doing so is tolerable. However, the most likely place for errors to creep in is in the inter-object pointers, which in the Active GST are addresses, not unique-names. To help detect incorrect pointers, an 8-bit "key" is computed from each object's unique-name and type. Wherever a pointer to the object is stored (e.g., in an active capability or fixed-part hash table link), and every time the pointer is dereferenced, the

[4]Formerly, a commonly observed hardware failure was the appearance of a word of all zeros or all ones without any error indication. Accordingly, such values, where possible, are treated as illegal. This applies even to logical single-bit fields, which are implemented as two-bit fields in which 00 and 11 are illegal. C.mmp's parity hardware also directly addresses this problem (see Chapter 2).

key is checked against the referent before the referent is manipulated. A mismatch signals an inconsistent data structure.

The reference counts stored in the fixed-parts of unpassivated objects are also potentially unreliable. A few validity checks can be used to test for reasonableness (e.g., the active reference count cannot exceed the total reference count), but in large measure the counts must be trusted. To minimize the ill effects of trusting an incorrect reference count, three techniques are used. First, during certain sensitive operations, reference counts are deliberately left too large. An error will then leave the count artificially high, but at least the object won't be deleted inadvertently. Second, when the total reference count becomes zero, the object is not immediately deleted, on the theory that if outstanding references actually remain, the object may be referenced before it is deleted and the reference count error detected. (The count is then set artificially high). Finally, when a fixed-part is deleted from primary memory, the key (discussed above) associated with the unique-name and type is altered in the released storage. This helps catch dangling references before they are used to access meaningless data.

One final mechanism is used to help ensure Passive GST integrity. When an object is written to disk, it never occupies the same physical space that its immediately preceding version did. Thus, in general, two copies of an object may simultaneously appear on disk: the "current" version and the immediately preceding one. Each version carries a time stamp, and the addresses of both versions appear in the fixed-part directory stored on the drum. In principle, then, if the current version is later discovered to be inconsistent, the "backup" version can be used instead.

11-4 RETROSPECTIVE

The GST is the foundation upon which all of the object-oriented structures of Hydra rest. It is heavily exercised and its weaknesses become easily visible. On the other hand, the GST has also received the most programming attention, and therefore criticisms about it tend to be more fundamental.

Probably the single greatest complaint about the GST is its performance. The implementation biases of the GST tend to view objects as comparable to files: of moderate size and relatively long-lived. The overheads in storage and processing time are not unreasonable in this light. As will be seen in Chapter 16, however, data on actual usage indicate that approximately five objects are created and deleted per user *per second*, and that over 98% of all objects are never passivated. (They are deleted first.) The file-like view of objects is therefore clearly inappropriate; a more realistic model would be procedure-activation records (frames) or LISP cells. The GST mechanisms are far too costly to support such usage efficiently, and substantial internal

redesign (certainly involving microcode support) would be required to repair this defect.

Despite its performance problems, the GST achieves its reliability goals. The error detection mechanisms are effective and increase execution overhead by no more than 10%. Their presence in the initial implementation significantly reduced debugging time by catching uninitialized pointers, synchronization errors, and the like. The existence of good detection facilities permitted the implementation of fairly extensive error recovery logic as well. However, after initial debugging, most of the observed errors were due to hardware failures, and rather than allowing recovery to proceed, we frequently chose to stop the system in its tracks and turn the machine over to the hardware engineers.[5] As a result, many of the recovery algorithms were never heavily exercised (and some were never implemented), and our experience with them is therefore more limited than we would like.

Finally, there are two general problems with a capability-based system such as Hydra for which we do not have solutions:

Accounting. No mechanism for accounting for GST resources exists, making it impossible to monitor or restrict usage. In the presence of object sharing, it is difficult to devise a fair strategy for such accounting. Is the creator responsible alone, or is responsibility shared in proportion to the rights possessed to an object?

Incremental backup. Most traditional file systems have the ability to back up logical portions of the system to guard against disk crashes and other massive failures. In Hydra, the concept of restoring a "portion" of the GST is complicated. What if a capability is restored for an object that no longer exists? How can a mechanism know how "deep" into the representation subgraph of an object to go when saving that object?

Although these problems can be tolerated in an experimental system, designers of a "production" operating system would have to address them.

In closing, we should perhaps recall that the GST, at the time of its design, was an ambitious, capability-based, virtual memory system. It has been operational for six years and serves as a stable base for Hydra, despite specific performance problems. We would doubtless implement it differently now (even on the same hardware), taking cognizance of the usage data we have acquired, but we are confident that the basic mechanisms for reliable storage of objects and capabilities are appropriate and sound.

[5]It is difficult to be sure that no unnoticed damage has been done to the GST structure in the presence of malfunctioning hardware, and therefore the safest course is to reboot the system. The probability is high that damage is confined to the Active GST, which will be discarded at the time of the reboot.

CHAPTER

TWELVE

SCHEDULING AND SYNCHRONIZATION

One of the principal motivations for Hydra's design was our desire to experiment with alternative operating system facilities. Scheduling policies are particularly rich fields for experimentation, especially in a multiprocessing environment. Consequently a great deal of attention was devoted to designing a scheme with which user-level programs could specify scheduling policies.

The final design is the result of trying to satisfy several potentially conflicting objectives. First, we wanted the maximum possible flexibility in specifying schedulers. Second, as with other user-defined facilities, we wanted to allow several schedulers to coexist; we felt that it should be possible, at one instant, for different processes to be under the control of different schedulers. Third, we wanted to ensure that an error in a scheduler would not result in a total collapse of the system. Finally, since we believed that some applications on C.mmp would be time critical, we did not want large software overheads associated with all scheduling decisions.

To satisfy these objectives, we divided the processor scheduling problem into two parts: *short-term scheduling* (at the level of a few milliseconds) and *medium-term scheduling* (significant fractions of a second and greater). The *Kernel Multiprogramming System (KMPS)* makes short-term decisions frequently and rapidly, leaving to the user-level schedulers, called *Policy Modules* (or *PM*s), the less frequent medium-term decisions. In fact, the Policy Modules have two responsibilities: they make absolute decisions about medium-term scheduling, and they influence short-term scheduling by supplying (to KMPS) a set of scheduling parameters for each process.

KMPS is responsible for multiplexing among a set of processes supplied by the PMs. The PMs can make medium-term decisions by incrementally modifying this set. Since KMPS multiplexes (only) the processes in the set, inserting a process into it effectively allows that process to execute; it is a medium-term decision to schedule that process. Conversely, removing a process from the set constitutes a decision not to schedule that process. KMPS makes its short-term decisions based on the per-process parameters supplied by the PMs, so although the PMs do not have absolute control of short-term policy, they can strongly influence it.

In this chapter we will be examining several facets of this scheduling

system. We will first look at the abstraction that KMPS presents to the Policy Modules, the way scheduling is parametrized and the way KMPS and the Policy Modules communicate. We will then consider how KMPS actually does its scheduling, and how this scheduling interacts with synchronization mechanisms used by the kernel and by user-level programs.

12-1 SCHEDULING PARAMETERS

As noted above, there are two aspects to the interface between KMPS and user-defined PMs:

1. The specification of the set of processes to be multiplexed by KMPS
2. The specification of the per-process parameters that influence KMPS's short-term scheduling decisions

We will consider the second of these first.

The set of short-term scheduling parameters was derived from a number of assumptions about the way that C.mmp would be used and, therefore, the kinds of policies that people would wish to implement with PMs. Among these, two are foremost:

1. We expected that C.mmp would be composed of a heterogeneous mix of processors. Some processors might be faster than others, only some might have floating point hardware, only some might have special microcode, etc. User programs, through their Policy Modules, must be able to specify their requirements for particular processors.
2. We had to allow Policy Modules to specify "important" expectations and limitations on the processes they were scheduling. PMs, for example, had to be able to limit the time their processes executed and the memory resources they consumed. In addition, PMs had to be able, in some cases, to assert the relative importance of the processes under their control.

On the basis of these considerations, we decided to model KMPS after a preemptive, time-sliced, priority-driven scheduler. Basically, KMPS treats the processors as an anonymous resource pool and tries to keep the collection of highest priority processes running at all times. At the end of a time slice, it services processes round-robin within a priority level to ensure fair service among processes of equal importance.

This quick description of KMPS correctly implies that some of the per-process parameters supplied by a PM are related to priority and time slice size; the remaining parameters express resource constraints and permit somewhat finer grained control of the short-term scheduling. The full set of parameters is listed in Table 12-1.

Table 12-1 KMPS scheduling parameters

Parameter	Meaning
Priority	The priority of the process; higher-priority processes preempt lower-priority processes.
TimeSliceLength	The maximum time the process may run before scheduling is reconsidered by KMPS. At the end of this time, KMPS will select another process of the same priority (if any) to run.
NumberOfSlices	The maximum number of time slices (of length *TimeSliceLength*) that may be consumed by the process before it must be returned to the Policy Module for possible reassignment of scheduling parameters. (This parameter may be specified to be "infinite.")
ProcessorMask	A bit mask that designates the set of processors on which the process is permitted to execute.
WorkingSetLimit	The maximum amount of primary memory that any LNS within the process may consume without consulting the Policy Module. This limit is expressed as a number of pages.
WaitTime	The time the process may remain in KMPS after it has blocked on a port or a Policy Semaphore. If this time is exceeded, the process is returned to the Policy Module.

A simple example will illustrate how a PM might use these parameters. A PM may specify a set of processes to KMPS and say, in effect,

> Allow each of these processes to run for (say) 30 sec—as thirty 1-sec time slices. Give them all equal priority. As each one completes its alloted 30 sec, give it back to me so that I can reassess the situation.

KMPS takes the processes (and those given to it by other PMs) and tries to let all of them make "fair" progress on their 30-sec allotments. Since all the processes have the same priority, they will be dynamically assigned to processors on a round-robin basis. Because of the nature of the multiplexing algorithm, each process will typically "move"—execute on several processors—while it is under control of KMPS.

The scheduling parameters are normally used for finer-grained control than is exhibited by this simple example. *Priority*, for example, is often used simply to indicate the relative importance of processes. It can, however, also be used to increase the effective parallelism in the system. By increasing the priority of I/O-bound processes, and correspondingly decreasing the priority of processor-bound ones, a PM can improve system throughput. If a process is I/O bound due to frequent terminal interactions, this improves response as well.

The specification of execution time as a combination of *TimeSliceLength*

and *NumberOfSlices* gives the PM some control over the "scheduling grain" of processes at the same priority level. By making *TimeSliceLength* relatively short, the PM will force frequent context switching between processes—thus ensuring similar rates of progress; this might be advantageous for interactive jobs, for example. The cost of this, of course, is additional scheduling and context switching overhead. Hence, for a collection of large, compute-bound jobs, the *TimeSliceLength* might be made large.

There are at least two reasons for using *ProcessorMask*. One is that some global policy has divided the processors into two or more groups and assigned processes to specific groups; this might be done, for example, to guarantee a certain level of service to particular processes. The other reason for using *ProcessorMask* is that the process needs a (hardware) facility not available on all processors—writable microstore for example.

WorkingSetLimit guards against a process consuming excessive primary memory. It gives the PM the opportunity to reconsider its scheduling decision(s) in the case that a process expands its memory requirements. Likewise, the use and utility of *WaitTime* is difficult to explain until after the paging mechanism has been discussed (Chapter 13). In effect, however, PMs use this parameter to avoid unnecessary, and time-consuming, paging when a process is blocked for only short periods. The impact of this parameter is graphically illustrated by the performance data in Chapter 16.

In order to use the scheduling parameters effectively, the PMs must have information about the total system load as well as information about the performance of the PM's own processes. As we shall discuss later, KMPS provides this information, which includes, for example, per-processor idle time, the amount of I/O activity, and the amount of primary memory available. By relating the characteristics of individual processes to those of the (dynamic) system load, the PMs can make intelligent medium-term scheduling decisions. They can, for example, balance the mix of processes to achieve high throughput.

Given all this mechanism, one might ask why a PM doesn't simply set the scheduling parameters cleverly, then give all its processes to KMPS and let KMPS do the scheduling. Most PMs won't do this simply because they will want to periodically review the scheduling parameter decisions. Even if a PM did want to do this, however, there is a problem—the management of primary memory—to be discussed at length in the next chapter. In brief, all processes in the set being multiplexed by KMPS must be (simultaneously) resident in primary memory. The size of memory, therefore, limits the size of this set. Generally there will be more processes that wish to run than can fit into the KMPS set, and hence the PM must stay involved.

12-2 PROCESS AND POLICY OBJECTS

In this section we will discuss the ways in which PMs communicate with the kernel about the the set of processes to be multiplexed. The kernel defines two object types for this purpose: PROCESS and POLICY.

A PROCESS object is simply the formal Hydra object that represents what we have been informally calling a "process." From a technical point of view, the C-list of a PROCESS holds capabilities for the LNSs which represent the stack of protection domains corresponding to the dynamic nesting of *Call* and *Return* in the process.[1] The data-part of the PROCESS holds the scheduling parameters and some of the current execution state. The PM always refers to processes via capabilities for PROCESS objects.

A POLICY object is *not*, as one might first suspect, a Policy Module. It is, rather, a communication vehicle between KMPS and the PMs—a mailbox through which the KMPS can notify a PM when something "interesting" happens to a process controlled by that PM. Each process has exactly one POLICY object associated with it, and that is the mailbox used when the kernel wishes to inform the PM about an event concerning that process.

PMs must be executable, of course, and so are, in general, processes. It is important to realize, however, that a "Policy Module process" is not distinguished in Hydra. Hydra does not know which (if any) processes implement PMs; it knows only how to communicate with something that is behaving like a PM—namely, by sending messages to a POLICY object.[2]

The fact that KMPS only knows about POLICY objects, and not about processes that implement PMs, provides a great deal of flexibility in the design of PMs. The simplest PM, for example, can be a single process servicing a single POLICY object. This is the way that our first PM, PM0, was built. Alternatively, the PM can be implemented as several processes all servicing the same POLICY object. This is the way that our second PM, PM1, was built, and it has two immediate advantages: it allows faster response, and it provides added reliability in the event that one of the processes fails.

Hydra defines several operations on PROCESS and POLICY objects. These operations are typically invoked by Policy Modules in response to requests by user programs.

[1] *Call* and *Return* affect the *protection* structure of a computation, but not the *process* structure. Hence the *Call* mechanism and the process mechanism are relatively independent.

[2] In principle, PMs can be controlled by other PMs, and so on, but we have not seen a genuine need for this feature. Generally, processes implementing the PM itself are given "infinite" scheduling parameters so that they never leave KMPS and so do not need a PM to schedule them. To satisfy Hydra, they are associated with their own POLICY object.

MakeProcess(D:slot(index), T:capa(index,PROCESS creation template), P:object(PROCEDURE,CallRts), procedure-parameters)

MakeProcess creates a new process whose initial protection domain is the LNS formed by merging the <Procedure-parameters> into procedure *P*. A capability for the new process is returned in slot *D*. The process does not begin to execute until a *Start* operation is applied to it (see below).[3]

MakeProcess is an asynchronous analog of *Call*; an LNS is formed in exactly the same manner as for a *Call*, but it is not immediately invoked. Instead a process is spawned (forked) and this LNS is the base domain of this process. Notice that, as in all object creation operations, a creation template is required. As an operational matter, only Policy Modules are given such templates. Therefore only PMs can create processes. No hierarchical structure (for protection or control) exists between the new process and the process executing *MakeProcess*; the PROCESS capability returned by *MakeProcess* may be shared in arbitrary ways.

AttachPolicy(Prcs:object(index,PROCESS,AttachRts), Pol:object(index,POLICY,AttachRts))

Associates the POLICY object, *Pol*, with the process *Prcs*. This POLICY object will subsequently be used as the mailbox through which KMPS communicates with the controlling PM.

AttachPolicy, which must follow *MakeProcess*, defines the Policy Module that will be responsible for scheduling the process. The scheduling parameters for the process may now be established by the Policy Module via the following Kall:

SetSchedParms(P:object(PROCESS,SetPCBRts), M:mem[n])

Copies the scheduling parameters (in a standard format) from the block of memory specified by *M* into the data-part of PROCESS *P*.

SetSchedParms actually sets all scheduling parameters except *ProcessorMask*. *ProcessorMask* can be set in every LNS; its initial value is inherited from the PROCEDURE object. This is done so that procedures may be coded to take advantage of special processor features without the caller needing to be aware of this. When a *Call* is executed, a new *ProcessorMask* is established, and if necessary, the process will be moved to another processor that satisfies this new mask. The same thing happens on *Return*.

After establishing the scheduling parameters for a process, a PM will typically ask KMPS to include it in the set of processes to be multiplexed:

[3]*MakeProcess* is a special case of the general *Create* Kall defined in Chapter 5. There are a number of "Make" Kalls for creating the kernel-defined types. These are the analogs of the type-specific creation operations that would be defined by user-level subsystems.

Start(P:capa(PROCESS,StartRts))

Gives control of the process to KMPS. *Start* returns as soon as KMPS verifies that the process will fit into primary memory and that there is at least one processor in the current hardware configuration acceptable to its *ProcessorMask*. The process remains under control of KMPS until it blocks or exceeds its scheduling parameters, at which time the Policy Module is informed via the attached POLICY object.

A PM may also choose to remove a process from the set it has given to KMPS:

Stop(P:capa(PROCESS,StopRts))

Requests that process *P* be stopped and removed from KMPS's control. The process is not necessarily stopped immediately, but when it is, KMPS notifies the Policy Module via the POLICY object.

Note that processes may not be actually stopped at the time that the *Stop* Kall completes. In particular, KMPS will not stop a process that is inside a critical region in the kernel. To do so would both degrade performance and lead to potential deadlocks. Also, note that a process may stop for other reasons than an explicit *Stop* Kall, namely, becoming blocked or exceeding its scheduling parameters. Indeed, these other reasons are the most common ones. A PM generally uses *Stop* only on behalf of a user request or to implement a strongly preemptive policy.

Whenever KMPS stops a process given to it by a Policy Module, either because of a *Stop* operation or something else, it places a *stop message* in the POLICY object attached to the process. This message includes the identification of the process, the reason it was stopped, the amount of processor and memory resources used by the process, and some information about the total system load.[4] The Policy Module receives these stop messages with the *ReceivePolicy* operation:

ReceivePolicy(M:mem[16], Pol:object(index,POLICY,ReceivePolRts))

Retrieves the first message from *Pol* and writes the information into the block of memory specified by *M*.

When a process is stopped, the PM must decide what to do. There are two common cases. First, the process may have stopped because it blocked, say, attempting to receive a message from a PORT. The PM must record this fact and wait to restart this process until the kernel notifies it that an

[4]This load information is inserted in the stop messages for the convenience of the PM. It consists, for example, of information about the number of free page frames in primary memory and the amounts of processor and I/O time used. The PMs use this information to alter their global scheduling strategy.

appropriate message has arrived at the port.[5] Since a process has been stopped, there is probably some primary memory available now, so the PM may wish to start other processes. It will use the information about the total system load to make this decision. Second, the process may have stopped because it exceeded its scheduling parameters, i.e., it consumed the resources previously allocated to it by the PM. In this case the PM must decide whether to restart this process or some other(s). Again, it will probably consult the information about system load to make this decision. In either case, it must decide on the per-process scheduling parameters to use for each of the processes that it starts. It will probably use a combination of the total load characteristics and those of the individual processes to make this decision.

Up to this point we have been concerned with those Kalls that are typically used by Policy Modules. Generally speaking, user processes do not invoke KMPS Kalls directly, but there is one exception—the *RunTime* Kall. This Kall is used by a process to negotiate scheduling requirements with its PM.

RunTime(T:integer)

Requests the attached PM to not interrupt this process for *T*/60 seconds.

Normally, a process has no influence on its PM's scheduling decisions; it must trust its PM to allocate reasonable resources to it. There are, however, occasions when this is unsatisfactory. Suppose, for example, that a process needs to perform a short series of operations that must be atomic (indivisible). It could use the normal mutual exclusion semaphores to define a critical section, but the cost may be excessively high compared to the size of the critical section. The process could use inexpensive spin locks *if* it could be assured that, once within the critical section, it will not be removed by its PM from KMPS' control. (If it were removed, the spin lock would prevent other processes from entering the critical section, but these other processes will endlessly test the lock and thus appear active (compute-bound) to the PM. In this way, they will compete for the PM's attention, which should be focused on the process within the critical section.) This is the purpose of *RunTime*.

Using the *RunTime* Kall, the process requests the kernel to *guarantee* that it will not be seized by its PM for a specified interval. KMPS, which implements *RunTime*, checks to see if the resources already allocated by the PM are sufficient to satisfy the process' request. If so, KMPS returns control to the process, indicating that its request has been satisfied. The process can now test the spin lock with the knowledge that should its PM now attempt to

[5]Only the performance, not the correctness, of the port operations depend on the PM's cooperation here. In particular, if the PM prematurely restarts the waiting process, KMPS will simply stop it again.

stop the process before it has consumed the guaranteed time interval, KMPS will reject the stop request. On the other hand, if the process does not have sufficient resources allocated to it, KMPS stops the process and returns it to the PM, indicating that the process has issued a *RunTime* request. The PM now has three choices:

1. It can allocate the necessary resources to the process and restart it, in which case the *RunTime* reports success.
2. It can fail to allocate sufficient resources and restart the process, in which case the *RunTime* reports failure.
3. It can simply fail to restart the process, in which case the process remains blocked indefinitely.

Note that no matter what action the PM takes, the integrity of the critical section is preserved. KMPS has acted as an intermediary in the negotiation between the process and its PM. Once the negotiation is concluded, KMPS enforces the agreement reached. The *RunTime* Kall provides a good example of the separation of policy and mechanism, since the processor resource allocation policy is established outside the kernel (by the PM), while the kernel supplies the mechanism by which that policy is implemented.

12-3 SYNCHRONIZATION MECHANISMS

Scheduling and synchronization are clearly related. Having examined the facilities KMPS provides for scheduling processes, we now turn our attention to the mechanisms it supplies for synchronizing them.

Mechanisms for coordinating asynchronous activities are required by users and kernel alike. There is no master-slave relation among the processors. Whenever a user executes a Kall, for example, the code of that Kall begins to execute immediately on the same processor that was running the user process. Since all processors can access all the shared memory, accessibility to the kernel's data is not a problem. On the other hand, the synchronization of updates to this data, and the corresponding software contention, *are* problems.

To address these problems, we chose to synchronize on data rather than code, and to use fine-grain synchronization. Two processors, or processes, are free to execute the same code simultaneously as long as they are operating on distinct data structures. There are, for example, many queues in the system, but only one set of queue manipulation routines. Each queue is separately protected by one of the synchronization mechanisms discussed below. The queue management routines lock a particular queue before accessing it. Thus many queue manipulations may be proceeding in parallel as long as they are acting on distinct queues.

By "fine-grain" synchronization we simply mean that the data structures

involved are generally small. Equivalently, we mean that there are a large number of independently locked structures—literally thousands of them in Hydra. Having many small structures decreases the probability of contention for any one of them, but at the expense of more frequent execution of the synchronization primitives. Thus keeping this cost low was a major design problem.

To span the spectrum of needs for synchronization, Hydra provides three different synchronization mechanisms:

Locks, low-level mechanisms designed for very fast, but short, *processor synchronization*. A processor that blocks on a lock remains physically idle until unblocked by another processor.

Kernel Semaphores, intermediate-level mechanisms designed to provide *process synchronization*. When a process blocks on a Kernel Semaphore, it is removed from its processor and held within KMPS until it is unblocked. It is not swapped out of primary memory.

Policy Semaphores, used to provide longer-term synchronization for users. A process blocked on a Policy Semaphore is not only removed from its processor, but is sent back to its Policy Module, leaving KMPS and (possibly) primary memory.[6]

The reason for the three levels of synchronization should be obvious: each is more expensive than its predecessor in execution time but less expensive in terms of the resources tied up by a blocked process. Locks are very rapid, but may disable a processor (even from servicing interrupts) for some time. Kernel Semaphores are more expensive, especially when they block and force a context swap. Blocked processes do not consume processor resources, but they do consume memory since the process remains in KMPS. Policy Semaphores are the most expensive since blocking may imply several paging operations, but blocked processes consume almost no resources.

Much of the success of Hydra's multiprocessing properties are the result of the use of these three levels of synchronization primitives. We will examine their performance in more detail in Chapter 16. Here we shall just say a a bit more about their implementation and resulting characteristics.

Locks A lock is a small (two-word) data structure that acts as a fast mutual exclusion semaphore. There are two operations defined on a lock, *Lock* and *Unlock*. If a process blocks on a lock, the processor running the process sets a bit in the lock structure to indicate that it is waiting for the lock, disables all

[6]From KMPS' standpoint, blocking in a *ReceiveMsg* Kall in the Message System is equivalent to blocking on a Policy Semaphore, so this mechanism handles all user-level interprocess and I/O synchronization.

I/O and scheduling interrupts, and executes a "WAIT" instruction.[7] If the *Unlock* operation (on another processor) finds that processors are waiting on the lock, it will send an "unlock interrupt" to all such processors. The interrupted processors then loop to try for the lock again; one of them will get it and the others will return to the idle state.[8]

Purists will recognize the potential for "individual starvation" in this scheme. Given enough contention for a single lock, it is theoretically possible for one of the processors to remain blocked forever, i.e., to always lose the race. The likelihood of long-term starvation is infinitesimal, however, and we chose to permit its possibility rather than unnecessarily complicate the mechanism.

Kernel Semaphores Kernel Semaphores are four-word data structures consisting of a *process queue header*, a *count*, and a *lock* (as described above). The principal operations on Kernel Semaphores are *P* and *V*, which are essentially identical to those originally defined by Dijkstra [Dij68].

When a *P* operation on a Kernel Semaphore blocks, KMPS places the process on the semaphore's process queue, selects another process to run on this processor, and performs a context swap to the new process. (Note that no other processor is involved in this operation.)

When a *V* operation notices that some process is blocked on the semaphore, it removes the process from the semaphore's queue, enters it in the set of feasible processes. It then selects a processor that is executing a lower priority process. This processor is interrupted to cause it to reconsider its own scheduling in light of the existence of a new feasible process.

A more detailed description of the implementation of Kernel Semaphores is given in the following section.

Policy Semaphores In contrast to locks and Kernel Semaphores, Policy Semaphores are a true object type. Although implemented by the kernel, they behave as any other object. Users can create objects of this type, exchange capabilities for them, and so on. The abstract semantics of the operations, *P* and *V*, are just like those for the corresponding operations on Kernel Semaphores. The difference between Kernel and Policy Semaphores centers on what happens when a process blocks.

When a process blocks on a Kernel Semaphore it remains in KMPS, on the semaphore's queue. When a process blocks on a Policy Semaphore, on

[7]The WAIT instruction is a nice feature of the PDP-11; it causes the processor to cease executing instructions until an interrupt restarts it. Because of WAIT, Hydra's locks do not cause memory references (and hence contention). Logically, however, a "jump-to-self" would work just as well.

[8]The "unlock interrupt" is just the priority-7 interprocessor interrupt. All other interrupts have lower priority than this, so by setting its priority to 6, a blocked processor can ignore everything except the unlock operation.

the other hand, it will generally be returned to the controlling PM—and that usually implies that the process will be swapped out of primary memory. A process is not *necessarily* returned to its PM when it blocks, however. When it blocks it is first put on a special queue, the *wait queue*, in KMPS. It stays there for a limited time specified by the *WaitTime* scheduling parameter. If the process should be unblocked before *WaitTime* expires, it is returned to the execution mix just as is done with Kernel Semaphores. If the *WaitTime* expires before the process is unblocked, the process is stopped and returned to the PM. At that time, the process is also eligible to be swapped out of primary memory.

The choice of whether to use locks, Kernel Semaphores, or Policy Semaphores (with or without *WaitTime*) is based on both the probability of being blocked and the expected duration of the blocked period. In Chapter 16 we give data on the choices made in the kernel and some applications—and their impact on overall performance.

12-4 IMPLEMENTATION

In this section we will describe some aspects of the implementation of KMPS, namely, the scheduling mechanism and Kernel Semaphores. Although we have generally avoided low-level implementation descriptions in this book, there are two reasons for presenting this material here. First, some people may believe that scheduling and synchronization in a distributed operating system are necessarily complex; this example should help debunk that view. The implementations we shall show are among the most sensitive to the distributed nature of Hydra, and they are not especially complex. Second, and more importantly, this example will serve to illustrate the internal organization and programming methodology used throughout the kernel.

In Chapter 3 we briefly described the notion of *data abstraction* and used it to motivate the type-extensible, object-oriented "virtual machine" provided by Hydra. The internal implementation of Hydra, however, also heavily uses the data abstraction philosophy—the implementation of KMPS provides a good illustration of this.

Before beginning we need to say a few words about the language in which the implementation is described. Hydra is actually implemented in Bliss/11, an untyped "systems implementation" language. Bliss does not directly support the data abstraction paradigm; instead the paradigm is enforced only by programming convention. This works well enough in practice, but it does not make Bliss a suitable "publication language." Unfortunately, there is no other widely known language that is quite suitable either. Therefore, as in Section 5-5, we have used a Pascal-like notation with a few additions and modifications. We expect that our readers will be sufficiently familiar with modern languages that the programs themselves, together with

some explanation, will be a adequate definition of the notation. (It should be noted that the term "procedure" as used in this section does not refer to "Hydra procedures;" similarly, "process" and "semaphore" are being redefined in a language context.)

Conceptually, KMPS is composed of six abstractions: queues, locks, processes, processors, semaphores, and the scheduling mechanisms. These abstractions are defined by a set of modules:

```
module QueueModule is
    type QueueKind is
            enumeration(FIFO, Priority, ReversePriority);
    type Queue (qt:QueueKind) is private;
    procedure Enqueue (q:Queue, ps:Process);
    procedure Dequeue (q:Queue, pri:integer, mask:ProcessorSet)
            returns Process;
    implementation
        . . .
    end module;
```

```
module LockModule is
    type LockType is private;
    procedure Lock(l:LockType);
    procedure Unlock(l:LockType);
    implementation
        . . .
    end module
```

```
module ProcessModule is
    type Process is private;
    procedure ContextSwap(ps:Process);
    procedure Priority (ps:Process) returns integer;
    procedure ProcessorMask(ps:Process) returns ProcessorSet;
    procedure TimeSliceEnd(ps:Process) returns boolean;
    procedure WhichProcessor (ps:Process) returns Processor;
    var processors: Queue(ReversePriority);
    implementation
        . . .
    end module;
```

```
module ProcessorModule is
    type Processor is private;
    type ProcessorSet is private;
    procedure AnyProcessor returns ProcessorSet;
    procedure SchedulingInterrupt(pr:Processor);
    procedure Me returns Processor;
    procedure MeMask returns ProcessorSet;
    procedure Running(pr:Processor) returns Process;
    procedure Blind;
    procedure UnBlind;
    implementation
        . . .
    end module;

module SemaphoreModule is
    type Semaphore is private;
    procedure P(s:Semaphore);
    procedure V(s:Semaphore);
    implementation
        . . .
    end module;

module SchedulerModule is
    procedure FindProcessor(ps:Process) returns Processor;
    procedure FindProcess(pri:integer) returns Process;
    implementation
        . . .
    end module;
```

These module definitions illustrate our major departure from Pascal-like languages. The type definitions, variables, and procedure headers are visible to users of the modules; the implementations, denoted by ellipses here, are not.

We will describe the implementations of only the last two of these modules—*SemaphoreModule* and *SchedulerModule*. We will informally describe the semantics of the other abstractions, however, and that will help to define the notation as well.

Let's begin with *QueueModule*. The main abstraction defined by this module is the familiar notion of a queue, but it has been specialized to queues of processes and to the kinds of queuing operations we want to do. Note that we simultaneously define several kinds of queues: "FIFO," "Priority," and "ReversePriority." The phrase **is private** attached to the declaration of type *Queue* simply means that the actual representation of the type will appear in the implementation-part of the module, and hence it is not visible to the user. Since the type declaration is visible, however, the user

may declare variables of type *Queue*, e.g.,

var *feasible: Queue(Priority)*

The parameter in this declaration is important; it says that this particular queue is priority ordered.

Since type *Queue* is specified to be "private," there isn't much that one can do with the variables of the type except pass them to the procedures declared in the module header. Hence, if *P1* is a variable of type *process*, we can write:

Enqueue (feasible, P1)

Although we have not provided a formal specification of its semantics, the clear implication is that the process represented by *P1* will be enqueued on the feasible queue; since we know that that queue is priority ordered, *P1* will be inserted into its proper priority position.

The *Dequeue* operation is somewhat more interesting; here we can see the influence of the kinds of queuing operations we want to perform. Note that it has three parameters: a *queue*, an *integer*, and a *ProcessorSet*. It will dequeue the first element of the specified queue that (a) has a priority greater than or equal to the integer value and (b) whose *ProcessorMask* value has a non-empty intersection with the specified *ProcessorSet* parameter. If, for example, *PSM* is a variable of type *ProcessorSet* whose value includes processors 1, 3, and 5, the call

P1 := Dequeue(feasible, 0, PSM)

will dequeue the first process on the feasible queue with a priority of at least zero and that can run on one of these processors. Since 0 is the minimum priority defined by Hydra, and since the feasible queue is priority ordered, this call will actually return the highest priority process that is capable of running on one of these three processors.

We have already informally described locks; *LockModule* is simply the definition of them. There is nothing exotic in its definition. Because of the type declaration one can declare variables of type *Lock*. Because of the procedure declarations, one can invoke (only) *Lock* and *Unlock* operations on these variables.

The other two modules for which we shall not give implementations are those that define types *process* and *processor*. The type *process* is, of course, merely the abstraction of the intuitive notion of a process. Its representation contains the process state, including the scheduling parameters described previously. The procedures *Priority* and *ProcessorMask* retrieve the values of the two scheduling parameters we need here. The operation *TimeSliceEnd* determines whether the specified processor has exceeded its current time

slice parameter. *ContextSwap* switches state between the currently executing process and that of its parameter (it is effectively a coroutine invocation). The operation *WhichProcessor* allows us to determine which processor, if any, the process is currently executing on.

Note that *ProcessModule* maintains a queue variable, *processors*, that is visible. This queue is simply the list of processes that are currently executing. It is maintained (in *ReversePriority* order) by ContextSwap. As we shall see, it will be necessary sometimes to know the identity of the lowest-priority process that is currently executing—and this queue lets us determine that quickly.

ProcessorModule defines two types. Type *Processor* is the abstraction of a hardware processor. The type *ProcessorSet* is the abstraction of a set of processors and is used to encode the *ProcessorMask* scheduling parameter discussed previously. The variable *AnyProcessor* is simply the *ProcessorSet* that contains all processors that are currently configured into the system. The *SchedulingInterrupt* operation allows one to cause an interrupt on the specified processor.[9] The operation *Me* returns the processor on which the operation is executed. The operation *MeMask* returns a *ProcessorSet* containing just *Me*. The operation *Running* returns the process that is executing on the designated processor. Thus "*Running(Me)*" returns the process executing on the processor on which this operation is executed. The operations *Blind* and *UnBlind*, respectively, disable and (re)enable all interrupts on the current processor.

Now let's turn to the implementation of the two modules of most interest—*SemaphoreModule* and *SchedulerModule*. First, *SemaphoreModule* is implemented as follows:

```
module SemaphoreModule is
    type Semaphore is private;
    procedure P(s:Semaphore);
    procedure V(s:Semaphore);
    implementation
```

[9]The *SchedulingInterrupt* operation is actually implemented as a priority-four interprocessor interrupt. See Chapter 2.

```
type Semaphore is
    record
        l: LockType,
        count: integer:=1,
        q: Queue(FIFO)
    end record;

procedure P(s:Semaphore) is
    var ps:process;
    begin
        Blind;
        Lock(s.l);
        s.count := s.count-1;
        if s.count < 0
          then
            Enqueue(s.q, Running(Me));
            ps:= FindProcess(0);
          else ps:= null;
        end if;
        Unlock(s.l);
        if ps ≠ null then ContextSwap(ps);
        UnBlind;
    end procedure;

procedure V(s:Semaphore) is
    var ps: Process;
    begin
        Blind;
        Lock(s.l);
        s.count:= s.count + 1;
        if s.count ≤ 0
          then ps:= Dequeue(s.q, 0, AnyProcessor);
          else ps:= null;
        end if;
        Unlock(s.l);
        if ps ≠ null then FindProcessor(ps);
        UnBlind;
    end procedure;
end module;
```

The implementation of these routines is similar to that found in other systems that use semaphores. Both *P* and *V* first make themselves "blind" to interrupts to ensure that they are indivisible. They then lock the particular semaphore on which they are operating; this permits other *P* and *V* operations to proceed asynchronously on other semaphores, but ensures mutual

exclusion with respect to this particular one. The difference between disabling interrupts (*Blind*) and the *Lock* operation is, of course, one of the differences between uniprocessor and multiprocessor implementations of semaphores. On a uniprocessor only *Blind* would have been needed.

The body of *P* is relatively simple. It decrements the semaphore's counter and then determines whether the result is negative, indicating that the process must be blocked. In the event that the process must be blocked, it first enqueues the current process on the (FIFO) queue associated with the semaphore and then determines the next process to run by calling *FindProcess*. *FindProcess* will return the highest priority process that can run on the current processor. (We'll explain the parameter to *FindProcess* shortly.) The *P* implementation then invokes *ContextSwap* to transfer control to this new process. The only unusual thing about this implementation is that the invocation of *ContextSwap* is moved outside the critical region in order to increase the potential for parallelism. A uniprocessor implementation would probably have invoked

ContextSwap(FindProcess(0))

directly in the **then**-branch of the **if** statement and would not have bothered with the variable *ps*.

The body of *V* is similarly straightforward. The *count* field is incremented and tested against zero. If it is non-positive there is a process on the semaphore's queue that should be awakened. It is dequeued and passed to *FindProcessor* to find a suitable processor to execute it; if no suitable processor is available, *FindProcessor* will simply insert it on the list of feasible processes. The use of *FindProcessor* is the major departure from uniprocessor implementations; on a uniprocessor one would determine only whether the priority of the new process exceeds that of the currently executing one, and if so, perform a context swap to it. On a multiprocessor, however, there are several processes executing simultaneously. Not only must we determine whether any of them has lower priority than the new process, but we would like to preempt the one with the lowest possible priority. We'll see how this happens below. Finally, note that, like the call on *ContextSwap* in *P*, the scheduling operation (*FindProcessor*) is moved outside of the critical region to increase potential parallelism.

Finally, let's consider the implementation of *SchedulerModule*; this module actually defines the (parametrized) scheduling policy of KMPS.

```
module SchedulerModule is
    procedure FindProcessor(ps: Process);
    procedure FindProcess(pri: integer) returns Process;
    implementation

    var feasible: Queue(Priority);

    procedure FindProcessor(ps: Process) is
        begin
        var newps: Process;
        Enqueue(feasible, ps);
        newps := Dequeue(processors,Priority(ps),ProcessorMask(ps));
        if newps ≠ null
            then SchedulingInterrupt(WhichProcessor(newps))
        end if
        end procedure;

    procedure FindProcess(pri: integer) is
        begin
        return Dequeue( feasible, pri, MeMask);
        end procedure;

    interrupt procedure ReconsiderScheduling is
        var ps: Process;
        ps := FindProcess( Priority(Running(Me)))
        if ps ≠ null
            then FindProcessor(Running(Me)); ContextSwap(ps)
        end if;
        end procedure;

    interrupt procedure TimerInterrupt is
        begin
        if TimeSliceEnd(Running(Me))
            then SchedulerInterrupt(Me)
        end if;
        end procedure;

    end module
```

The implementation of *FindProcess* is trivial; it simply dequeues a process from the feasible queue. When called from outside the module (notably from *P*), the parameter to *FindProcess*, *pri*, is always zero, and hence *Dequeue* will simply find the highest priority process that is eligible to run on the current processor.

The implementation of *FindProcessor* is somewhat more interesting. It first inserts the parameter process onto the feasible queue. It then attempts to dequeue a process from the processors queue; recall that this queue contains the processes that are running and that it is in *reverse* priority order. Therefore, if one exists, the dequeued process will be the *lowest* priority process currently running on some processor that is also capable of running the parameter process, *ps*. *FindProcessor* sends a scheduling interrupt to this processor; as we'll see in a moment, the effect of this will (usually) be to schedule the process on the interrupted processor.

Note that there are two internal ("hidden") procedures in *SchedulerModule*; both are "interrupt procedures." One is invoked in response to the scheduling interrupt, and the other is invoked in response to the interval timer interrupt. The timer interrupt handler should be self-evident; it merely checks to see whether the current process has exceeded its time slice[10] and, if so, forces scheduling to be reconsidered by sending an interrupt to its own processor.

We can consider the scheduling interrupt handler, *ReconsiderScheduling*, in more detail. Recall that *FindProcessor* causes a scheduling interrupt on a processor that it believes is executing a process of lower priority than a newly eligible one. Therefore, *ReconsiderScheduling* simply attempts to find a process on the feasible queue with a higher priority than the one that is currently executing. It will probably find the one placed there by *FindProcessor*, although it is also possible that another processor has either already removed that process or inserted one of even higher priority. In any case, if *ReconsiderScheduling* finds a higher-priority process, it first attempts to find another processor to execute the currently executing process and then performs a context swap to the new process. As we shall illustrate below, the interaction of *FindProcessor* and *ReconsiderScheduling* can, in principle, cause a rippling effect in which nearly all processors reconsider their scheduling.

Let's consider an example that illustrates the interaction of these routines in the scheduler. Figure 12-1 shows an initial configuration in which there are three processors (Nos. 0, 1, and 2), three running processes (*A*, *B*, and *C*) on the processors, and a fourth process (*D*) blocked on a Kernel Semaphore. The priority of the processes are 50, 14, 6, and 20, respectively. The *ProcessorMask* values for processes *A*, *B*, and *C* indicate that they can execute anywhere, while *D* can execute only on processor 1.

Assume process *A*, executing on processor 0, *V*'s the semaphore, causing

[10]We have ignored the bookkeeping necessary to keep track of the scheduling parameters and the mechanisms that stop processes and return them to their PMs.

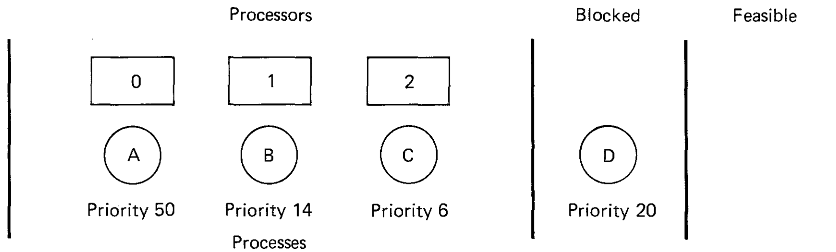

Figure 12-1 Execution state prior to A unblocking D

process *D* to become unblocked. As part of the code for the *V* operation, processor 0 will execute

FindProcessor(D)

to find a processor on which to run *D*. *FindProcessor* discovers that *D* can run only on processor 1. Processor 1 is currently running process *B* at a lower priority, so processor 0 enqueues *D* on the feasible queue and sends a scheduling interrupt to processor 1. Processor 0 has now completed its scheduling responsibilities associated with *V*'ing the semaphore, and returns to process *A*.

Processor 1 receives the scheduling interrupt and begins to execute *ReconsiderScheduling*. Because it is currently executing a process of priority 14 (*B*), *ReconsiderScheduling* executes

FindProcess(15)

to determine if there is a feasible process of greater priority than *B*. There is, namely, the newly unblocked process *D* at priority 20. Now *ReconsiderScheduling* must find a processor to run *B*, and therefore executes

FindProcessor(B)

B's mask indicates it can run on any processor, and since processor 2 is running at the lowest priority, *FindProcessor* will put *B* on the feasible queue and send a scheduling interrupt to processor 2. Processor 1 has passed the scheduling burden to processor 2 and now begins running process *D*.

ReconsiderScheduling now executes on processor 2. Since it is running process *C* at priority 6, *ReconsiderScheduling* calls

FindProcess(7)

which returns process *B*. Now, rescheduling of *C* must be attempted, so processor 2 invokes

FindProcessor(C)

There is no processor running at a lower priority than *C*, so *FindProcessor* just leaves *C* on the feasible queue. Processor 2 now begins to run process *B*. Figure 12-2 illustrates the state after these operations.

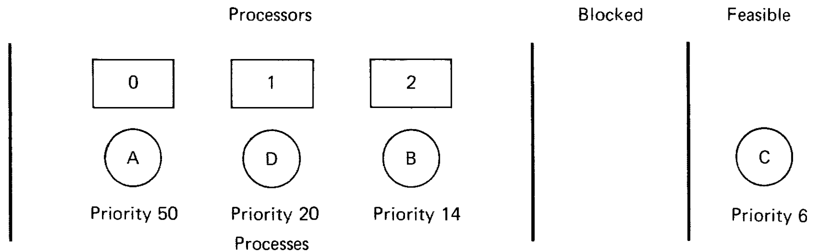

Figure 12-2 Execution state after A unblocks D

12-5 RETROSPECTIVE

KMPS was the first operational component of the Hydra kernel, and it has changed remarkably little over the lifetime of the system. This correctly suggests that the basic distributed scheduling mechanism is appropriate to the task of utilizing the inherent parallelism of C.mmp. KMPS has exhibited almost none of the usual chronic problems of highly parallel programs: deadlocks and processor starvation. The two times in the history of Hydra that bugs of this character were unearthed in KMPS, they were associated with the addition of new (and not quite correctly implemented) facilities.

We feel strongly that the data abstraction approach to coding KMPS and other parts of the kernel is, in large measure, responsible for the fact that it has had few errors and has been eminently maintainable. If we were to implement the system again we would like to use a language that more effectively supports this kind of modularization. Despite its relatively stable life, KMPS is not without problems. Perhaps the most notable difficulty is the complexity of the interface it presents to the Policy Modules. This interface evolved over time, with features being added as the Policy Modules grew in number and sophistication. The asynchronous nature of the information transmission (i.e., stop messages rather than procedure calls) seems consistent with the natural asynchrony of multiprocessor scheduling, but we are dissatisfied with the complexity of the information transmitted. For

example, a PM may receive the notification that a process has unblocked *before* it knows the process blocked in the first place. Such peculiarities are bound to complicate the logic of the PM and suggest that the interface is less than ideal. The large scheduling turn-around time (i.e., the computation needed for KMPS to return a process to its PM, have the PM grant the process additional time resources, and restart the process in KMPS) also indicates an improperly organized interface.[11]

On the other hand, the use of a mailbox for communicating between the kernel and PMs seems to us like the right approach. It has the distinct advantage that errors and failures in a PM cannot cause the kernel to fail. Moreover, while poor performance of a PM may impede the efficiency of the processes it controls, it will not affect the performance of the kernel or the processes of other PMs. Our major mistake here was simply not using ports, as we did for I/O devices (see Chapter 14). In a second implementation we would unify these three concepts.

It is interesting to note two features of KMPS that were overlooked in the original design and added subsequently. First, although we always expected that C.mmp would have a heterogeneous collection of processors, we did not foresee all the mechanisms that would be required to optimize scheduling. In particular, a process with no special scheduling constraints ought to be allowed to execute on the most "desirable" available processor. For example, if both a PDP-11/20 and and 11/40 are available, the process should execute on the faster 11/40. Furthermore, if an 11/40 becomes available, KMPS should consider moving a process to it from an 11/20. We neglected to include a mechanism for this rescheduling situation and had to add it later.

Second, we initially failed to recognize the need for inexpensive "kernel" processes; that is, processes that execute exclusively in kernel space and therefore do not need the full overhead of a PROCESS object, an LNS, a CPS, a stack page, etc. After we finally appreciated the desirability of such processes we implemented them, but were unable to make them as inexpensive as we would like. We suspect that the lack of this feature markedly, and adversely, affected the structure of some kernel components, particularly the I/O System (see Chapter 14).

In summary, and unlike much of the rest of the system, we feel much better about the implementation of KMPS than about the abstraction it provides—especially the interface with PMs. In terms of its original goals, KMPS certainly allows a broad range of PMs to be defined. It *is* possible for several PMs to coexist simultaneously. It does perform short-term scheduling decisions rapidly, and the rich set of scheduling parameters do not adversely affect this. The implementation is relatively compact and has a clean structure.

[11]Relevant performance measurements are presented in Chapter 16.

CHAPTER

THIRTEEN

PAGING

The design of the Paging System is influenced greatly by the underlying C.mmp architecture, which puts rather different demands on paging mechanisms than is common in other large computer systems. There are two principal problems.

1. The PDP-11 architecture provides a user program with only a 16-bit address. This means that a process may address at most 64K bytes of memory without intervention by the operating system, far less than the process' expected share of C.mmp's large memory.
2. The memory relocation hardware divides the address space into eight pages of fixed size, making it difficult to manipulate small segments (such as records in Pascal or similar languages).

An accepted technique for paging in other large operating systems is *demand paging* [Den70], whereby the system hardware and software work together to dynamically map a program's large address space onto a smaller amount of physical memory. Typically, only a portion of the user's program or data is in memory at any one time; the rest is kept on secondary storage until it is actually referenced. This approach works because most programs show a high degree of locality in their memory references.

On C.mmp, however, the situation is reversed—the user program sees a small address space over a much larger physical memory. Even to do demand paging within the 64K virtual space is almost impossible: a single PDP-11 instruction can touch up to six different pages and at the same time have side effects on registers, making instruction suspension too difficult. Faced with these problems, we adopted the strategy of *not* hiding the hardware characteristics from the user programs. Instead, we gave the user protected control over the hardware relocation registers and provided a mechanism that would allow (force) him to explicitly specify his working set.

The *working set* of a program [Den70] is a concept originally developed in conjunction with demand paging. In brief, the working set of a program is that portion of the program's virtual address space which is being accessed so frequently that it should be immediately available. Demand paging algorithms are designed to identify the program's working set on the basis of the pattern of memory requests and to attempt to keep each program's working

set in primary memory while the program is executing.

On C.mmp, the working set of a large program will usually be larger than the set of eight pages that is currently addressable. On the other hand, the working set will probably be less than the total memory that the program could potentially access. Hydra therefore provides an explicit representation of this working set.

13-1 THE USER'S VIEW OF PAGING

The Hydra Paging System defines and supports three object types:

- PAGE, which is a virtual primary memory page of 8K bytes
- CPS (for "core page set"), which is the representation of the working set
- RPS (for "relocation page set"), which is the representation of the user's relocation registers[1]

(We must be careful to distinguish between a PAGE *object* and its physical equivalent, a *page of primary memory.* Context will usually disambiguate the meaning, but if necessary we will emphasize which we mean by referring to page *objects* or page *frames*, the latter being the piece of primary memory.)

Page objects need not be associated with a particular LNS or process. They may be created at any time, and capabilities for them may be stored in catalogues, UNIVERSAL objects, procedures, or anywhere else. They may be shared as freely as other Hydra objects. The set of pages reachable on a capability path from a particular LNS constitutes that LNS's *virtual address space.* It should be clear that this means that an LNS's virtual address space is both nonlinear and unbounded. (See Figure 13-1.)

The CPS represents the working set of an LNS. Conceptually, every LNS has associated with it a single, unique CPS object. The C-list of a CPS only contains capabilities for page objects, and the presence of such a capability in a CPS is defined to mean that the page is included in the current working set of the LNS. LNSs may move pages in and out of the CPS at will, and by so doing alter their working set. There is no theoretical limit to the number of pages an LNS may have in its CPS, and the order of pages in the CPS's C-list has no significance.

Users do not have direct access to CPS objects. Instead, Hydra provides two Kalls to manipulate the working set.

[1]For efficiency reasons, an RPS is not implemented as a true object. However, it is convenient to think of it that way.

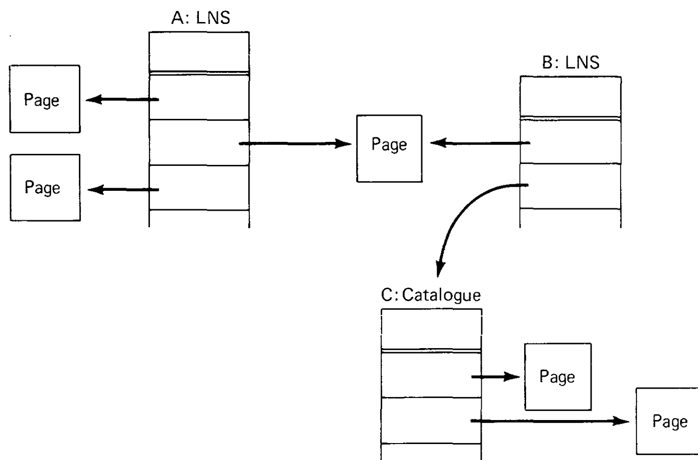

Figure 13-1 Pages in the virtual address space of two LNSs

CPSLoad(CPSSlot:integer, Page:object(PAGE, CPSLoadRts))

Loads a page capability into a slot in the CPS, thus adding the page to the working set of the LNS. *CPSLoadRts* is an auxiliary right for pages.

CPSUnload(CPSSlot:integer)

Removes a page from the CPS and the working set of the program.

Loading a page into the CPS does not immediately make it addressable; the working set may contain more pages than the seven which can be addressed at any one time.[2] The user designates which of his working set pages to make addressable by loading pages from his CPS into his RPS. The RPS associated with each LNS has seven slots corresponding to seven of the eight user-space relocation registers on a C.mmp processor; placing a page in RPS slot i has the effect of making the page addressable beginning at address $i \cdot 20000_8$. One Kall is available to manipulate the RPS:

RPSLoad(RPSSlot, CPSSlot:integer)

Loads the page in CPS slot *CPSSlot* into RPS slot *RPSSlot*. Implicitly unloads any page already in that RPS slot. *CPSSlot* may be 0, in which case the effect is to empty the RPS slot.

A few details about these mechanisms should be explained:

- *CPSLoad* conceptually brings a page into primary memory. If the page is

[2]There are eight user-space relocation registers, but one (the first) is assigned to the *stack page*, over which the user has no direct control.

not already in primary memory, *CPSLoad* only initiates the transfer of the page from the secondary store. The first call on *RPSLoad* for that page will block until the necessary I/O completes.

- Pages have an auxiliary right, *WriteRts*, which controls whether the user may write into the page. If the page capability specified in *RPSLoad* lacks *WriteRts* or the generic right *ModifyRts*, then the relocation register will be set up with the Write-Protect bit set. Any attempt to modify the page will result in a hardware trap.
- The relocation registers corresponding to empty RPS slots have the NXM bit set. (See Section 2-1.2.) Any attempt to touch an address in the range of those registers causes a hardware trap.

Mechanisms exist for specifying in a PROCEDURE object a number of implicit *CPSLoad* and *RPSLoad* operations which must occur before any LNS instantiated from the procedure may start. This ensures that the CPS and RPS are configured correctly when transfer is made to the LNS's starting address.

13-2 THE WORKING SET AND SCHEDULING

Paging and scheduling policies in Hydra are closely connected. In particular, a process under control of KMPS is guaranteed to have its working set in primary memory. Thus when a Policy Module gives a process to KMPS to be run, KMPS must first swap in the working set. When KMPS returns a process to the Policy Module, the working set is eligible to be swapped out. This policy is implemented at a number of places:

1. When a PM starts a process, KMPS must swap in the working set of the currently active LNS. The process does not enter KMPS' feasible queue until the swap-in is complete.
2. When an LNS executing in KMPS Calls a procedure, the working set of the calling LNS is immediately eligible to be swapped out and the working set of the new LNS is swapped in. A similar action occurs when an LNS returns to its caller.
3. KMPS may refuse to start a process, or may prematurely stop a process, on the basis of paging demands. This occurs during process start or during the Kalls *Call*, *Return*, or *CPSLoad* for one of two reasons:

    a. There is insufficient physical memory to accommodate a new or expanded CPS.
    b. The PM-supplied scheduling parameter, *WorkingSetLimit*, is exceeded by the (requested) number of pages in the CPS.

    In either case, the PM may try to start the process later when it thinks

there is more primary memory available or after it adjusts *WorkingSetLimit*.

Thus, working set policies are closely bound to scheduling policies and are under only limited control of the Policy Modules.

13-3 IMPLEMENTATION

The Paging System manages a virtual memory at three storage levels:

Primary memory. About 140 pages of shared memory are available for users (based on the configuration in Table 2-2).

Drum. A variable number of fixed-head disks (which we refer to as "the paging drums"), each capable of holding 128 pages, are available for swapping.[3]

Disk. A single 130-megabyte disk provides permanent storage for pages, and may act as a swapping medium of last resort if there is insufficient space on the drums.

In general, the Hydra Paging System is not too different from other operating systems. The only real difference arises from the fact that pages may be shared among processes, and those processes may be executing on different processors at the same time. In particular,

- A page may be in the working set of several processes, and hence swap-in and swap-out requests may come independently and asynchronously.
- A page may actually be in the RPS of two executing processes, which means that two processors may be writing into the page simultaneously.

To deal with these complications in an orderly way, we treat pages as finite-state machines. The portion of the Paging System dealing with pages is conceptually table-driven; each page exists in one of several states, and demands on pages from higher-level software are turned into state transitions. The most important page states are listed below.

[3]The number of drums available depends on which processors are running and on the drum requirements of other Hydra systems, such as the GST.

State	Meaning
Active	The page is in at least one working set.
Inactive	The page is in no working set, but it is in primary memory and a valid copy exists on disk or drum.
Swapped	The page is in no working set and has no primary memory allocated to it.
Dirty	The page is in no working set, but it is in primary memory. No valid copy exists on secondary storage.

A semaphore is associated with each page to provide mutual exclusion while the page state is changing. The most important functions on pages are listed below.

Function	Meaning
SwapIn	The page is being added to a new working set and must be brought into primary memory.
SwapOut	The page is leaving a working set.
WriteOut	Write the page to secondary storage (invoked by the Paging Demon).
Revoke	Release the primary memory associated with the page (invoked by the Core Module).

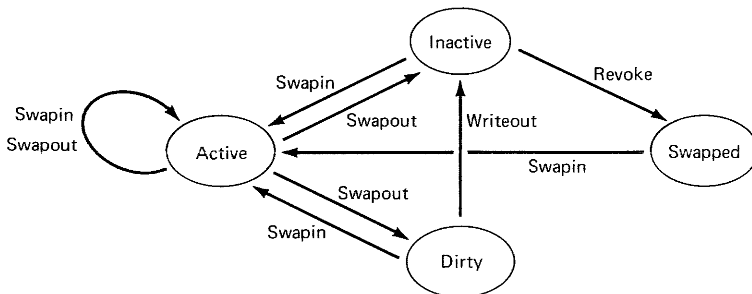

Figure 13-2 Legal state transitions for pages

Many of the page-state functions are legal only when the page is in a certain subset of states; the legal transitions are depicted in Figure 13-2. As an example, *Revoke* is legal only when the page is in *Inactive* state, state *Active* indicates that someone is referencing the page, *Dirty* indicates that the page must be written out before it can be revoked, and *Swapped* indicates that there is no memory to revoke in the first place.

13-3.1 Page Replacement Policy

Page replacement policy governs the treatment of pages in primary memory which are no longer part of any active CPS. To free the memory for another page takes a significant amount of time, since if the old page had been altered it must be first written out to backing store. This updating task can take up to 30 ms, and there is always the chance that the page will be called back into memory almost immediately. (This often happens with the pages of the caller of very short procedures, such as "Catalogue Lookup.")

The page replacement policy is made possible by close interaction between two modules within the Paging System: the Page Module and the Core Module. The Page Module is the subsystem for implementing the PAGE object type; it manages the page states and transitions noted above. The Core Module is responsible for the allocation of C.mmp's entire complement of shared memory; it is used by the Page Module and by Hydra's internal storage allocator. The Core Module maintains a Core Table which records the state of every page frame (either *Allocated* or *Free*).

The replacement policy attempts to balance the need of the Page Module for page frames and the need of Hydra's other systems for memory (for the GST, for I/O request blocks, for the Message System, etc.). This is done by using a process called the Paging Demon to maintain a pool of free page frames which can be acquired by the Page Module for users or by the kernel storage allocator for other modules. The system works as follows: when a page is no longer in any working set, its state changes to *Inactive* or *Dirty*, depending on whether the page contents were modified since the last time it was swapped out. (This information comes from the "dirty bit" in the relocation registers.) The associated page frame is correspondingly marked *Inactive* or *Dirty*. If the page should be brought into a working set later, the Page Module will invoke the Core Module to reclaim the page frame and set its state to *Allocated*.

Meanwhile, the Paging Demon process continually monitors the free storage pool. (These page frames are marked *Free* in the Core Table.) When the free pool shrinks below a threshold level, the Demon scans the Core Table looking for page frames in the *Inactive* or *Dirty* states. *Inactive* page frames are moved to the free pool immediately after invoking the *Revoke* operation on the associated page object. When the Demon encounters a *Dirty* page frame, it invokes the *WriteOut* function on the associated page object, causing the page to be swapped out to drum or disk. Eventually the page frame will change from *Dirty* to *Inactive* and can be reclaimed for the free storage pool.[4]

[4]In this discussion we have ignored many issues in synchronization between page objects and the Core Table. These and other problems (such as I/O) require careful programming, but are otherwise not interesting.

13-4 RETROSPECTIVE

The Paging System has worked well throughout Hydra's history, and much of the credit is due to the clean structure that resulted from the Paging Module's finite-state-machine design. The division of the Paging System into Core Module and Page Module resulted in some fairly tricky interconnections, but was probably necessary because of the Paging System's responsibility to both high-level and low-level software. Indeed, most operating systems have difficulty in fitting paging and I/O into any clean hierarchical arrangement.

Two less foresighted aspects of the Paging System were the decisions to implement both stack pages and CPS objects on a one-per-process basis. Both of these objects are conceptually local to an LNS, but we wished to avoid the overhead that would be incurred by having to allocate both a new page and a new object whenever an LNS was instantiated. In the present design, the *Call* and *Return* Kalls alter the hardware stack-limit registers to isolate the area of the stack available to one LNS from the area used by LNSs higher in the call stack. Similarly, the CPS object's C-list is partitioned as LNSs are called. This design means that an LNS cannot be sure of how much stack space or CPS space will be available to it.[5] We would certainly redesign these mechanisms if we had the opportunity.

The Paging Demon works quite well in balancing memory requirements, although it was not the first mechanism implemented for this purpose. Initially we tried a priority scheme that attempted to determine how soon a page might be referenced, based on its "depth" in the stack of working sets associated with a process. *Call* and *Return* would initiate I/O requests to swap code pages in and out of primary memory. Despite various optimizations to short-circuit unnecessary I/O, this mechanism performed poorly and was scrapped in favor of the Paging Demon.

Finally, it seems evident that the interaction of paging and scheduling policies is not ideally supported by the kernel mechanisms. In general, Policy Modules should be able to manage the set of processes in core independently of the set of processes in the scheduling queues. We feel that the KMPS/PM interface is already complex enough to discourage the addition of any more features. Obviously, this is a difficult problem area and one which, despite substantial attention, has not been adequately resolved. If we were building Hydra anew, we would rethink this tricky interface.

[5]It can, however, determine if adequate stack and CPS resources are available before endeavoring to perform its function.

CHAPTER

FOURTEEN

INPUT/OUTPUT

The Hydra kernel provides a primitive mechanism for accessing the peripheral devices connected to the C.mmp processors. In keeping with the principle of policy/mechanism separation as discussed in Chapter 3, the I/O system seeks to supply only a base-level means of performing input-output operations. High-level policies for convenient use of peripherals are relegated to non-kernel software.

The I/O system is perhaps the closest approximation to a pure mechanism that the kernel supplies (with the possible exception of KMPS—see Chapter 12). As such, it provides an abstraction that closely parallels the one provided directly by the hardware, except in two practically important ways:

1. The operations defined by the I/O system are safe (i.e., protected) versions of the corresponding hardware operations. This prevents both blunders and malicious programs from destroying vital data and is clearly essential to a multi-user system.
2. The abstraction simplifies the interface to the peripherals by replacing the heterogeneous connection structure (miscellaneous devices attached to various UNIBUSes) with a homogeneous one (all devices equally accessible).

These two properties of the I/O mechanism are discussed in greater detail below.

14-1 THE HARDWARE ENVIRONMENT

As we saw in Chapter 2, all devices on C.mmp are controlled by registers located in the *I/O bank page*, which is a page of addresses in the kernel address space. Interrupts are passed to the CPU through *interrupt vectors* located in the processors' local memory. The most primitive (and clearly unsafe) kernel mechanism one can imagine would merely copy user-supplied values into specified I/O locations. The actual kernel mechanisms try to restrict this uncontrolled access as little as possible and still achieve a system that provides safe access and a modicum of convenience. In doing so, they must address the following questions:

1. Several devices may be attached to a UNIBUS through a single shared controller (e.g., disks, terminals). Can these devices be regarded as independent by higher-level software?
2. I/O devices can be connected to only a single UNIBUS, and thus may be manipulated directly by only one processor. Must higher-level software that performs an I/O operation execute on the appropriate processor?
3. How are interrupts and associated status information reflected to the higher-level software?
4. High-speed devices generally perform direct-memory access (DMA), stealing cycles from the processor. Such accesses require valid contents in the relocation registers that are used implicitly, or chaos will result. How are the contents of these registers maintained during DMA transfers?
5. Some devices (e.g., tapes, keyboards) have very limited buffering or other properties that impose real-time constraints. How are such constraints met, particularly when the controller is shared by multiple devices?
6. Some devices exhibit recoverable errors. Should an attempt be made to retry or correct those errors behind the program's back?

14-2 THE USER'S VIEW OF I/O

Perhaps the easiest way to present our answers to these questions is to give a brief description of the steps the I/O system executes to perform a single I/O operation. For definiteness, we will assume that a high-level program wishes to read a block of words from a particular disk drive.

In Hydra we make an explicit attempt to make I/O look like interprocess communication, and therefore I/O is integrated with the Message System (Chapter 6). All I/O devices are represented by objects of type DEVICE, which resemble PORTs with one input channel and no output channels. Thus, in our example, any user with a capability (with *ConnectRts*) to the disk device may connect a port to that device. Once the connection has been established, the user may send messages to the disk, each message representing a request to perform an I/O operation. The format of the message is defined by the I/O system; it includes

- The operation to be performed (e.g., "read")
- A buffer (usually in the message itself)
- The disk address from which to read

Sending the message logically initiates the operation, and the user program may proceed in parallel with the transfer. When the disk has successfully completed the read operation, it puts the data into the message buffer and replies the message back to the user. The user can invoke the *ReceiveMsg* Kall at any time, suspending his process until the reply is received at his port. He can then retrieve the data from the message (Figure 14-1).

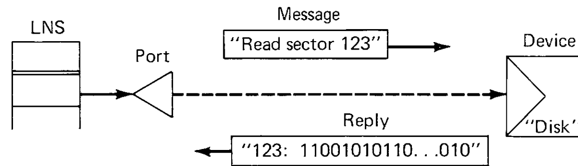

Figure 14-1 A user's view of I/O

Returning to the questions posed in the preceding section, we see that the I/O system enables higher-level software to regard peripherals as uniformly accessible, independently protected entities. In particular,

1. The point of physical connection to a controller and UNIBUS is irrelevant to the user program.
2. A hardware-generated completion interrupt is mapped to a Message System reply, thereby allowing the user program to synchronize with the transfer completion when it chooses.
3. No explicit manipulation of relocation registers by the user program is required to ensure proper DMA transfer of data from the disk.
4. The user program is relieved of the chores of optimizing arm motion and minimizing rotational latency, which require real-time knowledge of the read/write heads' position.
5. The user program can ignore the possibility of transient, recoverable errors. When it receives a reply to an I/O request, either the request was satisfied or some unrecoverable error occurred.

It might be argued at this point that the mechanism described is far from primitive, since it answers most of the questions posed earlier by saying, in effect, "let the kernel do it." Indeed, more primitive mechanisms that are still "safe" could be defined but would not mesh as conveniently with other pieces of the kernel abstraction, notably the Message System and the protection mechanism. On the other hand, the abstraction offered by the I/O system is at a substantially lower level than the ones typically present in conventional, multi-user operating systems (e.g., OS/360 and relatives, TOPS-10 and relatives). For instance, the I/O system does *not* provide:

1. *Mechanisms to ensure mutual exclusion of access to peripherals.* Any program with an appropriate capability can connect to an I/O device and send it arbitrary requests. Thus, in principle, two separate user programs could simultaneously connect to the same tape drive and interfere with each another. It is the responsibility of higher-level software to ensure appropriate mutual exclusion by restricting the distribution of device capabilities.
2. *Device-independent requests.* While messages tend to have one of a few

common formats, no attempt at standardizing formats across device types has been made. A message that the disk interprets as a read request may cause a read-backward operation if mistakenly sent to a tape drive or may be rejected outright by a line printer. All operations are close analogues of those that the hardware performs directly, and only infrequently does a single request induce multiple device operations.[1]

3. *Buffering facilities.* Multiple buffering can easily be achieved by queuing multiple requests in the Message System, but even then the user program is responsible for managing the message resources required for requests, and it must ensure that replies are processed in the correct order.
4. *Access modes.* The I/O mechanism has no knowledge of volumes, file structures, directories, or other higher-level concepts frequently used to organize secondary storage media.
5. *Logical addressing.* Devices that require addresses (e.g., disks, DECtapes) do not support addressing abstractions such as "logical record number." All secondary storage addresses manipulated by the I/O system are physical, not logical. Higher-level software (e.g., a file system) must implement these abstractions and appropriate mappings.

14-3 IMPLEMENTATION

The internal organization of the kernel I/O system is rather conventional and straightforward. A few details, however, pertaining to the architecture of C.mmp deserve mention, since they provide good illustrations of the effects of certain hardware features on the kernel software.

The I/O system is composed of

A *controller-specific module* for each controller type, which includes the following routines:

A *request preparation ("prep") routine*, which checks the validity of I/O requests

A *start routine*, which initiates validated requests at the device

A *service routine*, which handles device interrupts by processing completed requests and starting new ones

A *request driver* and a set of *utility modules*, which route the request through the I/O system and the proper controller-specific module and provide utility functions for those modules

A *onfiguration table*, which describes the configuration and location of every controller and device on C.mmp

[1]A notable exception to this is the "write-and-verify" operation which we implement for those disks used by the GST and Paging. The hardware could have provided this operation, but it didn't.

To see how these elements interact, let's reconsider the disk request we examined earlier in this chapter. When the user program sends the message to the I/O device, a routine in the Message System calls the I/O driver, passing it the message and the device for which it is intended. This information is packaged into a data structure called a *request block*, which is easily transferred around the I/O system. The driver consults the configuration data structure and locates the proper *prep* routine for the controller that hosts that device. It invokes the routine, passing it the request block.

The *prep* routine analyzes the request to decide whether it is legal. If not, it immediately replies to the message (by passing it back to the Message System via a utility module), setting its type to reflect the rejection reason. If the message is acceptable, the *prep* routine must queue it for the controller's start routine, which can execute only on the processor to whose UNIBUS the device is connected. The *prep* routine then exits, requesting that the appropriate processor be notified that a new request has been queued for one of its controllers. (See Figure 14-2.)

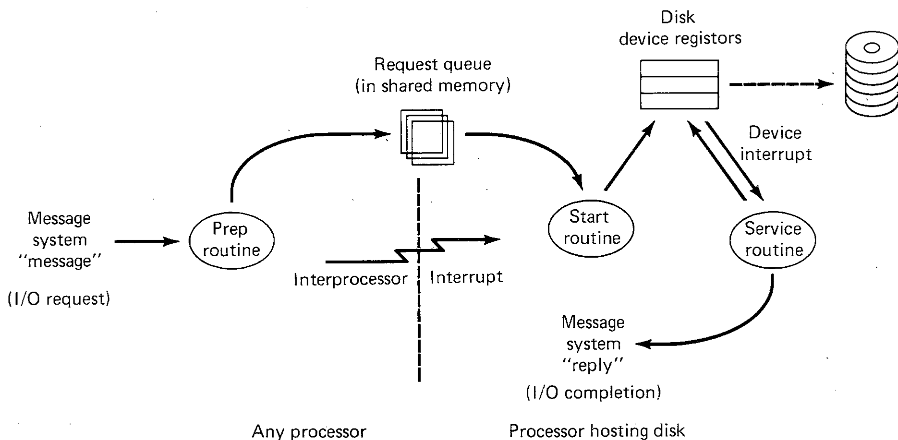

Figure 14-2 I/O communication structure

The driver uses the interprocessor interrupt hardware (Chapter 2) to alert the designated processor. At this point, control returns to the user program that originally issued the request.

The interprocessor interrupt will be processed asynchronously on the processor that receives it. Code in the I/O driver will field the interrupt and alert the appropriate controller-specific module by calling its *start* routine, which will check if the controller is idle, and if so, dequeue and initiate the request. When the operation completes, the controller's interrupt will be fielded by the controller's *service* routine, which replies to the original message, setting the reply type to reflect the outcome of the operation.

We see, then, that the kernel I/O system has a rather simple structure, with all the device-specific processing confined to a single module (in fact, three primary routines) per controller type. The preceding description ignores the details of parametrization that permit multiple identical controllers to share the same code, even if they are attached to different UNIBUSes. Issues concerning the synchronization of *start* and *service* routines, which may interrupt each other, have also been ignored in the description. The resolution of these details involves careful design, but nothing particularly unusual.

14-3.1 Interprocessor interrupts

A word or two concerning the use of the interprocessor interrupt mechanism is in order. Multiple interrupt requests for the same processor are merged by the PDP-11 interrupt hardware, implying that more than one *start* routine may have to be invoked by the interprocessor interrupt handler. In fact, the same routine may have to be invoked more than once, in principle. To simplify the bookkeeping, the *prep* routine actually queues each validated request on a queue associated with the destination processor. The interprocessor interrupt handler on that processor dequeues each request in turn from this queue and passes it to the *start* routine, which enqueues it for the controller unless the controller is idle. Therefore, no auxiliary structure is required to maintain the set of *start* routines to be involved by the interprocessor interrupt handler.

14-3.2 DMA Transfers

Aside from the complexities of managing its devices, each controller-specific module must be concerned with the memory addressing performed by DMA operations. All controllers that perform direct transfers to memory generate full, 18-bit addresses on the UNIBUS. By convention, the device control modules force these addresses to be in the I/O-space of C.mmp's relocation machinery. Before a DMA operation is started, the controller's *buffer address register* is loaded with a value that has been adjusted to refer to some relocation register in the I/O-space. This register is loaded with the appropriate value to access the page containing the buffer required for the operation. It is the responsibility of the controller-specific code to ensure that the proper values appear in these registers for the duration of DMA activity.

Figure 14-3 shows an example of this situation. Suppose processor 6 has both a disk and a drum on its UNIBUS, and is executing process *A*. At the same time, the I/O space relocation registers are set to allow concurrent DMA transfers to pages in processes *B* and *C*, which may be simultaneously executing on other processors. If process *A* should be rescheduled on another processor, the I/O-space relocation registers will not be affected.

Likewise, should I/O to either *B* or *C* complete, the I/O system will alter the I/O-space registers without affecting *A*.

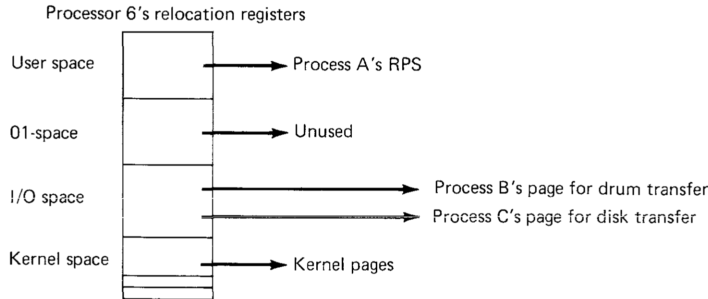

Figure 14-3 Relocation register contents during DMA I/O

Because the I/O-space relocation registers are a resource shared by all DMA controllers on a UNIBUS, a utility module in the I/O system provides a simple allocation mechanism that ensures proper use of this resource. In principle, a controller may be unable to initiate an operation for lack of a relocation register, but in practice, no UNIBUS has sufficiently many DMA devices (more than 7) to force this situation.

14-3.3 Error Recovery

In practice, the receipt of an I/O interrupt does not always imply a successful completion. Generally speaking, the more complex the device and its controller, the more things that can go wrong. However, the user program generally prefers to think of a request as either succeeding or failing; transient errors are not of interest at the higher level.[2] Accordingly, transient errors are handled within the I/O system and never passed back to the user program.

Specifically, when a service routine fields an interrupt, it checks for successful completion of the operation. If an error occurred which the routine considers to be recoverable, it initiates some corrective action (usually a retry of the failing operation). This generally induces a subsequent interrupt which the service routine must also process, implying that a certain amount of state information must be maintained for requests that have been physically initiated but not successfully completed. Since more sophisticated controllers can process multiple requests simultaneously (e.g., one seek per disk drive), the service routine must be prepared for multiple requests being

[2]For example, a particular disk controller that Hydra supports can generate over 30 different error conditions, many of which are recoverable. It is difficult enough for the I/O system to respond intelligently to all these cases; user programs should be spared the inconvenience.

retried at the same time. All these complexities, however, are hidden from the user program, which eventually receives either a "success" or "failure" reply.[3]

Memory contention can also represent a problem for the I/O system. If contention is severe, shared memory will not be able to respond fast enough during a DMA transfer, causing the transfer to abort. In this rare instance, we retry the operation several times, enabling the crosspoint switch's high-priority feature to bypass the normal priority resolution scheme (see Section 2-3.1). If even this is not sufficient, a special call on KMPS causes all processors to busy-wait, accessing only their local memories until the transfer completes.

14-4 KERNEL I/O

Before proceeding to an evaluation of the I/O system, we should indicate its relationship to the rest of the kernel. The design described above supports the user-level view of I/O as a special case of general message communication. However, the kernel itself has a fundamental need for input/output facilities as well, e.g., to implement the GST. The GST must be able to manipulate disks that hold the long-term representations of kernel data structures; other kernel modules must occasionally communicate with an operator's console or system log device. The I/O system must supply a convenient model for these mechanisms as well.

Because kernel services operate "below" the level of the protection mechanism (i.e., without LNSs or any implicit addressing of capabilities), it is inconvenient for them to use the same I/O facilities that user programs do. Accordingly, kernel subsystems issue I/O requests without going through the Message System. They specify devices by a unique name instead of by a Message System connection, and they invoke the I/O driver directly by subroutine call, instead of by sending a message. The subroutine arguments

[3]Strictly speaking, this may not be quite adequate. For example, if a particular sector on the disk has become difficult to read, it may be advantageous to relocate its contents. This decision must be made by higher-level software, not the I/O system, but the higher level must somehow be notified that the I/O system encountered difficulties. Such a mechanism does exist and is occasionally used. Since the outcome of a request is reported in the message reply type, a small number of types are reserved by convention to mean "success," another set to mean "failure." A single success type is universally considered "unconditional success"; the remaining ones can be assigned device-specific interpretations. In this way, the user program can be notified of unusual but non-fatal problems in satisfying its request.

correspond directly to the information a user would place in a message.[4] From this point on, the kernel I/O request is treated just as a user program's request would be; that is, it passes through the same *prep/start/service* sequence. When the request has completed, the *service* routine hands it back to the I/O driver, which then notices that the request originated in the kernel. Instead of performing a Message System "reply," the driver calls a subroutine supplied with the original request. This routine receives as arguments the same information that would form the message reply type for a user program's request. Typically, the routine will 'V' a semaphore, thus unblocking some process waiting for the I/O to complete.

Because the kernel has the same I/O facilities available to it that a user program does, no special coding is required in the device-specific modules to permit kernel access, and the kernel is freed from the correspondence between processors and devices. A small amount of additional machinery is introduced into the driver and utility modules, and the kernel I/O model is procedure-based rather than message-based.

14-5 RETROSPECTIVE

One generally expects a kernel mechanism to be small, i.e., to require a modest amount of code to implement it. The kernel I/O system is approximately 23,000 lines of source code; hardly a small program. However, this number is strongly influenced by the number of different devices it supports. If we consider only the driver and utility modules, we find they require approximately 5,500 lines of code; the remaining 18,000 or so constitute device-specific modules that implement 18 different types of controllers.[5] The device-specific modules have a very regular structure imposed by the requirements of the driver, and for a simple device are easy to write. (On several occasions, the low-level support for a new device was produced from scratch in less than two programmer-weeks). Viewed in this way, the size of the I/O system seems more a function of its regular structure than of any inherent complexity.

Access to the I/O system is reasonably well-integrated with the other kernel facilities. The general bias in the kernel interface is procedural; nevertheless, the decision to access I/O devices with messages from ports is clearly the correct one. I/O activity in C.mmp is inherently asynchronous, so

[4]An exception to this is the location of the data buffer itself. The kernel always specifies the buffer indirectly with an (address, length) pair. Users generally place the data buffer in the message, although they have the option of using a similar indirect specification.

[5]The I/O system supports three different disk controllers (two for moving-head drives, one for fixed-head drives), two tape controllers (one conventional magtape, one DECtape), two kinds of line-frequency clocks, an ARPANET interface, two kinds of bit-serial asynchronous communication lines, a line printer, various terminals, and at least six experimental devices.

the ability to perform I/O in parallel without spawning a process (expensive in Hydra) is important. The general waiting and replying mechanisms of the Message System fit well with the I/O system's needs and avoid introducing an additional set of mechanisms explicitly for I/O. More importantly, the fact that I/O devices are indistinguishable from ports permits a program to simulate the behavior of a device by supplying a port where a device is expected and interpreting messages appropriately.

The absence from the kernel of buffering, access modes, device independence, and other higher-level I/O structuring concepts does not imply that they are unimportant. Indeed, practical, flexible use of the kernel mechanism strongly suggests that some collection of such facilities be built on top of it. It is disappointing that no single high-level I/O model was ever constructed. As always in Hydra, the existence of such a facility would not preclude the direct use of the kernel mechanism in appropriate cases. Rather it would serve to simplify the vast majority of cases in which the functional I/O demands of a particular program are confined to a small, common set. The absence of this facility has hindered convenient program development for Hydra/C.mmp.

The kernel I/O system was originally designed to permit dynamic reconfiguration of peripherals. (It is for this reason that devices are a kernel type distinct from ports; auxiliary rights were to be used to protect reconfiguration operations.) However, much of the code required for reconfiguration was never implemented. It was originally thought that a console operator (or perhaps internal software checks) could request that a malfunctioning processor be shut down and that its key peripherals be switched to other UNIBUSs. However, the necessary hardware to support this reconfiguration of devices was never acquired, and the detailed mechanisms outside the I/O system were never designed.[6] Such a facility, if successful, would provide an interesting validation of the generality of the I/O system's structure and the flexibility of its internal communication and synchronization schemes. It should be noted, however, that the need for such a reconfiguration mechanism has almost never arisen.

The I/O system, unlike the GST, was not designed with reliability as a primary goal. That is, few internal consistency checks exist, and mechanisms to recover from internal errors are practically non-existent. Observation of failure modes suggests that a better scheme for handling lost device interrupts should probably have been included as part of the original design. (It should be noted that interprocessor interrupts *are* reliably transmitted; a mechanism in the driver maintains a list of processors that have been sent interrupts but have not responded. This list is used at appropriate intervals for retransmission). A lost interrupt from a device will effectively "hang" that device, since no time-outs are imposed by the software. Except for this

[6]In conjunction with the general system error-recovery mechanisms, it *is* possible to "late-start" a processor, dynamically adding it and its I/O devices to the system.

deficiency, however, the lack of systematic internal error recovery mechanisms in the I/O system has not been a significant influence on overall kernel reliability. Indeed, the regular structure of the I/O system has tended to minimize the number of latent and subtle bugs. The uniform approach to synchronization has virtually eliminated the occurrence of internal deadlocks. The remaining bugs generally appear in device-specific modules where they are relatively isolated and tend not to compromise the overall integrity of the I/O system.

With a more modern perspective, one might debate the use of substantial interrupt routines to process I/O completions. Today, one is much more inclined to view an I/O completion as triggering a process switch, and lacking hardware or firmware to perform this switch directly, one would build interrupt routines that merely 'V' a semaphore (or, if monitors [Hoa74] are used for synchronization, "signal" a condition variable). We seriously considered this approach in Hydra and rejected it on efficiency grounds, since process switching is time-consuming and processes are space-consuming. Consequently, interrupt routines (and everything they call) are subjected to certain global restrictions that prevent them from being rescheduled to execute on a different processor.[7] Such restrictions are awkward to enforce and have been the source of many hard-to-find bugs. With a less costly process mechanism we would definitely have organized I/O interrupt handling differently. Given the facilities we had, the choice of interrupt routines was undoubtedly correct, but we might have been able to find a less constraining set of global restrictions that would have engendered fewer bugs. Alternatively, we might have attempted to build a process mechanism that permitted rapid context-switching and consumed only a small amount of storage per process. Some preliminary designs aimed at producing such a mechanism were never completed.

[7]In particular, an interrupt routine is not permitted to 'P' any semaphore, since blocking would cause a context swap. This effectively prohibits any use of the GST within interrupts because the GST makes extensive use of semaphores for synchronization.

CHAPTER

FIFTEEN

ERROR RECOVERY

The term "error recovery" really describes a continuum of actions that collectively ensure some degree of robust operation. At one end of the spectrum, checking and reporting invalid parameters to a requested operation constitutes a (simple) form of error recovery. At the other extreme, a system restart with complete validation of the GST structure is a much more radical recovery technique. Between these limits are a variety of actions, some of which interrupt the continuity of system operations. In the programmer's vernacular, these are "crashes" of varying severity. The inherent redundancy of a multiprocessor at once complicates the crash recovery task and offers the hope of reducing the frequency or severity of crashes.

Hydra employs five broad categories of mechanisms to deal with the possibility of hardware and software errors:

Validation mechanisms try to ensure that hardware components are operating correctly before they are used. These mechanisms should be quite conservative about what components they allow into the system.

Fault-tolerant mechanisms are employed in situations where certain passive errors (such as lost interrupts) can be ignored. These mechanisms can be less than 100% effective; every little bit helps.

Error detection mechanisms are designed to catch errors quickly, before serious damage is done.

Diagnostic mechanisms are designed to analyze errors and determine the extent of damage and the possibility of recovery.

Recovery mechanisms are designed to take corrective actions, ranging from ignoring the error to notifying higher-level software to reloading the entire operating system.

In this chapter we will try to enumerate the various techniques employed by Hydra in these five categories and evaluate their success.

15-1 VALIDATION MECHANISMS

Hydra's validation mechanisms are invoked during system initialization to determine the status of various hardware components and data structures.

Memory Validation Hydra makes no assumptions about the configuration of the shared memory in C.mmp; the configuration must be determined when Hydra is booted from disk. A bootstrapping ROM finds the first good memory page and loads into that page a routine which walks through the entire 25-bit address space and builds a table of available memory. This table is used as the rest of Hydra is brought into memory, and it acts as a basis for the initialization of the Paging system. This validation is effective in detecting malfunctioning, as well as unimplemented (i.e., missing from the physical address space), memory pages.

Processor Validation During initialization, Hydra reads from disk a table indicating which processors are supposed to be available. In turn, each processor is started by being sent an "interprocessor start" interrupt from the bootstrap processor. Processors initialize and validate themselves, posting the results in shared memory; the bootstrap processor builds a table of "available processors" from the results of those initializations.

When a processor which is "supposed" to be operative does not start, a message is printed on the operator's console. The operator can manually start the processor at any later time. (This is partially a response to an observed failure mode: someone left the processor's "halt" switch down.)

GST Validation Before initialization completes, the Passive Fixed Part Directory on drum is cross-checked with information on the Passive GST disk. Any inconsistency in the data causes a longer GST initialization sequence in which the Passive GST is garbage-collected and the GST Directory is rebuilt.

15-2 FAULT-TOLERANT MECHANISMS

In a sense, all error-handling mechanisms are supposed to promote fault-tolerant behavior, but in this section we are concerned with the design of mechanisms which are completely insensitive to certain types of expected errors, especially *lost interrupts.* As has been seen in previous chapters, the interrupt mechanism is very important to Hydra. Interrupts are critical to scheduling and synchronization, as well as being the foundation of the I/O system. Moreover, a lost interrupt is a passive occurrence which is hard to detect. Hydra does not use time-out mechanisms to detect them—such mechanisms are more expensive than simply resending interrupts.

As an example of this, we saw in Chapter 12 that waking a processor

blocked on a lock is accomplished by sending an interprocessor interrupt to *all* blocked processors. If an interrupt to one processor does not get through, it probably will the next time, especially since the next interrupt will probably come from a different processor.[1] If a processor is truly unreachable from all other processors, it will be caught by the "watchdog" (discussed below).

KMPS also depends on the success of the interprocessor interrupt mechanism to do correct scheduling; a lost interrupt could result in a high-priority process remaining on the feasible queue. When KMPS interrupts a processor, it actually resends all previously sent interrupts until it receives positive notification that the interrupt has gotten through. The cost of this mechanism is very low (a couple of instructions).

The I/O system tries to tolerate lost interrupts by the use of the request queues. (See Chapter 14.) Any time a processor receives the "I/O" interprocessor interrupt, it processes *all* requests on its queue. Thus a lost interprocessor interrupt will be corrected on the second interrupt. Likewise, any interrupt from a device will cause the *service routine* to process all pending requests. If no other interrupt arrives, the device will appear to be hung.

15-3 DETECTION MECHANISMS

Various error detection mechanisms have been mentioned earlier in this book in conjunction with particular kernel systems, and there are others of a more general nature. These mechanisms must be very safe, because an undetected error can propagate damage beyond hope of recovery. We will discuss the mechanisms under three broad categories.

Hardware fault detection The PDP-11 hardware, as modified for C.mmp, detects a large number of errors which can result from the malfunction of either hardware or software. These include parity errors in shared memory, attempts to execute illegal or reserved instructions, attempts to access non-existent memory pages, attempts to violate write-protected pages, attempts to violate the stack conventions, and failures of UNIBUS devices to respond.[2] With the exception of parity and device errors, any of these faults may be triggered by software or hardware.

Unfortunately, many hardware faults are not detected directly. Parity errors in local memory, and misexecuting instructions are observed to occur without triggering a hardware trap, and thus must be caught by higher-level mechanisms.

[1]The common cause of this error is a misconfigured Interprocessor Bus—a situation not detected by the validation mechanisms.

[2]This last fault is the all-purpose "NXM" (non-existent memory) exception.

Software consistency checks Hydra, and especially the GST mechanisms, employs a wide variety of techniques to ensure that everything is going correctly. Checksums and back-pointers are used to ensure consistency of data structures. The Paging System uses its page state information to determine whether the application of particular functions to pages are appropriate. Finally, a count of the number of critical sections entered is kept to ensure that no mutual-exclusion semaphores are left locked after processing a Kall-for a user. (Leaving a semaphore locked is an invitation to deadlock.)

Over Hydra's history, all these software checks have been successful in catching both software and hardware errors.

Asynchronous monitoring Sometimes a malfunctioning processor will simply halt, and none of the above mechanisms are capable of detecting that. For that reason we implemented a "watchdog" system in which every processor periodically asserts its well-being and makes sure that every other processor does likewise. Should any processor halt, some other processor will eventually notice that fact and will initiate recovery actions.

15-4 ERROR DIAGNOSIS

The rapid diagnosis of a possible error is vital in a multiprocessor for two reasons. First, damage may be propagated rapidly by other processors. Second, the damage may cause the triggering of error mechanisms in other processors, making a complete analysis much more complicated.

The accuracy with which an error can be diagnosed is dependent in large part on the information the detection mechanisms can make available, and this varies greatly among the mechanisms. In the case of hardware-detected errors, for instance, the location (if not the cause) of the error is usually well specified.[3] Diagnosis of these errors usually consists of first determining if the error was the user process' fault. If so, we can simply reflect the error back to the user in some way. Likewise, parity errors occuring in the context of the user process cannot have affected the integrity of the kernel and therefore they can also be reflected back to the user.[4]

Errors detected by Hydra's software checks are more difficult to analyze because the source of the error is almost always unknown at first. Furthermore, all such errors occur within the kernel and therefore cannot be easily reflected back to the user. (This assumes that sufficient validation of the user's Kall arguments occurs so that a bad argument cannot trigger one of these consistency checks much further on.) In general, all that can be done is to validate and/or repair the environment before proceeding.

[3]As noted in Chapter 2, special tracking mechanisms were added to make this true.

[4]Although it must be anticipated that a hard parity error in a user's page will affect Hydra when it tries to swap out that page.

Errors detected by the watchdog system are the most difficult to deal with because most of the information needed to analyze the error is hidden in the malfunctioning processor and not available to the processor detecting the malfunction. Part of the recovery action for this type of error is an attempt to force the halted processor to copy its state to shared memory.

15-5 RECOVERY MECHANISMS

When an error has occurred and diagnostic information is available, a selection among several recovery actions must be made. The error handlers have four main options:

1. Dismiss or correct the error immediately.
2. Reflect the error to the user process and continue.
3. Reflect the error to higher-level kernel software and continue.
4. Stop and initiate a restart of the entire system.

In selecting among these alternatives, the error handler must consider not only the probability of being able to correct the error but the effect the error may have on other processors running concurrently with the error-handling.

In practice, very few errors can be handled locally. For instance, even though most hardware errors are transient, the PDP-11 architecture makes it very difficult to back up and retry an instruction; instructions can have too many side effects. Similarly, correction of software-detected errors requires more state information than is available in the error handler. Only the I/O system is able to deal well with controller-detected errors, because it has sufficient state information to reset the devices and retry the operations.

The GST is the only kernel module which is able to use redundant information to recover from errors. The GST keeps two copies of each object on the Passive GST—a "most-recent" copy and a "second-most-recent" copy. Should the most-recent copy be unreadable, it will automatically back up to the second-most-recent copy. Should no copy be readable, the object is eliminated and capabilities for it are deleted.[5]

When an error occurs in the context of a user process, that process is normally stopped, and an error indication is given to the responsible Policy Module, which can distribute that information as it sees fit. However, there is also a facility whereby an LNS may designate an error-handling routine which is given control when an error occurs. This routine can try to correct the error or can simply terminate the LNS with a *Return* Kall, in effect punting the error to the caller. This facility is very important for subsystems, because to stop an LNS at an arbitrary point could leave the subsystem's data

[5]It is not clear that this is a good policy—users are disconcerted when their objects disappear without warning. We should emphasize that this behavior is extremely infrequent.

in an inconsistent state. (A semaphore might remain locked, for instance, and effectively prevent all other subsystem operations.)

Reflecting an error to higher-level routines within Hydra would seem to be a very powerful mechanism, but unfortunately the necessary recovery mechanisms were never implemented. Much information needed for successful recovery, especially knowledge about which data structures are locked by the process, is not available.

In practice, therefore, if an error cannot be safely reflected back to the user process, it usually results in a crash-and-reload of Hydra. The advantage of this course is that it is extremely conservative: it does not allow other processors to propagate the damage, it provides an easy way to record a fairly large amount of data for later offline analysis by hardware and software engineers, and it automatically invokes the hardware validation mechanisms (mentioned earlier) during reinitialization. The disadvantage of this recovery mechanism is its severity: the work of all users is disrupted during the reload. Because this recovery action is (unfortunately) common, we will now describe it in more detail.

15-6 AUTORESTART

Once the autorestart mechanism has been triggered, it has a single goal: to restart system operations promptly with a minimal loss of data and a maximal chance of continued operation. It follows that the mechanism must be able to reconfigure the system hardware to eliminate faulty components and must be able to do so automatically, without relying upon a human operator (who may not be present at the time of the crash).

The first step in the recovery is to stop normal system operation (via the interprocessor bus) and select the *Suspect* and the *Monitor* processors. The Suspect is the potentially faulty processor, usually the processor which detected the error. The Monitor is selected at random from the remaining processors to control the restart process, since the Suspect may not be reliable enough to do so.[6] Suspect and Monitor now synchronize with each other, and the Suspect proceeds to record copies of its registers and local memory page in an area of shared memory and then runs a complete processor diagnostic. When the Suspect completes, the Monitor halts it and writes the crash data out to disk for later analysis. If the Suspect should be unable to complete this record, the Monitor will detect the fact and write out whatever portion of the record is available.

On the basis of the circumstances surrounding the error and the performance of the Suspect during recovery, the Monitor determines if any reconfig-

[6]In the case of an error detected by the Processor Watchdog, the processor detecting the fault designates himself the Monitor. After halting other processors, the Monitor attempts to start the Suspect via the interprocessor bus and force it into the autorestart sequence.

uration of the hardware is necessary. If the original error was a memory parity error, for instance, and if the error was repeatable, the bad memory page is deleted from the configuration table used by Hydra. The Monitor consults an error history table for the Suspect processor to determine if the Suspect has failed under similar circumstances in the past. If the Suspect accrues too many demerits in any particular error class, or too many total demerits, it will be designated "faulty" and will be excluded from the system.

Normally, a "faulty" processor is automatically excluded from the system and is forced to execute special diagnostic code under the watch of a small executive within Hydra. Should the processor seem error-free over a period of time, and if an operator has not permanently excluded it, the processor is automatically reinstated in the system. This prevents a flurry of random errors (e.g., power fluctuations) from indefinitely excluding an otherwise acceptable processor.

Occasionally, a faulty processor will be classified as "critical," usually by virtue of its being host to a critical system peripheral device. In such a case the processor is "quiesced"—it remains in the system, but it runs no processes, only device interrupt traffic. It is hoped that the processor will be more reliable under the lighter load.

15-7 RETROSPECTIVE

It may easily be argued that the error-handling mechanisms are too *ad hoc* and incomplete. We admit these properties and attribute them to the historical evolution of the mechanisms. The error recovery scheme was, in essence, an add-on to the kernel dictated by the unreliability of the C.mmp hardware. It would doubtless have been better if we had designed it as an integral part of the original kernel, but we never believed that the hardware would be so unpredictable. The Suspect/Monitor model was developed to deal with hardware failures (particularly misexecuting processors) and works well for such problems. Its general applicability to higher-level software errors is dubious at best.

The most serious shortcoming of the error-handling mechanisms is that continuation after most errors was not implemented. We suspect that most hardware errors encountered while executing in the kernel are safe, by which we mean (1) the processor detecting the error is performing a Kall for a user, and (2) no objects or data structures are locked (and hence they are in a consistent state). We should be able to back out of the kernel and stop the user process, instead of initiating a system restart. Unfortunately, we cannot verify this state because we do not record the objects that are locked, and because we do not believe that all kernel code could tolerate being interrupted at arbitrary locations, even with nothing locked.

Because so much of the total operating system is outside the kernel,

users need very good support for their own error recovery policies. In general, Hydra provides rather poor facilities. They have not been seriously missed largely because we never really explored the possibility of having one process debug another. We did, however, provide mechanisms to forcibly reawaken processes waiting on ports and Policy Semaphores, one essential requirement. Despite the absence of convenient mechanisms, Hydra's subsystems seem to maintain abstractions across crashes as reliably as conventional systems do. In principle, Hydra should be able to do better and, we believe, *would* do better if the recovery system had been planned at the outset.

The "error history" mechanism for processors works quite well in general but has some drawbacks. By associating every error with a particular processor, intermittent memory failures cause a slow increase in the error counts of all processors. In practice, such errors will eventually become "hard," the memory module will be eliminated, and the demerits assigned to guiltless processors will "evaporate" in time. (Or, more likely, the hardware maintainer will discover what has happened and manually adjust the error counts.)

The autorestart mechanism works well at what it is (realistically) intended to do. It reloads and restarts the system over 95% of the time without the loss of anything except currently executing programs. It does so without operator intervention (Hydra has no full-time operator) and leaves detailed records of the system state at crash time. These records have been of enormous assistance in tracking down subtle hardware errors (which escape the rather coarse diagnostics) and latent kernel software bugs. It provides a convenient means for adding a (repaired) processor to the running Hydra configuration long after it has come up, thereby encouraging prompt repair of processors excluded by the Suspect/Monitor mechanism.

Two brief anecdotes should illustrate the effectiveness of the error recovery mechanisms. On one occasion, a particular connection on the interprocessor bus was broken, preventing interrupt signals from being sent from a particular processor. The redundancy in the interrupt notifications was so good that no degradation in performance was noticed, and days went by before we realized that the bus was broken. On another occasion, a serious power failure stopped C.mmp dead in its tracks. When power was restored, Hydra was easily rebooted and it resumed operation without any damage to its data structures on secondary memory. Unknown to us, half of primary memory had been rendered inoperative by the power failure! Hydra simply rebuilt the memory tables and continued; several days elapsed before anyone noticed the missing memory!

On balance, we believe the mechanism has been a worthwhile addition to the kernel, both because of its effect on MTBF and because of the lessons we learned by having to add it to an existing kernel. Were we to rewrite the Hydra kernel, we would integrate the mechanism more thoroughly. We

believe the Suspect/Monitor model is effective in dealing with hardware failures and is consistent with the absence of centralized control we advocate for multiprocessor systems. We regret the need to build this specialized recovery mechanism, but regard the effort as a useful demonstration that reliability must be designed in, not added on.

A final anecdote illustrates that automatic recovery mechanisms can be "too good." Many users of Hydra are unaware of the system's ability to add processors to its running configuration. Occasionally, a processor excluded by the error recovery mechanism will pass all its diagnostics and be returned automatically to regular service. One unfortunate user was conducting a series of performance measurements that required precise control of the processor configuration. Unknown to him, Hydra automatically added a processor to the system during his experiments. It took him several days to figure out the anomaly in his data!

PART FIVE

MEASUREMENTS AND EVALUATION

CHAPTER

SIXTEEN

EXPERIMENTAL MEASUREMENTS

In this chapter we present the results of a number of experiments on C.mmp and Hydra. These results should help the reader to more accurately evaluate both the machine and its software. Whenever one measures a complex system, it is difficult to control all the variables; the measurements on Hydra/C.mmp are no exception. Thus, before beginning we would like the reader to keep the following points in mind:

The evolving nature of the system. The experiments reported here were performed over a period of several years during which the hardware and software evolved. Moreover, Hydra was designed to run with virtually any configuration of processors and memory and needed only a few I/O devices for the GST and its directory. For both these reasons, one cannot assume that any two experiments were performed on *exactly* the same configuration or the same software. We will try to explicate the differences where they matter.

Goals of the experiments. There are several reasons why one might perform measurements of *any* system:

To improve the performance of that specific system
To compare the system to others along one or more dimensions
To learn something about the way that the system is used in order optimize the design of future systems

Generally the experiments we shall describe started out with only one of these objectives. But, as when measuring any complex system, we sometimes learned more than we expected. A study into the decomposition of multiprocessor algorithms, for example, led to significant improvements of both the KMPS and PM scheduling algorithms. These kinds of changes contribute to the evolution of the system and thus exacerbate the problem of comparing and combining results.

Nature of the usage patterns. Hydra/C.mmp was and is an experimental system. It has never had a large user community and it would be difficult to predict with any confidence the nature of the load that the system would

experience if there were such a community. Since some of the experiments presume a load; their results must be interpreted relative to that load and *not* as "typical."

For all these reasons we will try first to simply describe each experiment in its own terms—without trying to relate its results to other experiments or trying to draw conclusions beyond the intended scope of the experiment. Then, a retrospective section will try to draw such additional conclusions and inferences as seem warranted by the collection of results.

The experiments we have chosen to describe are those that concentrate on the more novel aspects of C.mmp and Hydra—namely, the fact that C.mmp is a multiprocessor, the fact that Hydra runs in a distributed fashion (not master-slave), and the fact that Hydra is a capability-based system. Thus, we shall concentrate on such things as:

- The decomposition and performance of several algorithms, particularly as that performance reflects on design decisions in C.mmp and Hydra
- The effects of contention for resources at both the hardware and software levels
- The effects of the address space size and relocation structure on both programming and performance
- The size and speed of the kernel, particularly as they relate to supporting multiprocessing and the capability model
- The usage patterns of the GST, particularly as they relate to various implementation decisions in Hydra and the ways in which those decisions might be different in subsequent systems

Before beginning, however, we shall describe some of the tools that have been developed for measuring the system's performance.

16-1 PERFORMANCE MEASUREMENT TOOLS

For some experiments, of course, special measurement tools were developed. However, most experiments on Hydra/C.mmp used one or more of the following:

The Hardware Monitor. A device that permitted various hardware events (e.g., interrupt rates, memory references, etc.) to be monitored without disturbing the system.

The Kernel Tracer. A collection of software and microcode in Hydra that permitted a trace of certain software events (e.g., process blocking, I/O queueing, etc.) to be generated. A post-processor is used to present the trace output in a meaningful form.

The Snapshot Taker. A collection of software in the kernel that permitted a

large amount of internal state to be recorded at a specified time. A post-processor is used to present the snapshot information in a meaningful form.

Hercules. A user-level program that simulated an arbitrary number of user terminals, each executing a specified script. This program allowed us to place known and repeatable loads on the system.

Each of these tools has advantages and limitations that outline a useful domain of use.

16-1.1 The Hardware Monitor

The Hardware Monitor, or *K.mon*, is a device for detecting and recording certain hardware events on a PDP-11. It was initially designed and built for measuring the performance of C.mmp, but has been used for other PDP-11-based hardware systems as well. In this section we will cover only those general properties of K.mon that are relevant to the experiments described in this chapter. A more complete description can be found in [Ful73].

To understand K.mon at a sufficient level for the purposes of this chapter, one needs only to understand (1) a set of possible *event definitions*, and (2) a set of possible *event actions*; that is, the kinds of events that can be detected and the kinds of actions that can be taken in response to the occurrence of an event. Two factors affected the design of K.mon in this respect:

- The PDP-11 has the property that many, if not most, "interesting" hardware events are detectable as signals on the UNIBUS. Therefore, unlike the hardware monitors for other systems, K.mon event definitions and actions are defined primarily in terms of the information on this bus.
- While in principle one can simply record all possible events of interest and post-process the results to obtain those of real concern, in practice it is both more convenient and more efficient to provide for some "on-the-fly" processing power and to allow dynamic modification of the event definitions.

For both these reasons, K.mon was organized to connect to two UNIBUSs. To one of these, the "host" in Figure 16-1, K.mon is an essentially invisible, passive device. To the other UNIBUS, the "supervisor", K.mon appears to be a (rather sophisticated) I/O device. The host processor is the one being monitored. The supervisor processor controls the monitor; it is able to set the event definitions in K.mon, it provides the memory in which K.mon records data when events are observed, and it can be interrupted by K.mon whenever the event action so specifies.

There are five pairs of event definitions active simultaneously; each event

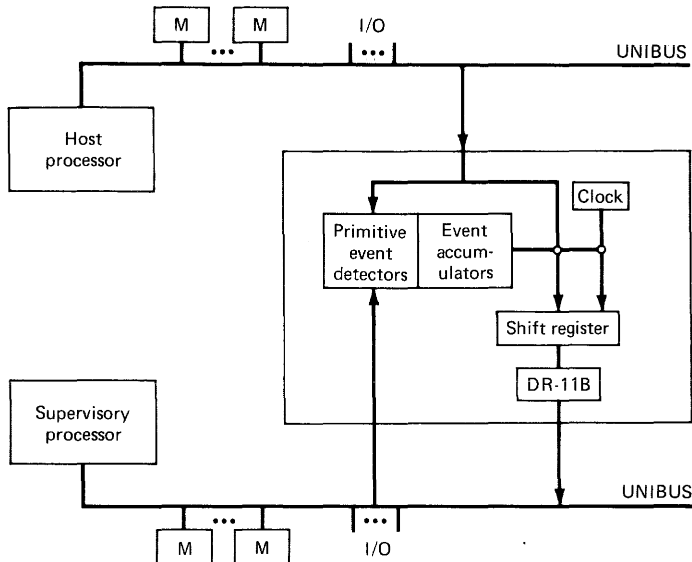

Figure 16-1 K.mon: the hardware monitor

definition consists of a primitive event specification and an accumulator; and the accumulator is initialized from a register associated with the event definition and is decremented each time the primitive event is detected. When this counter reaches zero the event is said to have happened; the counter is reinitialized and the action associated with the event is performed. (Obviously, by setting the initial value of the counter to one, one can cause the event to happen each time the primitive event is detected.)

A primitive event can be constructed by one or more of the following criteria:

- The address on the UNIBUS lies within a specified range.
- The data on the UNIBUS lies within a specified range.
- The UNIBUS is in a specified cycle (read, write, read-pause, etc.).
- The UNIBUS cycle is, or is not, an interrupt request.
- The two special "sequence bits" (explained below) have a particular value.
- Any of 16 high-impedance probes have specified values. (Typically, these probes are used for sampling signals from the processor, for example, to distinguish instruction-fetch from data-fetch cycles.)

Although there are only 5 event definitions in K.mon, there are 31 event action specifications—one such specification for each of the combinations of events that might happen simultaneously. Each event action can result in a number of operations:

- 0-9 words of data can be stored at a specified location in the supervisory processor's memory; this data includes the data and address from the UNIBUS, various other control signals, the accumulated event counters, the value of a high-resolution clock, etc.
- The supervisory processor can be interrupted.
- The two "sequence bits" mentioned above can be set, allowing primitive tests of the form "if event A follows event B."
- The high-resolution clock can be started or stopped.

A simple example should help to put all this in perspective and illustrate the power and flexibility of K.mon. Suppose that one wished to know how often a particular subroutine in the kernel blocked on a lock. To do this we would first connect a probe to the processor signal that distinguishes between the running and idle states. Then we would set up event definitions as follows:

First event. Use the address comparators to detect the execution of the first instruction of the subroutine. The action associated with this event will be to record the event counter and to set the "sequence bits" to a configuration reflecting that the subroutine is executing.

Second event. Use the address comparators to detect the execution of the last instruction of the subroutine. The associated action is simply to reset the sequence bits to some neutral state.

Third event. Use the address comparators to detect that instruction in the "lock" subroutine that is executed on failure to get the lock. If this instruction is executed and the sequence bits are set to indicate that the subroutine of interest is executing, record the event counter.

K.mon was used in all the memory and lock contention experiments presented in this chapter. It supplies the lowest-level measurements of any performance tool available to us.

16-1.2 The Kernel Tracer

The kernel tracer is a collection of software and microcode that permits one to record the occurrence of selected events within the kernel.[1] The time the event occurred, together with a small amount of additional information related to the event, is recorded for later processing. The collection of events that can be traced is defined by explicit code in the kernel. Calls to the tracing package are inserted at "interesting" points in the kernel source text; each call identifies the event and supplies the additional parameters that provide event-specific data. In order to define a new event, or modify the information associated with an existing event, the relevant portion of the

[1]There is also a special Kall that allows users to insert their own events into a trace.

kernel source program must be modified and recompiled.

Whether or not a particular event is *actually* recorded is determined dynamically. In an obvious way, toggles are associated with each event, and the appropriate toggle is tested before entering the tracing package.

Assuming that an event is enabled for tracing, the call on the tracing package will cause a record of the event to be written into a memory page specified by a user-level program. (We'll say more about this below.) The record consists of the following information:

A numerical event identification
The length of the record
The (physical) processor number on which the event occurred
The (KMPS-defined) process identification of the currently executing process on this processor
The current clock value (accurate to 4 μs)
The event-specific data supplied in the call on the tracer

Typically, event records are 10-20 bytes long, of which 8 bytes is the standard information.

Tracing is controlled from a user-level process. Three special Kalls exist for this purpose. The first specifies the events to be traced and supplies a list of pages (in the process' CPS) into which the data should be written. The second Kall blocks until a page is full of trace data and returns that page to the user, and the third Kall turns tracing off. Obviously the existence of the user-level process might perturb the system in some cases, and caution must be exercised to minimize this effect.[2] However, the added effort required to exercise this caution is more than offset by the flexibility provided. An experimenter may, for example, make a trace, process the resulting data, examine the results, and decide to alter and re-perform the experiment—all while executing in parallel with other users.

It is extremely important when defining a software tracer such as this one to keep its execution overheads as low as possible. There are two aspects of this: the overhead when tracing of a particular event is disabled, and the overhead when the same event is enabled. By keeping the former overhead low we can tolerate the existence of many trace definitions in the production version of the system; when errors or anomalous behavior is observed we can enable tracing as a diagnostic tool. By keeping the overhead for enabled events low we both minimize the perturbations introduced by tracing and permit a finer-grained analysis of behavior.

The tracer is implemented with a combination of in-line code, subroutines, and microcode. The enabling toggles, for example, are tested in-line so that the overhead in the event-disabled case is only 2 instructions. Microcode is used to actually make an event-record entry; the typical cost for this

[2]One can force the process to execute on only one processor, for example.

is 65 μs, or about 26 (non-microcode) instruction times.[3] Subroutines are used for relatively rare cases, such as when storage must be allocated.

Our experience indicates that this scheme makes it practical to trace events with a granularity of less than 1 ms with little perturbation.

Figures 16-2 and 16-3 show two common forms of post-processed trace output—process time lines and processor time lines. Each figure shows activity during approximately 1/4 sec of real time with a granularity of 2 ms. In Figure 16-2, each vertical column represents the execution of one process. Blank areas indicate periods during which the process is suspended; each period during which the process is executing begins with a line "CPU n," which indicates the processor executing the process. Other lines indicate whether the process was in user space ("USER") or what Kall was being executed.

Figure 16-3 is a processor time-line; each column represents the activity on a single processor. Blank areas indicate idle periods, "-CSW-" indicates a context swap, and the process being executed is identified by the line "USER n."

Many of the features of KMPS and the Message System are dramatically revealed in these traces, which follow the activity of the NCP handling an NVT—a "network virtual terminal." An NVT looks like a normal terminal to other subsystems, notably TMUX. Figure 16-4 shows how the relevant processes are connected in the Message System. Seven processes are shown; from left to right in Figure 16-2 they are a PM1 scheduling process, the NCP output multiplexor, the processor 12 idle job (to show IMP device interrupts), the NCP input multiplexor, the socket-pair process, the NVT process, TMUX, and the user's command language process. At time 0.910 we see the NVT starting after receiving a character from the ARPANET, causing it to (1) pass the character on to the CL, and (2) simultaneously echo it back over the ARPANET to the sender. At time 0.920 the NVT executes an *RSVPMsg* Kall to echo the character; the message is received by the socket process, which begins executing on processor 5 and continues on processor 13. The socket process in turn wakes up the output multiplexor at time 0.990, starting a series of communication between the output multiplexor and the IMP device (note the interrupt traffic at time 1.000). Meanwhile, the NVT has also replied the input data back to TMUX (the *WriteAndReply* Kall at time 0.935); however, TMUX has arranged to ignore normal replies of input, so the Message System routes the reply directly to the CL without waking up TMUX. The CL process, however, has been swapped out, so we see (at time 0.940) the PM process waking up and starting the CL, which begins executing at 0.975 on processor 6. This input causes the CL to send some output to TMUX at time 1.015, causing the PM process to start TMUX (which actually begins executing at time 1.06 on

[3]An earlier assembly-language coding of this operation required nearly 350 μs.

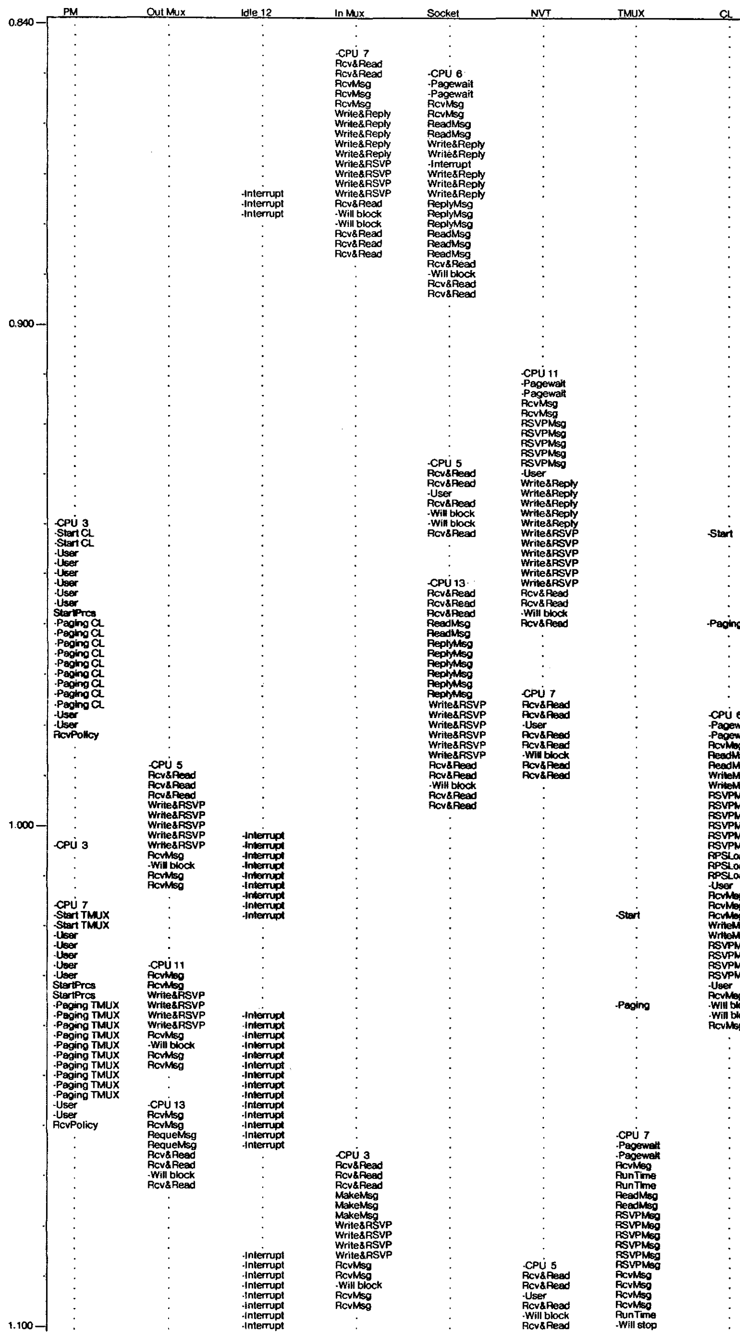

Figure 16-2 A process time-line

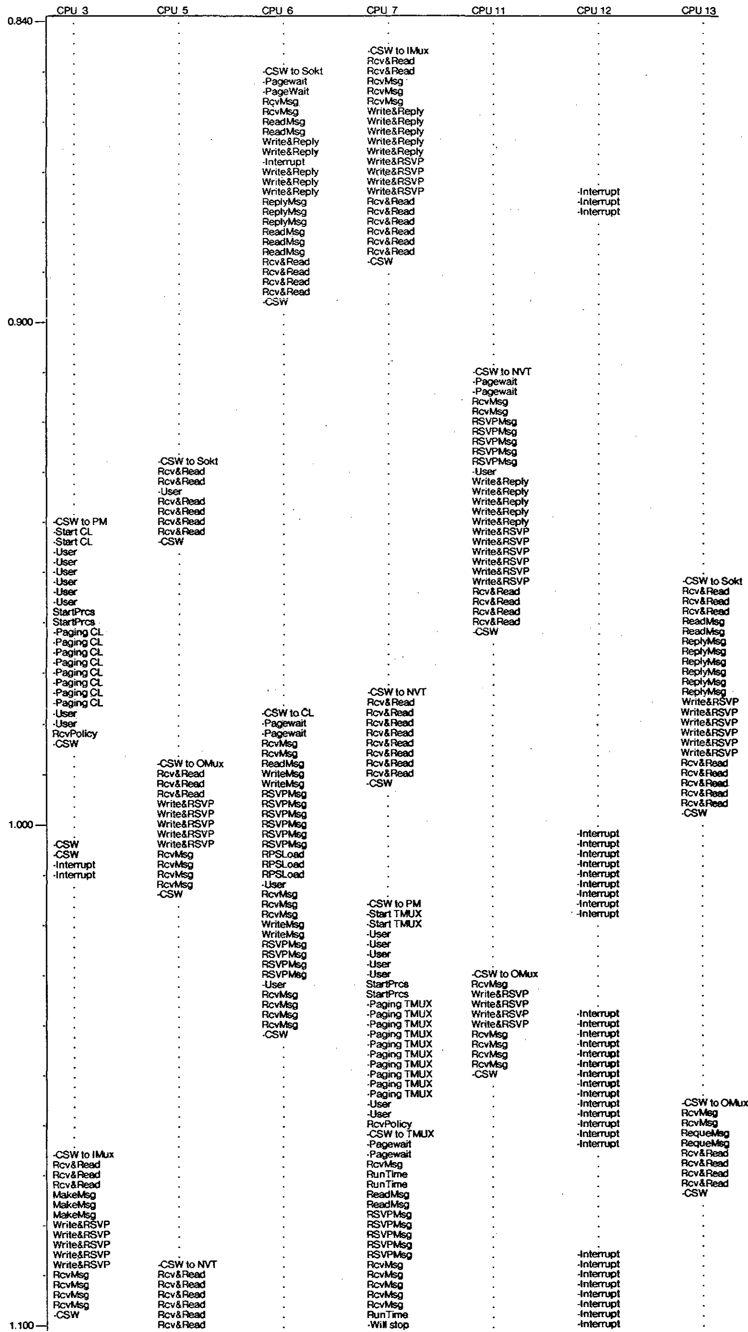

Figure 16-3 A processor time-line

processor 7). TMUX in turn forwards the output to the NVT, which is awakened at time 1.090.

Notice how the *WaitTime* scheduling parameter helps here: both TMUX and the CL had to be started by the PM, taking about 30 ms even though it is likely that no paging I/O had to take place. In contrast, the socket and NVT processes can be started immediately, since they have not been blocked longer than their *WaitTime* value.

Finally, notice the dynamic nature of the scheduling. Figure 16-2 shows how the socket process executes on three different processors (6, 5, and 13) within the 1/4 sec period traced. Conversely, Figure 16-3 shows how processor 7 executes, in turn, the input multiplexor, the NVT, the PM, and TMUX.

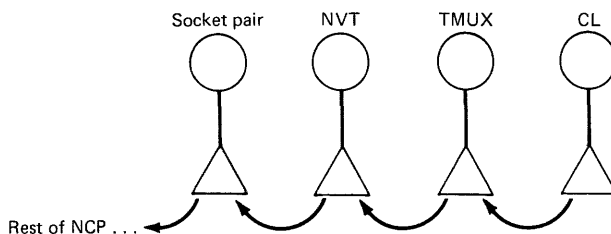

Figure 16-4 Communication structure for NVTs

The tracer is probably our most useful tool for software engineers. Many designers of user-level applications have used it to determine the performance bottlenecks in their systems. The process and processor time-lines provide valuable insights into the operation of the system—insights which would be hard to obtain in other ways.

16-1.3 The Snapshot Taker

The purpose of the Snapshot Taker is to permit a relatively large amount of state information to be recorded at one moment. Like the tracer, this mechanism is implemented within the kernel and is invoked by a special Kall from a user-level process. The parameters to this Kall specify the pages into which the data is to be delivered.

One can imagine two versions of a tool such as this: one would "freeze" the state of the system while recording that state; the other would not. Neither version is ideal for all circumstances. The first can cause massive perturbations in the system, making subsequent data questionable. The second can yield inconsistent data as the underlying structures are modified during measurements. As it happens, we were primarily interested in sampling, so only the second form (no freezing) was implemented for the experiments reported here.

16-1.4 Hercules: The Script Driver

Hercules is a "terminal emulator," or "script driver." It is intended to allow an experimenter to place a controlled load on the system so that various performance properties can be measured. It accomplishes this by emulating the activity of a number of users at terminals; the activity of each pseudo-user is controlled by a "script" supplied by the experimenter. A script may include:

- Commands to "type" input to the pseudo-terminal; this input may consist of both fixed strings and random numbers drawn from specifiable distributions.
- Commands to postpone further activity on the pseudo-terminal; delay can be for fixed intervals of time, for random intervals drawn from a specified distribution, or until a specified response has been received at the pseudo-terminal.
- Commands to generate trace data.
- Commands to repeat or (conditionally) skip portions of the script.

In addition, the actions listed above can be made conditional on the "terminal number." This facility permits several pseudo-users to be executing the same script and yet exhibit different behavior.

As an example, suppose one wished to measure response time to trivial tasks. A common design for this experiment involves placing the system under a load consisting of N people performing editing tasks and measuring the response as a function of N. A Hercules script can be constructed that will simulate N terminals; the initial part of the script might delay itself by a random amount of time based on the terminal number—this will avoid synchronous activity. The next portion of the script would contain the character strings to be "typed" to log into the system and create a file. The remainder of the script could be a loop containing a sequence of editing commands, delays for the response from the editor, and delays simulating the user's "think time." Trace output might simply consist of time stamps at the completion of "type-in" and at the beginning of the editor's response.

Hercules runs as a normal user process under Hydra, and thus it consumes resources and potentially perturbs the performance of other processes. This has not, however, been a problem. For many initial or exploratory studies, the perturbation is negligible. In those cases where final, reproducible results are desired, the KMPS scheduling parameters are set so that one processor is dedicated to Hercules alone. This reduces by one the number of processors effectively in the system, but eliminates perturbations other than from memory contention and I/O, both of which are small.

16-2 EXPERIMENTS AND RESULTS

Each of the following subsections describes an experiment and its results. Many of these experiments are parts of Ph.D. theses, and so we have included only the parts relevant to our purpose. We will try to convey the intuitions behind the experiment's design and its major results. More details may be found in the cited references.

16-2.1 Oleinick's Rootfinder Experiment

Relatively early in the Hydra/C.mmp project, Peter Oleinick [Ole78] began several experiments to obtain quantitative performance measures for parallel algorithms on multiprocessors. Rather than attempting to measure a spectrum of algorithms, he decided to focus on a small number of algorithms and investigate various implementation tradeoffs in depth for each of them. We shall describe the first of these experiments in this section.

To be suitable for Oleinick's purposes, the algorithms to be studied had to have two properties: they had to be complex enough to permit interesting implementation tradeoffs, and they had to be simple enough to permit attention to be focused on the implementation issues, not the algorithm *per se.* His choice was further restricted by the fact that asynchronous multiprocessor algorithms had not been studied in depth and not many were known. He finally settled on a simple extension of the binary search algorithm for finding the roots of a monotonically increasing function in a bounded region, an algorithm we call *RootFinder.*

The uniprocessor implementation of the binary search algorithm is well known. One simply divides the interval in half and evaluates the function at the midpoint. By comparing the sign at the midpoint with those at either end of the interval, one can determine which subinterval contains the root. From this point one has a smaller interval and simply repeats the process.

The obvious extension for multiprocessors is to divide the original interval into N + 1 subintervals and let one processor evaluate the function at each of the N interior points. For Oleinick's study the sub-intervals were chosen of equal size even though a different, optimal division was known [Kun76]. The function chosen was the normal integral, evaluated using a truncated power series in one region and a continued fraction in another. The details of these computations are not relevant to the present discussion, except to note that the time to evaluate the function is related to the argument value and has a known distribution.

Two implementations of RootFinder were used for the experiments. *RootFinder-1* stored its code in a single memory page which was shared among all processes. *RootFinder-N* provided separate code pages for each process. RootFinder-1 was thus much more subject to memory contention than RootFinder-N.

Much of Oleinick's work involved discovering why the observed performance of RootFinder did not match its expected performance. Under ideal circumstances we would expect RootFinder to generate a "pattern of performance" similar to that illustrated in Figure 16-5; each of the N processes (processors) completes at the same moment and after some brief bookkeeping operations by one of the processes, they all proceed on the next set of evaluations. Thus, we expect the overall time to find the root to decrease as the logarithm of the number of processes.[4]

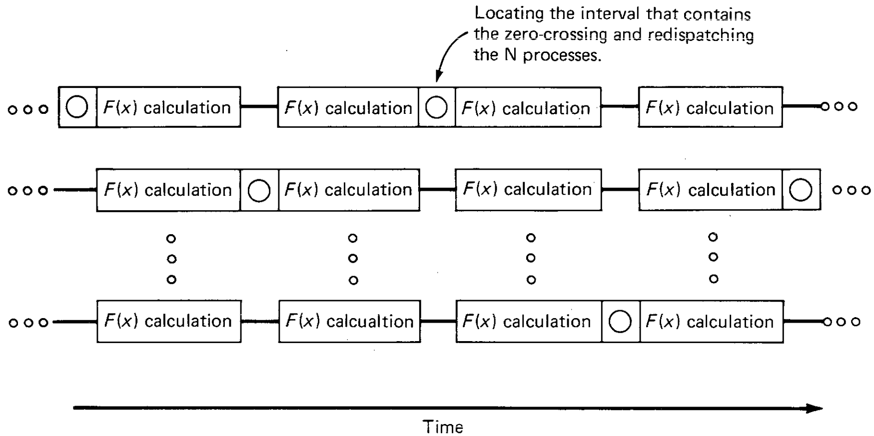

Figure 16-5 Expected RootFinder performance

Oleinick was able to identify three sources of perturbations which caused the expected performance not to be observed.

Variation in the time to compute F(x). Since the function being studied has the property that the time to evaluate F(x) is a function of x, the assumption that all processes would terminate at the same time was obviously false. Oleinick measured the actual compute time and found it to resemble a normal distribution with a mean of 100 ms and values ranging from 50 to 170 ms. The effect of this non-constant calculation time is to slow the entire assemblage of processes to the speed of its slowest member because all interior points must be evaluated before the next subinterval is chosen and the next cycle begins.

Variations due to technological factors. The expected performance can be obtained only if all processors and memories on the system are the same speed. In reality, this is not the case.

[4]We assume that the number of processes is no greater than the number of available processors. In this case, Hydra will, in effect, dedicate the processors to arbitrarily-chosen processes, and the process/processor distinction disappears.

Processor Differences. The C.mmp configuration contained both model 20 and model 40 PDP-11s when the experiments were run. The model 20 is 50-60% slower than the model 40. Although processes are preferentially scheduled onto PDP-11/40s, any experiment using more processes than there were 11/40 processors observed significant slowdown. Surprisingly, even within a single model of the PDP-11, significant differences were observed; there was about a 7% difference between the fastest and slowest 11/40 and an 8.3% difference between the fastest and slowest 11/20.

Memory Differences. The C.mmp configuration also contains both core and semiconductor primary memory; the speed difference between the two technologies is about 5%; additionally, within the memories of the same technology, speed differences of 2-3% were observed.

Memory Contention. The semiconductor memories are capable of delivering about 1.5 million references/second; the comparable number for the core memories is 1.7 million references/second. Either memory will saturate when three processors are repeatedly accessing it. This phenomenon was observed, as shown in Figure 16-6. We will further examine it in Sections 16-2.4 and 16-2.5.

The cumulative effect of all these factors resembles the effect produced by the variation in the time to evaluate F(x)—the total cycle time is limited by the speed of the slowest processing element.

Operating system performance variations. Although Hydra is a multi-user system, it is possible to reduce operating system overheads by running only a single application that makes minimal requests for services (i.e., does not do I/O or manipulate capabilities). Most of Oleinick's data was collected in this way. It is never possible to eliminate *all* performance perturbations introduced by Hydra. The major sources of these perturbations are:

The Kernel Tracer. Detailed analysis of the performance of the rootfinder was obtained with the aid of the kernel tracer. The use of the tracer lengthened some kernel operations—notably synchronization.[5]

I/O devices and interrupts. One cannot eliminate all I/O traffic and its associated interrupts. The occurrence of such interrupts can cause non-negligible perturbations.

Kernel Processes and Special Functions. Policy Modules, the GST demon, and the paging demon can all be awakened at unpredictable times and consume measurable processor resources.

Again, the effect of these variations is to slow the assemblage of

[5]Oleinick's data was collected using the version of the tracer that did not use any microcode, so the perturbations are worse than for later experiments.

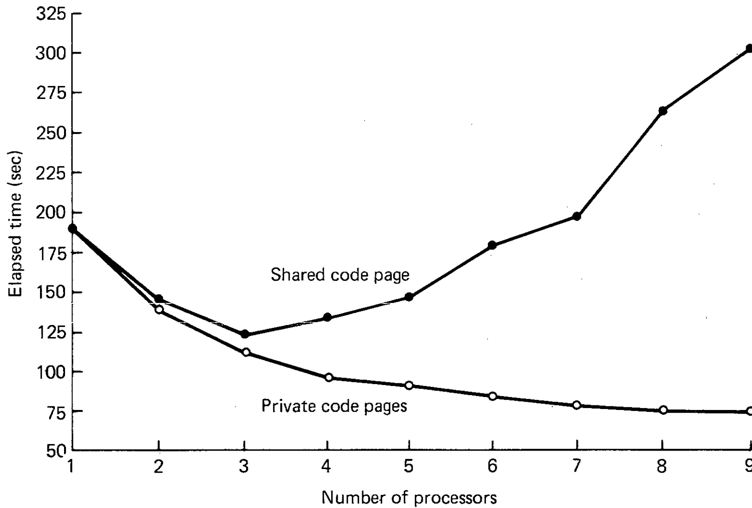

Figure 16-6 Performance degradation due to memory contention

processes to that of its slowest member.

The results of Oleinick's study of perturbation sources is summarized in Table 16-1. At the left is the maximum performance variation observed due to the factors listed at the right; the factors are listed in decreasing order of importance. Note that these are the *maximum* degradations observed before remedial action was taken; thus, for example, the 1:3 slowdown due to memory contention is eliminated by simply giving each process a private code page in a different memory unit.

Oleinick also used the RootFinder to study the effect of various synchronization mechanisms. Users cannot use the kernel "lock" mechanis , but they can program a "spin lock," a lock that does busy waiting. Also, kernel semaphores are not normally provided to user-level programs, but for the purposes of this study a special Kall was defined that gave access to this mechanism. Thus he was able to study the effect on the performance of RootFinder of spin locks, kernel semaphores and regular semaphores (POLICYSEMAPHORES). In addition, two Policy Modules were available while this experiment was run: PM0 and PM1. PM1 included some performance improvements as well as the ability to set the *WaitTime* scheduling parameter for a process. The results of his experiments are shown in Figures 16-7 and 16-8 (the value of *WaitTime* is denoted by "e" in these figures).

Figure 16-8 is especially interesting; it illustrates the effect of the cost of synchronization mechanisms on the total compute time as a function of the granularity of computation between synchronizations. The lower dotted line represents optimal performance. The upper dotted line represents half-optimal performance—that is, half the time is used for useful computing

Table 16-1 Factors affecting the performance of RootFinder

Magnitude	Cause
1:3.4	Variation in *F(x)* calculation (a property of the algorithm and the function under study rather than of the system).
1:3	Memory contention for shared code pages. (This effect is eliminated when each process is given a private page for its most commonly executed code. It would also be eliminated by the cache system that was never implemented.)
1:2.8	Bottleneck due to Policy Module scheduling. (This effect is eliminated when the wait time is set high enough—about 300 ms for Oleinick's experiment.)
1:1.6	Variation in processor models. Since performance degraded as soon as *any* 11/20 was used, it was better to run without them.)
1:1.3	Perturbations due to operating system factors such as interrupt processing.
1:1.1	Perturbations due to varying performance of primary memories and processors of the same model.

and half for synchronization. One could have chosen a "three-quarters" optimal, or any other line. Oleinick, however, chose the half-optimal line as a minimal level of acceptable performance. The points where the performance curves cross this line, then, characterize the minimal acceptable computation intervals between synchronization events for that mechanism.

This graph confirms and quantifies (for Hydra, anyway) the intuition that as the inter-synchronization interval becomes shorter it is increasingly important to have rapid synchronization primitives.

16-2.2 Baudet's Relaxation Experiment

Gerard Baudet's thesis [Bau78] is primarily concerned with the design and analysis of algorithms for asynchronous multiprocessors such as C.mmp. Actually, the bulk of the thesis is more relevant to numerical analysts than to an evaluation of Hydra/C.mmp, but he did perform one experiment on C.mmp that we shall report here.

The problem of concern is the solution of a partial differential equation, the Dirichlet problem for LaPlace's equation in a rectangular domain. Using the method of finite differences, the problem can be reduced to the solution of a set of linear equations, $Ax = B$. In turn, this system can be solved by iterative methods such as that of Jacobi. Baudet considered a number of ways in which these iterative methods can be turned into parallel algorithms. In each of the methods he studied, the solution vector is divided into k partitions, and a separate process is assigned the responsibility for computing

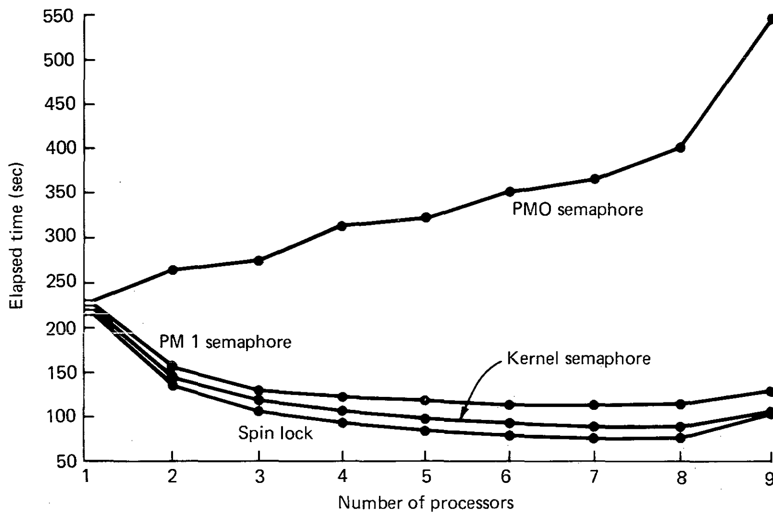

Figure 16-7 RootFinder using different synchronization primitives

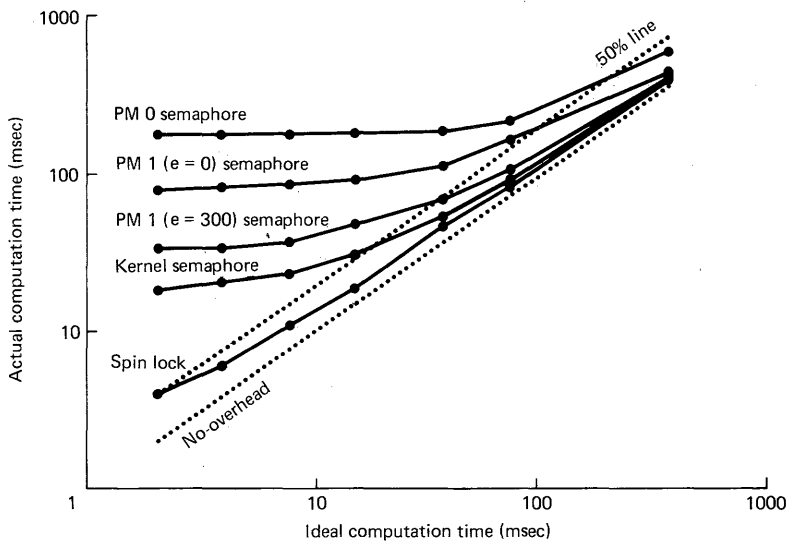

Figure 16-8 Effect of inter-synchronization interval

the "next iterate" values for the components of its partition. He used the following methods, which only differ in the choice of the values used for the "previous" iterates:

Asynchronous Jacobi's Method (AJ). This method consists of repeating a cycle

in which the value of the new iterate is computed using only the values of the iterates computed on the previous cycle. This scheme involves synchronizing all processes at the end of each such cycle.

Asynchronous Gauss-Sidel Method (AGS). This method also involves a cycle as in AJ, but each process is allowed to use the new iterate values from its own partition in computing subsequent new iterates. Iterate values from other partitions are taken from the previous iteration. Like AJ, this scheme also involves synchronization at the end of the cycle.

Purely Asynchronous Method (PA). This method does not involve a major cycle; each process is allowed to use the most recent value of each iterate in the entire vector, x. Since there is no major cycle, there is also no synchronization.

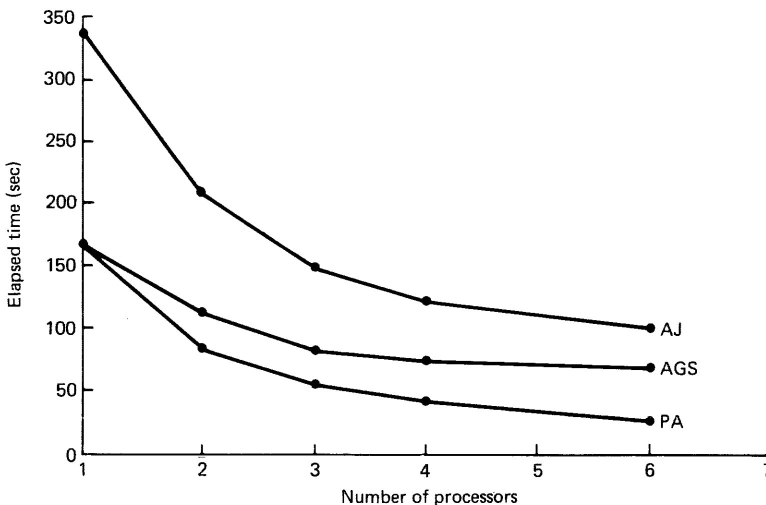

Figure 16-9 Performance of algorithms for the Dirichlet problem

Baudet's thesis gives conditions under which these algorithms converge and an analysis of the expected performance. Here we will report only the experimental results, which consisted of solving Dirichlet's problem for a rectangular grid with 21 x 24 points; this results in a linear system with $n = 504$. Convergence was assumed when the initial error had been reduced by a factor of ten. At the time this experiment was performed, C.mmp had only six processors, so the results displayed in Figure 16-9 are given for $k = 1, 2, 3, 4, 6$.

The results in Figure 16-9 would, of course be different for different problems; they can only be construed as an example of the behavior of asynchronous algorithms. Also, it should be remembered that two effects are

combined in these results: that of using more recent iterate values and that of synchronization. The difference between AJ and AGS is purely the first of these—AGS uses the most recent iterates from its own partition. The differences between PA and the other two are a combination of both. Notice, by the way, that PA achieves essentially linear speed-up in this case—the best that can be hoped for.

16-2.3 Oleinick's HARPY Experiment

The previous two sections are concerned with the performance of single algorithms; in this section we discuss the results of an experiment on the performance of a much larger system: HARPY.

HARPY [Low77] is a system that recognizes phrases and sentences of connected speech. The details of the speech recognition task itself are not important to us here, so we will give only a brief description of the system. HARPY represents its knowledge of both speech and the task domain as a weighted, directed graph. Each node in the graph represents a phoneme and the weighted arcs represent legal transitions from one phoneme to another, with the weight representing the probability of that particular transition. A preprocessor constructs the graph, assigning weights based on both the legality of particular transitions (as derived, for example, from the syntax rules of English) and the likelihood of particular utterances in the task domain.

Given the preprocessed graph and a representation of an utterance, HARPY applies a *beam search* to determine the most likely interpretation of that utterance. That is, it searches several of the most likely paths through the graph simultaneously, keeping track of the one with the highest probability. When the terminal nodes of the graph are reached—the most likely path is accepted as the interpretation of the utterance.

The parallel implementation of the HARPY system underwent a series of refinements [Ole78]. The final version consisted of a set of processes, each of which could either find a successor node or compute the probability of reaching the state represented by the (partial) path leading to that node. The performance of this system on two tasks is shown in Figures 16-10 and 16-11. The two graphs exhibit quite different performances; the primary cause of this difference is the amount of work to be done. The DESCAL task (Figure 16-10) is that of a speech-activated desk calculator and has roughly a 30-word vocabulary. The 1,000-word task has information retrieval as its task domain. Utterances such as "Please help me," "Who was the author?," "When was it published?," and "What about program verification?" were part of the 1,000-word task.

Several points should be noted from these graphs:

1. The single-process version of HARPY on the DESCAL task is comparable to the uniprocessor versions on both a PDP-10 (KA10) running the

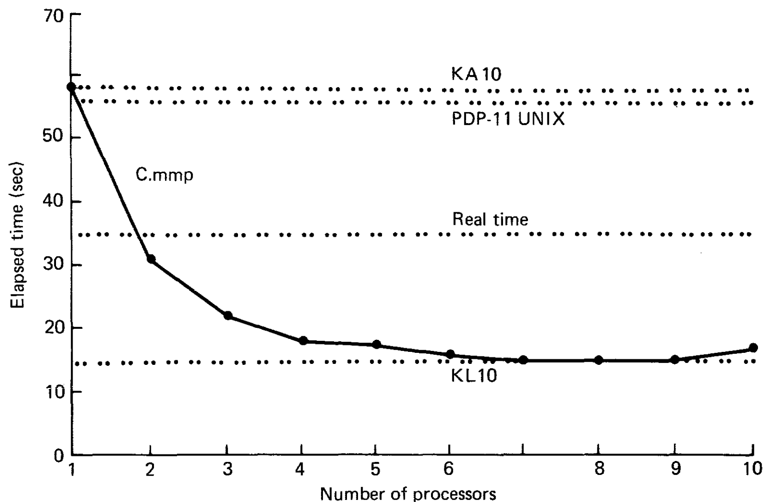

Figure 16-10 HARPY performance on DESCAL task

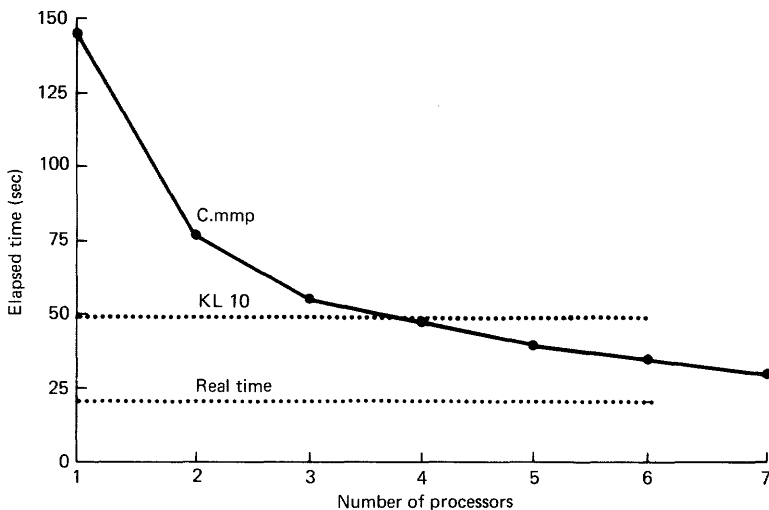

Figure 16-11 HARPY performance on 1,000-word task

TOPS-10 operating system and a PDP-11/40 running the UNIX operating system. Similarly, the ratio between the single-process C.mmp version and the PDP-10 (KL10) version is about 1:3, better than the ratio of the raw machine speeds which is about 1:5. It appears as though neither the decomposition into processes nor Hydra's support of multiprocess computations have affected performance significantly.

2. The C.mmp version outperforms the KL10 version on the 1,000-word task after three processes are used; this is the best that could have been expected given the speed ratio for the single-process versions. The C.mmp version approaches and finally equals the KL10 version on the DESCAL task, but never does better because there isn't enough work to keep the processors busy. In this case, communication and synchronization (needed to determine that there is nothing to do) dominate the processing times.
3. Performance actually worsens at 10 processors on the DESCAL task; this is where the first PDP-11/20 processor is used. As with RootFinder, the synchronous nature of HARPY makes it undesirable to use processors of different speeds.

16-2.4 Marathe's Memory Interference Experiment

Madhav Marathe [Mar77] studied the effect of memory contention on C.mmp in some of the first experiments to use the Hardware Monitor.

Prior to the design and construction of C.mmp we were extremely conscious of the potential for serious performance degradation as the result of contention for access to the shared primary memory. Some manufacturers had estimated a 10% degradation for each additional processor (i.e., a 40% degradation for a 4 processor system); other manufacturers had experienced a 50-70% degradation for the second processor. Our intent was to design C.mmp so that contention would not be a serious problem; the basis for the design was a set of analytic models of the contention phenomenon such as those by Strecker [Str70], Bhandakar [Bha73], and McCredie [McC73]. The design assumed PDP-11/20 processors (the only model of the PDP-11 available at the time). Marathe's experiment was designed to determine the effectiveness of the design decisions as well as to determine the effect of using the (faster) 11/40 processors.

The use of K.mon constrained the experiment somewhat since K.mon is capable of monitoring only one UNIBUS, whereas it would be preferable to measure the total switch traffic. However, we can obtain an upper bound on the effect of contention. In the presence of contention the processors are serviced in priority order (see Chapter 2), so K.mon was set up to monitor the lowest priority processor. This processor will experience the worst degradation.

To measure the effect of contention, K.mon was provided with 6 one-shot flip-flops that change their state after a specified interval. The intervals chosen were 0.5, 1, 2, 4, 14, and 50 μs, respectively. By reseting these flip-flops at the beginning of a memory request and examining them at the end of the request, the duration of the request can be classified into one of the six intervals. By accumulating the number of cycles in each interval, a histogram of cycle lengths is constructed. Both to avoid systematic errors and

because of some limitations of K.mon, the histograms were accumulated by looking at a burst of about 160 cycles, delaying for a random period, accumulating another burst, and so on. In all, 100,000 cycles were tallied in each experiment.

Experiments were conducted on three workloads:

Idle. No user processes were executing. Except for clock interrupts and occasional demon activity, nothing was running. This load is as light as one can imagine; one cannot expect contention to be less. Thus this value can be used for comparison with the other two.

RootFinder-N. This load consisted of 16 processes executing Oleinick's RootFinder program discussed previously. Each process had a separate code page. Access to the common data is minimal in this algorithm. Since the processes are primarily processor-bound, this workload is expected to produce contention similar to a number of independent users executing different programs, but making heavy use of the processors.

RootFinder-1. This load also consists of 16 processes executing Oleinick's RootFinder program. However, all processes share a common code page. Since about 70% of the memory references generated by a PDP-11 are to the code page, this load makes heavy demands on access to a single memory port. A large amount of contention is to be expected in this case; indeed, we would not expect contention to ever be this bad in practice.

The results of the experiment for these loads is shown in Table 16-2.

Table 16-2 Memory cycle length under load

	Number of cycles		
Cycle length	Idle	RootFinder-N	RootFinder-1
0.0-0.5 μs	0	0	0
0.5-1.0	88,439	85,134	69,453
1.0-2.0	11,404	13,876	11,601
2.0-5.0	71	958	3,344
5.0-14.0	79	31	15,421
14.0-50.0	7	1	181
Above 50	0	0	0
Average cycle length	0.85	0.88	2.34

We think this is one of the most encouraging results for C.mmp. As can be seen, contention in RootFinder-N causes less than a 5% degradation; the

"worst case," RootFinder-1, causes degradation by a factor of almost three. The clear implication from this experiment is that, although the user should be cautious about sharing code pages, in most applications memory contention is simply not a factor. Although RootFinder-N in no way simulates a timesharing system, we expect independent users to share code pages much less frequently than RootFinder-1.[6] Furthermore, even shared code pages would not be a problem if C.mmp's caches were implemented.

16-2.5 McGehearty's Memory Contention Experiment

Patrick McGehearty [McG80] took a somewhat different approach to measuring the memory contention in C.mmp. Since contention can be influenced by the mix of instructions executed, McGehearty's experiments were run using a synthetic program whose mix of instructions was picked to match that measured on a large variety of programs on C.mmp [Mar77]. This mix was run on the bare hardware so that there were no perturbations due to the operating system, I/O traffic, timer interrupts, etc. All measurements used the system clock to determine the time to complete a specified number of iterations of the synthetic program.

A number of interesting results are reported in [McG80], but we shall consider only two: the total contention, and the effect of the contention resolution scheme used in C.mmp's crosspoint switch. For both of these, the machine configuration consisted of 10 PDP-11/40 processors and 16 memory units. [Of the 16 memories, 5 contained 32 pages of semiconductor memory (each), 10 contained 16 pages of core memory (each), and 1 had 6 pages of core memory.]

Figure 16-12 displays the total contention effect in terms of the incremental processing power obtained from the *i*th processor. Two cases are considered: (1) all memory *units* are equally likely to be accessed, and (2) all memory *pages* are equally likely to be accessed. Since some ports contain more pages than others, they are more likely to be accessed in the second case, and thus more contention will occur. For example, under the assumption that all pages are equally likely to be accessed, the 10th processor will deliver about 85% of its potential processing power. This corresponds to an average effective power of about 93%—only a 7% degradation due to contention.

While conducting the contention experiments, McGehearty noticed that different processors received drastically different service. We discovered the cause in the priority resolution circuitry in the switch. Recall from Chapter 2 that when several processors make simultaneous requests for a port, those requests are latched into an internal register in the switch. All requests in

[6]In a timesharing environment, sharing of code belonging to an editor or compiler is usually encouraged, but experience with other systems suggests that this is far less of a factor than is usually presumed.

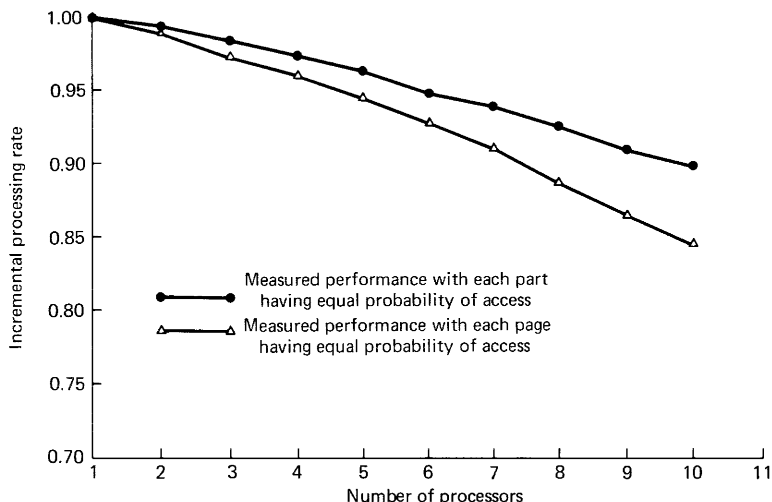

Figure 16-12 Incremental processing power with memory contention

this register are serviced before any other requests are allowed into the register. Moreover, the requests in the register are serviced in priority order, with the highest numbered processor having highest priority. The intent of this scheme is to prevent a processor from being starved, and indeed, to provide fair service to all processors.

At first glance it may appear that the design satisfies these criteria. In reality it does not quite do so. Consider the case of three processors repeatedly accessing the same memory. Initially all three requests will be latched into the register and will be serviced. On the second cycle, however, only the two higher-numbered processors will have requests latched into the register; the third processor will have just completed a memory reference and will not have had time to insert its next request. On the next cycle, the middle processor will probably not have its next request latched—again, because it had just completed a memory reference and will not yet have had the chance to make another. In subsequent cycles the lowest-numbered processors will alternate being excluded while the highest numbered processor always gets service.

Figure 16-13 shows the effect of this behavior as the number of competing processors is increased. The bottom heavy line shows the execution speeds of the processors alone; the line is not perfectly horizontal because the processors have slightly different speeds. The benchmark was then run with different numbers of processors; first with processors Nos. 15 and 14, then with Nos. 15, 14, and 12, etc. The curves are normalized so that processor No. 15's time is 1.0; they show that as the number of processors increase, the performance of the lower-numbered (lower switch priority) processors is

worse than the higher-numbered processors. In the final 10-processor run, processor No. 0 is experiencing about a factor of three worse degradation than processor No. 15.

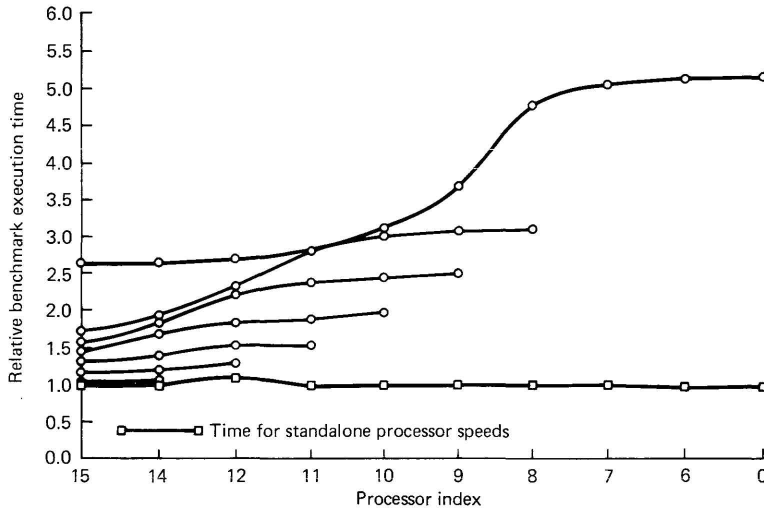

Figure 16-13 Starvation due to switch contention resolution strategy

There is little evidence that this effect is significant for production programs on C.mmp. It does, however, provide additional motivation for either avoiding heavily shared code pages or implementing the cache scheme.

16-2.6 Marathe's Lock Contention Experiment

Another potential source of serious performance degradation is contention for software resources. In particular, since Hydra is a distributed system in which all processors could in principle be executing kernel code simultaneously, there is a potential for long delays due to such contention. The designers were acutely aware of this potential performance problem, and consideration of it lead to two major decisions:

1. Lock data items rather than code, and have a large number of locks (i.e., protect data in smaller rather than larger chunks).
2. Provide synchronization mechanisms with different resource consumption characteristics: locks, Kernel Semaphores, ports, and Policy Semaphores.

Marathe [Mar77] studied contention for kernel locks; Jain (see Section 16-2.7) later studied the other two synchronization mechanisms. Marathe's experiments consisted simply of using the hardware monitor to detect entry

to the lock subroutine's code (noting the address of the lock cell), to detect entry to the code that is executed when blocking occurs, to time the interval between lock requests, and to time the interval between a block and corresponding wake-up.

The experiments were performed on four workloads: two versions of RootFinder, a synthetic load that exercises most of the kernel operations, and a general multi-user session. The two versions of RootFinder were: RootFinder-N(K) which used kernel semaphores for synchronization, and RootFinder-N(S) which used the regular user-level semaphores. In all cases but the multi-user session, there were sufficient processes to keep the processors busy. Unfortunately, only some of the data is available for the multi-user session.

The first interesting result of Marathe's experiments is shown in Table 16-3. There are literally thousands of lock cells in Hydra;[7] however, some are used a great deal more than others. As shown in the table, the most frequently used locks are:

Processor list lock. This lock protects the "processors" list, the list of instantaneous processor-process bindings. KMPS maintains this list for scheduling purposes.

Feasible list locks. When these experiments were run, the PM assigned a uniform priority, namely, zero, to most tasks and highest priority, namely, 255, to others. Thus, only two feasible lists ever contained processes.[8]

Page-pool locks. This is the set of locks in the storage allocator that are used to mutually exclude simultaneous allocations of small buffers from the same page. There are typically from 20 to 60 locks (pages) in this set.

Free core lock. This lock is used to protect the list of available page frames.

Stop mailbox lock. This lock is used to protect the mailbox through which the kernel passes information about stopped processes to a Policy Module.

KMPS space lock. This lock is similar to each of the page-pool locks, except that it is used for a special storage pool used by KMPS.

All other locks are listed as "miscellaneous," although they collectively represent a significant number of the locks used; the use of no one of them is significant in isolation.

A second set of interesting results from this experiment concerned the time spent in the various critical sections (Table 16-4). The average instruction time during these measurements was 2.8 μs; the data indicates

[7]There are so many lock cells *because* data structures, rather than code segments, are locked.

[8]In Chapter 12, the list of feasible processes was treated as a single, logical entity. The implementation actually subdivided it into an ordered set of eight sublists to increase potential parallel access.

Table 16-3 Kernel lock use in four applications

	Application			
Lock	RootFinder-N(S)	RootFinder-N(K)	Synthetic	Multi-user
Processor list	15.8%	30.1%	11.5%	34.2%
Feasible 1	11.8	28.3	10.5	6.0
Feasible 8	3.4	0.6	0.0	0.3
Page-pool locks	17.2	0.0	39.4	23.4
Free core lock	4.6	0.0	5.4	6.1
Stop mailbox lock	0.9	0.0	0.0	0.2
KMPS space lock	9.3	0.0	0.1	0.0
Miscellaneous	29.6	41.1	25.2	23.1

Table 16-4 Average time within kernel critical sections

	Application		
Lock	RootFinder-N(S)	RootFinder-N(K)	Synthetic
Processor list	348 μs	409 μs	379 μs
Feasible 1	192	239	260
Feasible 8	156	169	
Lock on a page	338		431
Free core lock	558	307	685
Stop mailbox lock	282	264	297
KMPS space lock	109	123	134
Miscellaneous	318	461	441
Average	279	378	279

that from 40 to 240 instructions are executed in critical sections.

Table 16-5 is a summary of some of the run-dependent data for these experiments as well as the "bottom line" information on the contention for locks.

We believe this data is very significant. Processor synchronization causes *less than 1% degradation* in a 16-processor system. Furthermore, Marathe's theoretical model of Hydra-like lock contention, validated against the actual Hydra data, predicts only a 1.7% performance degradation at 48 processors. As a practical matter, even though Marathe's data isolated several locks which had a *relatively* high level of contention, the low *absolute* level did not justify our spending time on obvious remedies.

16-2.7 Jain's Semaphore and Port Experiment

From Marathe's data we can draw two inferences, namely, that contention for locks is not a serious problem and that the critical sections that are guarded

Table 16-5 Summary of lock contention data

Lock	Application		
	RootFinder-N(S)	RootFinder-N(K)	Synthetic
Active processors	13	14	12
Test length	17.4 sec	32.9 sec	20.3 sec
Lock operations	2,955	5,041	4,360
Times blocked	130	577	146
Time between locks	5,888 μs	6,531 μs	4,646 μs
Percent of locks that blocked	5.5%	11.7%	6.1%
Percent of time in blocked state	0.29%	0.83%	0.74%

by locks are not generally large. However, Hydra uses another synchronization mechanism, Kernel Semaphores, and provides two others to users, ports and Policy Semaphores. Navindra Jain [Jai79] undertook to analyze the use of these mechanisms. We will separately describe his results for Kernel Semaphores and for Policy Semaphores (and ports). For each of these we will examine the cost of executing the mechanism when a process is blocked, the frequency and duration of blocking, the duration of critical regions, and the duration of computational intervals between critical sections. From this data we can derive the fraction of processing power lost due to blocking.

Performance of Kernel Semaphores A Kernel Semaphore is intended for use only within the kernel. When blocking, the processor is released but the memory associated with the user-process is not. Thus, the intended function of these semaphores is to protect those critical regions whose duration is expected to be significantly longer than the context-swap time, but shorter than the time to swap processes out of primary memory.

The first data collected by Jain was on the overheads associated with executing the P and V operations. The coding of the operations has been optimized so that the costs in the event that a non-blocking P, or a V that does not wake up a sleeping process, are negligible (about 15 instructions). However, the other cases are shown in the following table.

Operation	Overhead
P that blocks	2.1 ms (about 750 instr)
V that wakes a process	1.0 ms (about 360 instr)

The cost of a P operation is measured from the time at which the P operation is invoked until the next user process begins running. Thus it includes KMPS scheduling and context-swap time as well as the code in the P

itself.[9] The total CPU effort expended by blocking on a semaphore is the sum of the two components, or 3.1 ms.

The second aspect of Jain's experiment involved collecting data during normal multi-user sessions using the kernel tracer. Due to the amount of data gathered in this way, it was not possible to enable tracing for long periods. Instead, shorter (1-2 minute) samples were collected at random periods.

Jain went on to study the distribution of blocked intervals during multi-user sessions. He used the kernel tracer and collected short samples (1-2 minutes) at random intervals. Table 16-6 illustrates the typical behavior observed.

Table 16-6 Kernel semaphores: blocked periods

Interval range	Number	%
0.0-0.5 ms	240	11.6
0.5-1.0	247	11.9
1.0-2.0	129	6.2
2.0-3.0	54	2.6
3.0-4.0	51	2.5
4.0-5.0	59	2.9
5.0-10.0	280	13.6
10.0-25.0	286	13.9
25.0-50.0	39	1.9
50.0-75.0	26	1.3
75.0-100.0	19	0.9
100.0-250.0	67	7.5
250.0-500.0	154	7.5
500.0-∞	415	20.1

Blocked interval:	
Mean:	240 ms
Median:	10 ms (approx.)
Frequency of blocking:	20 blocks/second/processor
Total samples:	2,066
Fraction of processor lost:	0.61%

Several things should be noted from Table 16-6. First, there are a significant number of cases, over 30%, in which the blocked period was less than 3 ms. If it were possible to identify which semaphores these were—and if those semaphores consistently blocked for less than 3 ms, it would actually be cheaper to use locks. Second, there are a large number of cases with extremely long blocked durations. Kernel semaphores were not really

[9]The scheduling and context-swap times are about equal; the actual time in *P* is much smaller.

designed for this case. Again, it might be possible to possible to exploit another mechanism if these cases could be identified. Finally, partly because of the non-linear scale, the distribution appears trimodal—which suggests that there might actually be three or more separate cases contributing to the total behavior.

This last conjecture was in fact found to be true. We won't reproduce all the data here, but Jain found that the semaphores could be classified into five groups:

Kernel tables. No case of blocking was observed. We do not know whether it would be better to use locks, but the performance difference would be slight.

GST mutual exclusion semaphores. These semaphores are in the fixed-part of every object and are used to prevent simultaneous access by the kernel while manipulating the representation of objects. These contribute almost all the shorter blocked periods. They appear to be mostly uniformly distributed between 0 and 10 ms, with a few cases reaching above 30 ms. The average is about 4.3 ms.

Page semaphores. The semaphores are part of the representation of PAGE objects and are used to suspend processes waiting for paging operations. They have a sharp peak around 16 ms (the drum rotation time) and a few very long blocked periods that raise the average to 38 ms. (These long blocked periods involve multiple I/O operations.)

GST I/O semaphores. These semaphores are used during the transfer of objects between the Active and Passive GST; they prevent access to the object during the requisite disk or drum I/O. Blocked periods range from 5 to over 100 ms, with an average of 31 ms.

Sleep semaphores. These semaphores are used by demon processes in the kernel; all blocked periods for these exceeded 25 ms and they contribute the bulk of those in excess of 100 ms. Their average blocked period was 312 ms.

This finer analysis does not suggest that a change in the current use of locks and semaphores would substantially affect performance. The longest blocked periods, the sleep semaphores, are for demons that run in the kernel address space; there are no additional resources that could be released. The next longest blocked periods are during I/O for the GST and the Paging System; in both cases releasing the process' pages would only precipitate additional I/O. Note, however, the frequency of GST Mutex Semaphore blocking in the 1-10 ms range makes one wonder whether there is another mechanism with cost intermediate between locks and Kernel Semaphores that could be used here.

Tables 16-7 and 16-8 illustrate the distribution of the length of critical sections protected by Kernel Semaphores and the intervals between them. As can be seen, the typical interval is short. Although the average duration

shown on Table 16-7 is 27 ms, much of this is due to a special case where the duration is over 2,500 ms. Eliminating this case brings the average down to 4.15 ms.

Table 16-7 Kernel semaphores: time in critical sections

Critical section duration	Number	%
0.0-0.5 ms	20,236	42.9
0.5-1.0	8,329	17.6
1.0-2.0	7,489	15.8
2.0-3.0	2,463	5.2
3.0-4.0	2,029	4.4
4.0-5.0	1,483	3.1
5.0-10.0	1,898	4.0
10.0-25.0	812	1.7
25.0-50.0	815	1.7
50.0-75.0	601	1.2
75.0-100.0	232	0.4
100.0-250.0	254	0.5
250.0-500.0	34	0.0
500.0-∞	405	0.8

Time in critical section	
Minimum:	0.04 ms
Mean:	27 ms
Median:	1 ms (about 360 instr)
Maximum:	5,305 ms
Total samples:	47,143

Jain's results confirmed our expectations that time lost to synchronization is negligible; the time lost never exceeded 1%. This means that total contention in the kernel due to synchronization (locks and semaphores) *does not exceed 1.7%, or 1/4 of one processor.* We emphasize these results because they are contrary to the intuition of many system builders.

Performance of ports and Policy Semaphores Ports and Policy Semaphores were intended as the primary means of communication and synchronization among user-level processes. In the event of blocking, a context-swap is performed just as in the case of kernel semaphores. In addition, however, after a period specified by the policy module (*WaitTime*) the blocked process' pages are made eligible for swapping.

As with kernel semaphores, Jain first measured the cost of these operations. For our purposes, namely, an analysis of blocking costs, ports and

Table 16-8 Kernel semaphores: intervals between critical sections

Interval between critical sections	No.	%
0.0-0.5 ms	11,360	34.6
0.5-1.0	5,895	17.9
1.0-2.0	4,124	12.5
2.0-3.0	3,902	11.8
3.0-4.0	2,474	7.5
4.0-5.0	2,128	6.4
5.0-10.0	1,073	3.2
10.0-25.0	492	1.4
25.0-50.0	304	0.9
50.0-75.0	112	0.3
75.0-100.0	98	0.2
100.0-250.0	216	0.6
250.0-500.0	337	1.0
500.0-∞	334	1.0

Intervals between critical sections	
Minimum:	0.13 ms
Mean:	19 ms
Median:	1 ms
Maximum:	2,827 ms
Total samples:	32,790

Policy Semaphores behave identically—thus we will consider only the latter.[10] The cost of a *P* operation on a Policy Semaphore that blocks (as opposed to one that does not block) is 8.7 ms (about 3,100 instructions). As with kernel semaphores, this number includes everything from the time at which the *P* operation is invoked until the next process begins to run. Thus, in particular, it includes the scheduling and context-swap costs incurred in kernel semaphores. The cost to reawaken a process with a *V* operation is 6.5 ms (about 2,300 instructions).

There is an additional cost that is incurred between blocking and reawakening a process, which depends upon whether the blocked period exceeds the *WaitTime* parameter. If B is the blocked period and W is the value of *WaitTime*, then there are two cases:

$B \leqslant W$. There is an additional cost of 2.4 ms to cancel the request to the time-out mechanism.

$B > W$. In this case the kernel must inform the PM that the process has been stopped and make its pages eligible for swapping. Later, when the *V* operation is performed, the kernel must inform the PM that it is

[10]Obviously, the *ReceiveMsg* Kall for ports has some additional cost over a *P* for semaphores. Messages must be dequeued and passed to the user, for instance.

possible to restart the process. At both times, the PM must also execute. Also, at both times the kernel cost is a function of the number of pages in the process' CPS. Finally, there is a cost associated with actually moving the pages to and from the drum. If

n is the number of pages in the process' CPS
t_{pm} is the Policy Module time
t_{page} is the time used to actually move pages

then this cost is

$$6.5 + 1.8n + t_{pm} + t_{page} \text{ ms}$$

Thus, the total cost for executing a P operation that blocks is:

Condition	Total cost
$B \leq W$	17.6 ms
$B > W$	$21.7 + 1.8n + t_{pm} + t_{page}$ ms

At the time Jain made his measurements, PM1 was used exclusively. For PM1, the value of t_{pm} is roughly 66.2 ms. Jain also measured t_{page}; it requires 13.8 ms of CPU time and 48 ms of disk time per page swapped. The data for t_{page} actually involves three transfers for each page—one to initially write it out, one to perform a read-check before the core page frame is released, and one to read it back in when needed. Unfortunately, the actual number of pages transferred cannot be related to n since actual paging is performed by a demon process whose policy is related to total system load rather than to any property of the individual process. Ignoring t_{page}, the costs for Hydra together with PM1 are[11]:

Condition	Cost
$B \leq W$	17.6 ms
$B > W$	$87.9 + 1.8n$ ms

It is not meaningful to ask quite the same questions about ports and Policy Semaphores as about Kernel Semaphores. In particular, while the kernel might have implemented them more efficiently or might have provided a different mechanism, it cannot control their use. Thus the total time spent executing these operations, and particularly the time spent blocking and awakening processes, will be variable with the tasks executing at a given moment.

[11]This analysis is slightly different from Jain's due to an apparent arithmetic error on his part. The essential conclusions are identical, however, and our analysis appears to agree closely with the data from an independent experiment by McGehearty.

Jain did measure blocking during multi-user hours; to do this he used the kernel tracer just as he had done for measuring Kernel Semaphores. Again, the properties of the tracer limited the period over which a single measurement could be made, and experiments consisted of a number of samples taken at random intervals. The distribution of blocked intervals is given in Table 16-9.

Table 16-9 Ports and Policy Semaphores: blocked intervals

Delay duration	No.	%
0.0-3.0 ms	3	0.0
3.0-5.0	507	13.8
5.0-10.0	129	3.5
10.0-17.6	92	2.5
17.6-25.0	438	11.9
25.0-50.0	434	11.8
50.0-75.0	641	17.5
75.0-100.0	367	10.0
100.0-200.0	375	10.2
200.0-300.0	74	2.0
300.0-400.0	59	1.6
400.0-500.0	31	0.8
500.0-1000.0	241	6.5
over 1000.0	261	7.1

Blocked durations:	
Minimum:	2.4 ms
Mean:	490 ms
Median:	60 ms
Maximum:	58,919 ms
Total samples:	3,652
Sample period:	166 sec
Processor time in block/awaken:	8.5%

The average time lost to synchronization with Policy Semaphores is about 8%. This is larger than for kernel synchronization but still fairly small. Over 80% of the blocked durations are less than 300 ms (the standard value of *WaitTime* for PM1), so these processes will remain resident in primary memory. Also note that there are a significant number of processes that block in excess of 500 ms; these are probably processes waiting for terminal input. (Recall from Chapter 10 that every user talking to his command language has a JMON process waiting in the background.)

Thus it appears that a majority of user-level process are effectively using the same logical mechanism as kernel semaphores—but at over five times the cost each time a process blocks. While this does not substantially affect system throughput, it can be a significant influence on an individual user's

application, as is shown by Oleinick's data. Those processes that do block for more than 500 ms could probably be handled by making terminal I/O a special case.

These conclusions should be interpreted relative to the particular costs in Hydra. We still believe in the *concepts* of multi-level synchronization primitives and in policy/mechanism separation. The relatively high costs of kernel-entry and domain-crossing in the implementation of Hydra, however, make our version of user-level semaphores of dubious value.

16-2.8 Marathe's Small-Address Effect Experiment

The C.mmp programmer faces a problem not unlike that faced by the programmer of an IBM series 360 computer. The physical address space is much larger than the "offset" portion of an instruction. In both cases, to access the full space the programmer must maintain a set of registers and form addresses relative to these registers; the registers in question are "base registers" on the IBM/360 and "relocation registers" on C.mmp.

Most PDP-11 programmers, of course, do not face the problem of managing relocation registers because most PDP-11 systems do not provide more than a 56-64K address space. Programs either live within that limit or use traditional overlay techniques. It is interesting to ask, therefore, to what extent the ability to manage the relocation registers is used on C.mmp and what impact, if any, it has on performance. Marathe used the hardware monitor to study one aspect of this problem, namely, the use of relocation registers in the kernel [Mar77].

Table 16-10 Assignments of kernel space relocation registers

Register	Use	Contents
0	Stack page	Fixed
1	Common Data page	Fixed
2	Data page	Overlayable
3	Data page	Overlayable
4	Code page	Overlayable
5	Common Code page	Fixed
6	Local Memory	Hardware
7	I/O Device Registers	Hardware

To understand the following data, we first must discuss the kernel's use of its relocation registers. The registers are assigned as shown in Table 16-10. Hydra's use of them is typical of many large application programs. Some registers hold fixed values by programming or hardware convention; others are changed dynamically to address code or data. The stack page is mandatory. Judgments are made as to what code (data) to place in the Common

Table 16-11 Frequency of relocation register access: sixteen samples

Instructions executed in kernel	Accesses to to relocation registers	Instructions per access	Instructions per change
45,226	2,913	15.5	46.5
73,433	5,130	14.3	42.9
33,568	1,843	18.2	54.6
55,957	3,438	16.3	48.9
48,256	3,099	15.6	46.8
33,258	1,837	18.1	54.3
33,161	1,798	18.4	55.2
36,759	2,072	17.7	53.1
39,239	2,359	16.6	49.8
68,575	4,729	14.5	43.5
49,366	3,196	15.4	46.2
35,238	1,980	17.8	53.4
66,763	4,595	14.5	43.5
32,973	1,824	18.1	54.3
69,702	4,768	14.6	43.0
68,953	4,653	14.8	44.4
	Averages		
49,401	3,139	16.3	48.8

Code (Data) page and what code (data) to swap. Hydra has somewhat less flexibility in managing relocation registers than do user programs because relocation registers 6 and 7 in kernel space are not actually usable.

Marathe's experiment consisted of executing RootFinder-N (see Section 16-2.4) and sampling sixteen 1-second intervals. During each sampling interval, the number of kernel instructions and the number of accesses to one of the relocation registers were measured. In most (but not all) cases the kernel code saves and restores relocation registers using a standard Bliss/11 macro. The save-load-restore sequence involves three accesses to a relocation register; so in Table 16-11 we display both the raw data and an adjustment for the multiple references per change. As can be seen from Table 16-11, a relocation register is accessed, on the average, every 16.3 kernel instructions. It would appear, then, that relocation-register manipulation costs only 5.5% in kernel performance. Unfortunately, this number belies the real price of C.mmp's relocation register structure. First, it does not account for the cost of the other code and data needed to maintain the relocation-register values. Second, it does not account for the added difficulty of programming the machine. Users agree that the small address space is the worst feature of C.mmp and causes substantial increases in programming time and errors. Finally, it does not account for the (inestimable) cost of not being able to make the data-part of objects directly addressable.

16-2.9 The Small Address Effect on HARPY

In Section 16-2.3 we discussed the performance of the HARPY speech recognition system. The performance on two tasks was mentioned—the 30-word DESCAL task and the 1,000-word information-retrieval task. An important difference between the performance of HARPY on these two tasks was not explored in that section—and we will now do so here.

The data base for the 30-word task is small enough to fit into the 16-bit address space of a PDP-11. The data base of the 1,000-word task is not, and thus explicit memory relocation must be programmed. The question of interest is whether the need to use this facility has any effect on the performance of the 1,000-word version.

To answer this question, a special version of the 30-word system was constructed. Just as in the regular version of the 30-word task, the complete data base was held in primary memory and it was never necessary to alter a relocation register. However, the special version *did* test to see whether a relocation register change was needed—just as the 1,000-word version must do. The results of this experiment are shown in Figure 16-14. As is obvious, the system pays a healthy penalty—nearly a factor of three—to support the possibility that an addressing change might be needed.

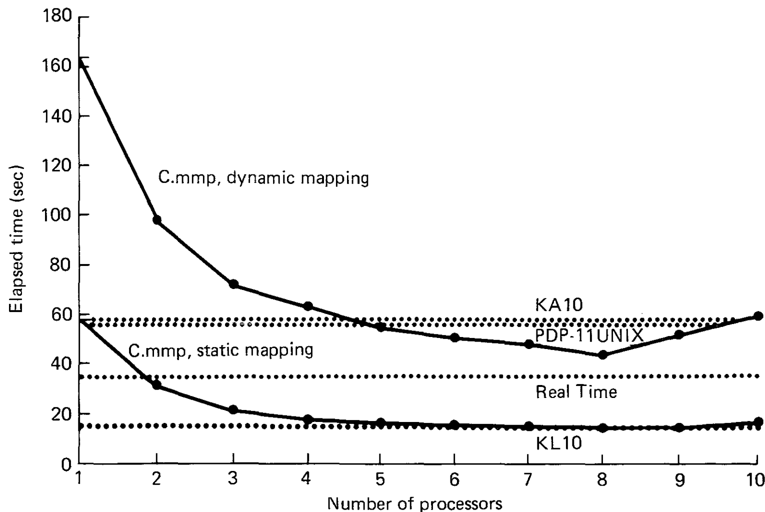

Figure 16-14 The effect of the small address space on HARPY

16-2.10 McGehearty's Kall Measurements

In the spring of 1977, McGehearty examined the frequency and cost of the various kernel Kalls found in interactive tasks. We shall report only a portion

of his results here; the remainder may be found in [McG80]. In particular we shall look at the results obtained by using the kernel tracer to examine five activities:

- Entering SOS, an interactive text editor written in L* (a list processing and system building language
- Leaving SOS
- Listing a small file on the line printer
- Entering TECO, another interactive editor written in Bliss/11
- Leaving TECO

The two editors and the listing program were invoked from the command language interpreter and involved using the directory system to look up the object to be edited. Returning from the editors involves returning to the command interpreter. Thus each of these activities involves substantial protection-domain crossing, moderate I/O, and relatively small programs and data. Taken together, we believe they are representative of the heaviest use of kernel facilities in interactive systems on Hydra/C.mmp. In a sense they represent the opposite extreme from compute bound tasks such as Baudet's relaxation experiment, which is completely compute-bound.

Table 16-12 gives the relative proportion of time spent in the kernel as opposed to at user-level during the execution of these tasks. Time inside the kernel is further divided into time spent in interrupt routines (hence handling I/O[12]) as opposed to time spent directly handling Kalls. Note that these applications were chosen because they are heavily Kall-intensive; they are probably not typical of a normal user load.

Tables 16-14 through 16-19 give a detailed list of all of the kernel operations used during these activities—their frequency and their minimum, maximum, and average costs. Table 16-13 summarizes this data.[13]

To interpret these data properly, one should bear in mind that in early 1977 essentially no optimization or tuning had been done to Hydra. Since that time,

- PDP-11/40 processors are used instead of the 11/20s on which this data was collected. This means that the Kall timings should be reduced by about 40% to come into agreement with later measurements.
- The Call mechanism was improved in several ways. The average Call time is now 20-30 ms rather than the 60 ms cited.
- A microcoded implementation of *RPSLoad* now exists; it requires 18 μs rather than the 350 μs taken by the version reported here.

[12]The I/O being handled is not necessarily related to the activity being measured; it includes kernel demon activity, clock interrupts, etc.

[13]Some Kalls are omitted in the detail tables because they have not been discussed in this book. Table 16-13 includes *all* calls, however.

With the exception of a few operations that were handled specially (e.g., *RPSLoad*), the minimum cost for any Kall is over 700 μs in this data—about 150 instruction times. Sadly, it takes this much code to validate the stack and do other kernel entry checks. We believe this is due mostly to the PDP-11 architecture.[14]

Table 16-12 Processor activity under an interactive load

Activity	Response (sec)	User (%)	Kernel (%)	Interrupt (%)
Entering SOS	11.0	26.9	51.8	21.3
Leaving SOS	5.0	15.3	50.8	33.9
Listing a File	10.0	9.2	56.6	34.2
Entering Teco	4.0	29.6	45.4	25.0
Leaving Teco	2.3	19.3	47.0	33.7

Table 16-13 Kall usage in interactive tasks

Activity class	No. of Kalls	% of all Kalls	% of Kalls
Call mechanism	248	6.8	35.9
Ports and Semaphores	1,155	31.7	18.4
Paging Kalls	1,193	32.7	9.7
Create and *Copy*	76	2.1	8.9
Capability manipulation	710	19.5	9.2
Policy Module Kalls	204	5.6	12.5
Update	16	0.4	4.7
Alter kernel tracing	17	0.4	0.4

Table 16-14 Kall timings: the Call mechanism

Operation	No.	Mean	Max	Min
Call	5	92.5 ms	163.6 ms	33.0 ms
TypeCall	55	67.2	128.5	31.2
(Average Call)	60	69.8		
Return	56	42.9	144.2	16.4
Merge	22	2.1	2.9	1.5
Compare	27	1.3	3.1	0.6
LNSLength	23	0.5	1.0	0.4

[14]This checking involves making sure there is enough stack space to handle the Kall or interrupt, decoding the Kall number and branching to the correct code, locating the Kall arguments, saving the user's registers, etc.

Table 16-15 Kall timings: ports and semaphores

Operation	No.	Mean	Max	Min
ReceiveMsg	70	7.8 ms	23.9 ms	2.9 ms
ReadMsg	47	2.6	3.0	2.3
WriteMsg	50	2.7	5.3	2.5
RSVPMsg	73	5.6	12.8	3.5
CreateMsg	17	4.5	5.9	2.8
ReplyMsg	13	5.0	6.7	3.1
GetMsgCapa	3	2.6	2.7	2.6
PutMsgCapa	2	3.7	4.2	3.3
Connect	2	3.2	3.2	3.2
Disconnect	3	6.5	9.2	2.3
RequeueMsg	4	4.0	4.2	3.9
P (Pol. Sem.)	23	2.3	7.4	1.2
V (Pol. Sem.)	24	2.0	4.7	1.9
P (Kern. Sem.)	340	2.4	9.0	.7
V (Kern. Sem.)	410	2.0	5.9	.7
PConditional (Kern. Sem.)	70	2.0	5.5	1.4

Table 16-16 Kall timings: paging Kalls

Operation	No.	Mean	Max	Min
CPSLoad	47	27.4 ms	74.8 ms	0.7 ms
RPSLoad	1,092	0.35	4.1	0.3

Table 16-17 Kall timings: object creation

Operation	No.	Mean	Max	Min
Create	21	29.4 ms	145.5 ms	7.5 ms
Copy	4	51.6	66.3	17.2
MakeData	8	11.3	17.2	6.7
MakePage	28	15.6	33.0	6.3
MakeUniversal	15	15.5	25.7	6.4

Table 16-18 Kall timings: C-list and data-part manipulations

Operation	No.	Mean	Max	Min
GetCapa	128	3.3 ms	51.6 ms	0.8 ms
GetData	177	1.8	5.7	0.7
PutCapa	68	3.2	26.8	0.7
PutData	59	1.7	3.3	1.4
Delete	170	2.0	28.7	0.7
CLength	22	2.0	16.8	1.0
DLength	26	1.0	1.4	0.9
AppendCapa	32	2.8	16.4	1.7
PassCapa	25	3.0	16.8	1.0
PassAppendCapa	1	10.1	10.1	10.1
Interchange	2	2.1	2.5	1.8

Table 16-19 Kall timings: Policy Module interactions

Operation	No.	Mean	Max	Min
StartProcess	36	22.2 ms	72.9 ms	8.9 ms
StopProcess	2	2.6	2.9	2.4
ReceivePolicy	66	7.7	13.1	2.2
AttachPolicy	2	2.6	3.2	2.0
Runtime	11	1.4	1.9	1.0
(Delay)		58.1	214.1	1.0
SetSchedParms	12	1.5	2.1	1.2

We can make several general observations about Hydra/C.mmp from this data.

1. Even the simplest Kalls take on the order of 500 μs on a PDP-11/40; faster communication between user and kernel code is needed. This involves the architectural details inherent in the PDP-11.
2. Capability manipulation is not inherently difficult; the minimum times are barely more than just kernel-entry costs. The longer maximum (and average) times result when objects must be swapped in from the Passive GST.
3. The *Call* mechanism is slow, but not necessarily because of anything intrinsically difficult in domain switching. Rather, it is our implementation of domain switching that is expensive. For instance, every *Call* creates a new LNS object, and *Create* alone averages 30 ms out of *Call*'s total of 70 ms. We believe that several other implementation approaches could have been used to advantage, and we shall say more about this later (using the data in the tables).
4. Interprocess communication in Hydra is really faster than procedure calls. The pair of composite Message System Kalls, *ReceiveAndRead* and

WriteAndRSVP, take about 14 ms, compared with 40-50 ms for the improved *Call* plus *Return*.

5. Process context switching is also slow, probably on the order of 1 ms. This is largely due to the large amount of hardware state involved in our modified PDP-11s and memory relocation hardware.

It is interesting to contrast this collection of data with that from, for example, Oleinick's HARPY experiment. The two, we believe, represent extremes. HARPY is basically compute bound, makes few calls on Hydra, and uses the full processing power of the machine; Hydra does not degrade its performance. The present data, on the other hand, represents a case in which most of the "action" is achieved by the programs' extensive use of the kernel operations—and here half, or more, of the time is spent in the kernel. During general user sessions, depending upon what users are doing at a given moment, the ratio lies somewhere in the middle of these extremes.

16-2.11 Size of the Hydra Kernel

Reporting the code size of the Hydra kernel is not an "experiment" in quite the same sense as the others in this chapter. Nonetheless, it seems relevant to discuss the size of Hydra and the distribution of that size among its component pieces.

Table 16-20 attempts to break down the total system into meaningful units, but any such breakdown has its own set of peculiarities. The portion labeled "Debugging," for example, contains most of the mechanisms used to debug Hydra, but some of them are located in "Autorestart" and in the other specialized modules. Similarly, most of the code for communication between the kernel and a Policy Module is contained in KMPS, but some is in Message System code. Finally, we are unable to properly reflect certain distributed costs. In particular, error detection and recovery, debugging facilities, and tracing all involve code that is distributed throughout all the modules; the tables given in the table are only for the service routines that provide the mechanisms that support these facilities. Error detection and recovery is, perhaps, the most notable example of this; we have no way to measure its impact, but some modules such as IO and the GST may have as much as 30% of their code devoted to this. Thus, although the table represents our best attempt at a meaningful breakdown, the numbers should be treated as indicative, not absolute.

The data in Table 16-20 was derived from a linkage-editor map of the system during October 1979. Of course, the size of the system fluctuates slightly as changes are made. The size of the various modules in words can be converted to a number of instructions by dividing by 1.5; from other sources we know that the typical PDP-11 instruction is 3 bytes long. Also, we have made various measurements of the size of the kernel in terms of lines of source code. On the average, one line of Bliss/11 source yields one

Table 16-20 Size of the Hydra kernel

Functional classification	Words	Percent
I/O		
Device handlers	30,786	23.6
Common support	4,832	3.7
GST	20,740	15.9
Operations	12,019	9.2
Debugging Support	11,467	8.8
Protection mechanism	8,564	6.6
Paging	7,897	6.0
Messages and Semaphores	7,617	5.8
KMPS	7,557	5.8
Initialization	5,644	4.3
Exception handling	4,493	3.4
Autorestart	3,852	2.9
Storage Allocation	2,838	2.2
Tracing	1,018	.8
Processor support	637	.5
Bliss support	663	.5
Totals	130,624	100.0

compiled instruction. This means that the entire kernel is about 100,000 lines of code.

The most obvious conclusion from this data is that Hydra is large—much larger than the designers imagined before they started. The second observation is that a large fraction of code is devoted to things unrelated to the philosophy of Hydra: I/O, error recovery and diagnostics, initialization, operations, debugging, etc.

In addition to the obvious conclusions, however, this data also provides some insight into other aspects of Hydra/C.mmp. Consider, for example, the storage allocation module. This module provides allocation of small areas of storage for buffers, messages, etc. The original intent was to use an algorithm called "quick fit" that had been in use in the Bliss/11 compiler and was known both to be fast and to avoid fragmentation problems [Wei76]. This algorithm, coded for the PDP-10, is less than 200 instructions. Yet, as you can see, the Hydra storage allocator is an order of magnitude larger. Why?

In the earlier section discussing the use of relocation registers by the kernel, we asserted that the data belied the difficulties caused by C.mmp's mapping structure. One of those difficulties is reflected in the size of the storage allocator. Although it would be unfair to lay the whole blame on the small address problem, it is a major factor. Because of the mapping hardware, allocated chunks must be entirely within a single page. Thus, while the "quick fit" algorithm could be applied within a page, it was

necessary to devise mechanisms that could search through multiple pages, could return an entire page if it became free, could request new pages when the currently allocated ones are full, and so on. Moreover, one had to devise policies that would tend to avoid situations in which only a small fraction of each page was occupied, and so on. As it became obvious that performance of the allocator was a major factor in the speed of some kernel operations, it was necessary to devise efficient intra-page search lists. In short, what had been a simple problem on a machine with a larger address space, became a *much* harder problem than simply managing the relocation registers.

16-2.12 McGehearty's "Stretch Factor" Experiment

Despite the fact that Hydra was intended to be used in an interactive, time-sharing mode, the lack of a large user community resulted in relative neglect of this aspect of the system's performance. Comparatively few measurements were made that characterize its performance in this mode, and essentially no effort was put into improving that performance—it was "good enough" for the light loads normally encountered. Thus, in this section we will simply present the results of the one controlled experiment in this area that was available at the time of this writing; it is not an ideal characterization of the system's performance as a time-sharing machine, but it is all the data we have. We expect much more complete results to be available in McGehearty's thesis, [McG80].

McGehearty used the script driver to put a controlled load on the system. The load consisted of n identical simulated terminal users. Each simulated user would request an amount of computing drawn from an exponential distribution with a specified mean. The simulated user would wait for the response to this request and then enter a period of "thinking" before making the next request for computation. The duration of the "think time" was also drawn from an exponential distribution with a specified mean. Each of the simulated jobs was a simple compute-bound loop and did no input or output other than accepting the next compute request and sending a character to indicate completion of a request.

McGehearty expressed the results of this experiment in terms of the "stretch factor." Under no load, a request for c seconds of computing should complete in roughly c seconds, corresponding to a stretch factor of 1. As load increases, however, the user will observe that it takes longer and longer for the system to complete the request. When the time to respond is $2c$, the stretch factor is 2. In general, then, the stretch factor is the observed response time divided by the no-load response.

Figure 16-15 displays the results for a 9-processor system. As can be seen, the number of jobs varied between 1 and 50. Three cases were run; in each case the mean think time was 10 seconds, but the mean compute time per request was 1, 5, and 10 seconds, respectively. Shown on the graph are

the theoretically optimal response, the mean measured response, and the 90th percentile lines (90% of all requests responded in less than the time indicated by these lines).

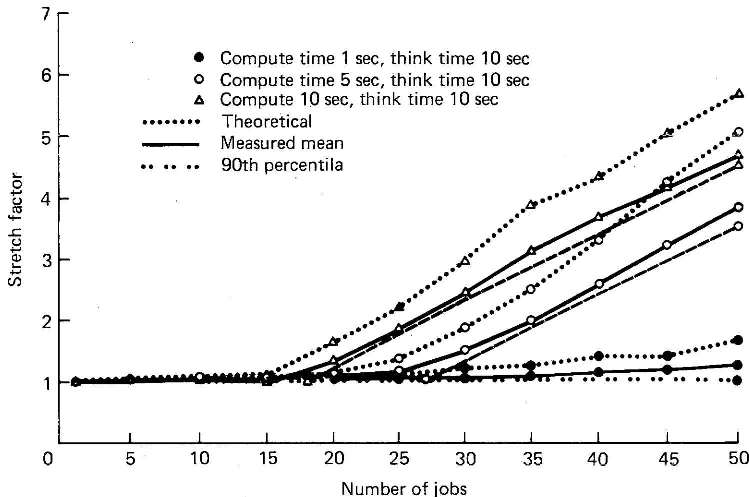

Figure 16-15 "Response stretch factor" as a function of load

The observed stretch factor in these experiments is quite close to optimal. Without Hercules we could not have observed this behavior, because it is seldom we have more than a dozen users on the system.

16-2.13 Almes' Study of the Active GST

Almes [Alm80] studied both the Active and Passive GST. His goal was to implement a parallel garbage collection algorithm; this algorithm is now used in the system to eliminate unreachable objects from the GST. In order to make the garbage collection acceptably efficient, Almes needed data on the way that the GST is actually used: object sizes, creation rates, any dependencies on object type, and so on. The complete data from this study is available in an appendix to Almes' thesis [Alm80]. Here we shall try to summarize only highlights and trends in the data; however, we strongly recommend careful examination of the complete data to anyone contemplating building a capability-based or object-oriented system.

We will first focus on Almes' experiments on the Active GST; the following section will consider the Passive GST study. The study of the active GST consisted of two experiments—one collected data during normal user sessions and the other under a simulated load. The second of these

resulted in more detailed analyses than were practicable during normal user sessions.

The Active GST during user sessions In the summer of 1978, a modified version of Hydra was run for three weeks. This version recorded a small amount of data concerning object passivation and destruction. Several interesting pieces of data were obtained:

- 1,007,621 objects were destroyed during this period. Of this number, over 98% had never been passivated—that is, they were created and destroyed without ever having been written to disk. Of the remaining objects destroyed, two-thirds were "old" in the sense that they had been created before the most recent system restart, and one-third had been created during the current session.
- 282,200 objects were passivated because their active reference count became zero while their total reference count was non-zero. (See Section 11-2.1 for a discussion of reference counts.)
- 128,070 object (representations) were passivated by the GST demon.
- 38,109 *Update* Kalls were made.

The most striking result in this data is that the vast majority of Hydra objects, 98%, or 38,000 per day during the test period,[15] are created, used, and destroyed without ever being passivated. In retrospect the reasons for this are clear—the majority of these objects are LNSs, DATA objects used for parameter blocks, and so on. Alas, the GST contained no optimizations for these special and frequently occurring objects; the full mechanism is available and applied to all objects.

The Active GST under artificial load Since the amount of data that could be collected during user sessions was limited, more elaborate experiments were performed under a collection of artificial loads. Each experiment was conducted in the following way:

1. Hercules, the terminal emulator, was used to simulate five users repeatedly performing some task. The tasks used included copying and editing a Commands object (a special kind of file that contains command-language programs) and retrieving a text file from the PDP-10 via the ARPANET.
2. After the simulated programs had been running for awhile, the kernel tracer was invoked to record all creations, activations, passivations, and destruction of objects.
3. Once enough trace data had been collected, the "snapshot taker" was invoked to capture the state of the Active GST. After the snapshot was

[15]The load on C.mmp was not heavy during this period. Even during its heaviest periods of use, there were seldom more than a half dozen people using the system.

taken, Hercules was terminated.

As a quick indication of the load generated by these tests, during one particular trace lasting 66 seconds, 1,343 objects were created and destroyed; this is a rate of 20 objects per second, or 4 objects per second per user. These objects had a mean lifetime of 8 seconds, and a median lifetime of less than 5 seconds. Because the load was light (5 simulated users), these numbers presumably are determined by the task characteristics and fundamental time constants of the system (CPU speed, Hydra Kall costs, etc.), not by contention for resources.

The major results of these experiments are listed below:

Creation rate. As noted above, the creation/destruction rate is much higher than we anticipated, and most objects are never passivated.

Number of types. During these tests, 38 distinct types appeared in the Active GST; there were 13 kernel-defined types and 25 user-defined types. Table 16-21 lists the most common types as well as those that use the most storage.

Object sizes. Figures 16-16 and 16-17 summarize the size of the C-list and data-part of objects. As can be seen,

- 50-60% of all objects have a C-list.
- The average C-list is 260 bytes (about 16 capabilities).
- 95% of the objects have a data-part.
- The average data-part is about 130 bytes.

There is a strong correlation between an object's size and its type; we will not explore this further here, but it implies that systems could exploit this correlation.

Locking. At any instant, 1-2% of the active objects are locked.

Reference counts. Figures 16-18 and 16-19 show the distribution of reference counts and provide an indirect measure of sharing. As can be seen, the mean total reference count is 5 and the mean active reference count is 4.

Objects in both the Active and Passive GST. Even though 98% of all objects are created and destroyed without having been passivated, 45% of the objects in the Active GST are also in the Passive GST. Moreover, there is a strong dependence on type:

- Eleven of the 38 types present in the Active GST had less than 10% of their objects on the Passive GST; many of these are kernel-defined types (e.g., LNS) that cannot be passivated.
- Seventeen of the 38 types present in the Active GST had more than 90% of their objects in the Passive GST; generally these were user-defined types.

Table 16-21 Active GST types

Most numerous types			Types with most total store		
Type	%	Cum %	Type	%	Cum %
Page	25.0	25.0	Universal	18.9	18.9
Universal	14.6	39.6	LNS	16.0	34.9
Procedure	8.8	48.5	Procedure	13.7	48.6
Semaphore	6.0	54.6	Page	9.8	58.4
LNS	5.3	59.9	CPS	8.3	64.7
Port	4.3	64.3	Port	4.9	69.6
Device	4.1	68.5	ObjectList	3.4	73.0
Process	3.8	72.3	Process	3.2	76.2
SoSFile	3.7	76.0	SubCatalogue	2.8	79.0
PMProcess	3.1	79.1	Type	2.7	81.7

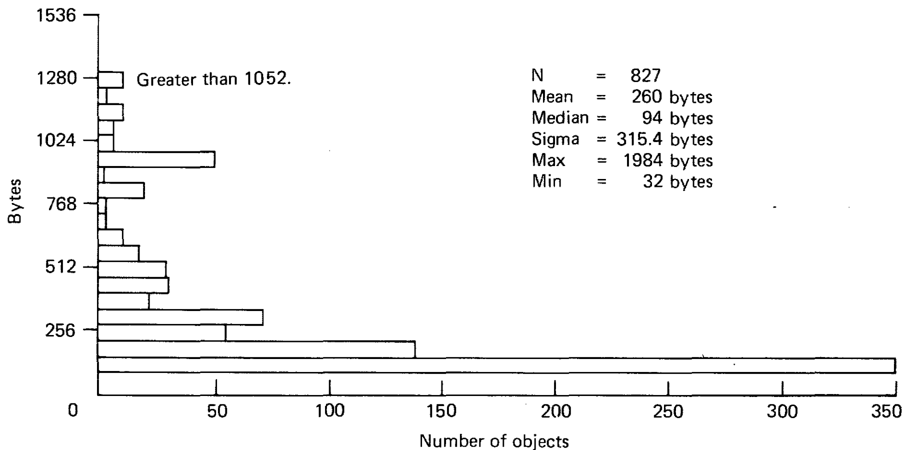

Figure 16-16 Active GST summary: C-list sizes

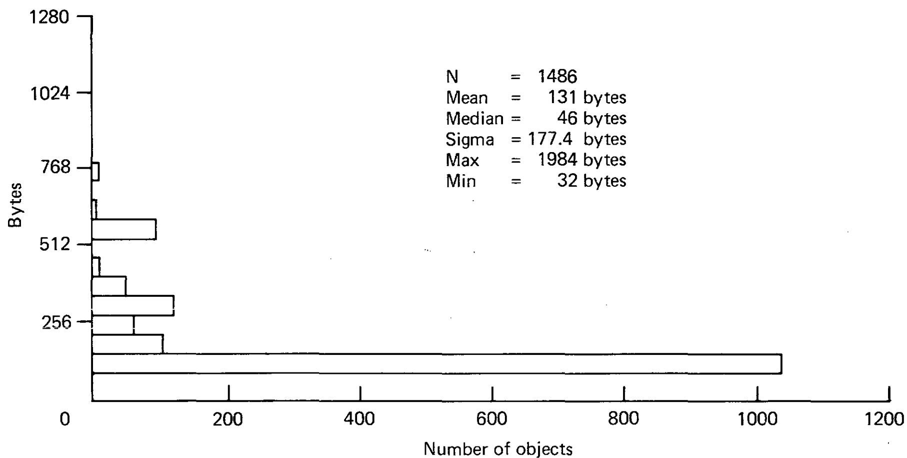

Figure 16-17 Active GST summary: data-part sizes

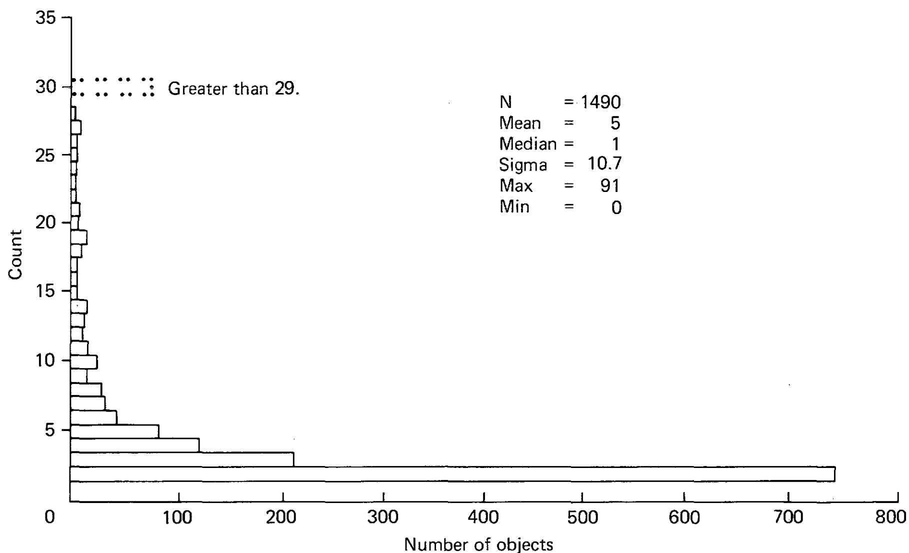

Figure 16-18 Active GST summary: total reference counts

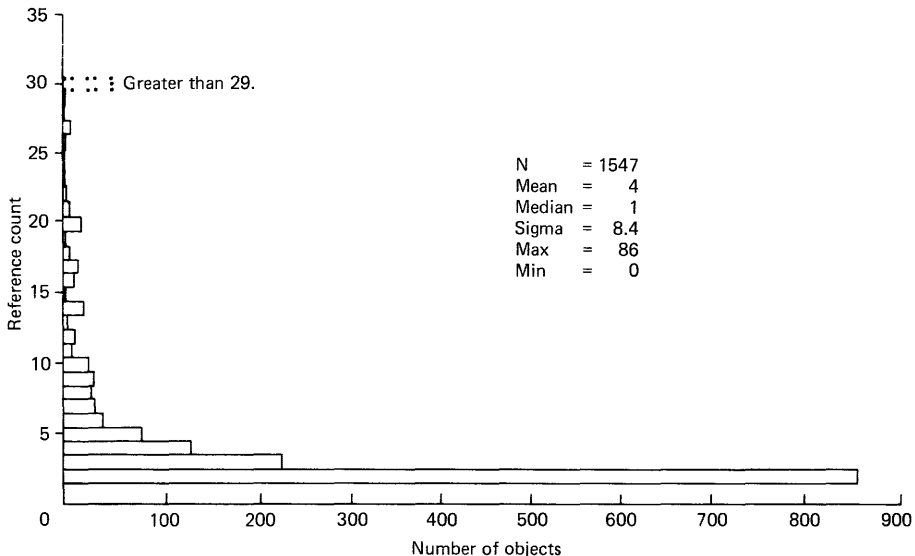

Figure 16-19 Active GST summary: active reference counts

16-2.14 Almes' Study of the Passive GST

The second major part of Almes' study concerned the use of the Passive GST and consisted of a static analysis of its contents—the number of objects, their types, sizes, and reference counts.

The standard operational procedure for "backing up" the GST involves copying the entire passive GST onto spare disk packs. This is done three times each week. Since one disk drive was always available to be assigned to a user program, it was possible to mount the backup packs on this drive and analyze them with user programs. To this end, a special program was written that

1. Eliminated all but the most recent version of an object (the system normally keeps the previously most recent version as well)
2. Eliminated all unreachable (i.e., "garbage") objects
3. Recorded the global name, type, time-stamp of last update, C-list and data-part sizes, (total) reference count, and the global name of each capability in its C-list

A post-processing program analyzed this data and produced a number of interesting statistics:

1. During the period under study, the Passive GST contained on the order of 20,000 objects.

2. Eighty-two distinct object types were observed, of which 14 are the kernel types. Of the remaining (user-defined) types, only 25 were in general use (many of the other user types were either obsolete or were used only for debugging the subsystems that supported the 25 generally used types). The most frequent types, and the most space-consuming types, are shown in Table 16-22.
3. The sizes of objects are shown in Figures 16-21 and 16-22. These sizes are somewhat different than those for the Active GST, as may be seen in Table 16-23.
4. The total reference counts are much smaller than in the Active GST. As can be seen in Figure 16-20, the mean total reference count is 2 and the median is 1. Fully 85% of all objects have a total reference count of 1.

It should be noted, by the way, that the size information may be a bit misleading. Almes measured only the space *in* an object. This makes good sense for all objects except pages since the actual 8K-byte segment represented by a page is not contained in the object's data-part. Unfortunately, it is not clear what should be measured in the case of pages; page segments are precisely 8,192 bytes long regardless of how much information is actually contained in the page. Since Almes was primarily concerned with the GST itself, not with total storage requirements, it seemed more reasonable to exclude the space devoted to the page images.

Table 16-22 Passive GST types

Most Numerous Types			Types with Most Total Store		
Type	%	Cum %	Type	%	Cum %
Page	35.3	35.3	Procedure	22.6	22.6
Universal	17.3	52.7	Commands	20.8	43.4
Procedure	7.9	60.6	Page	16.6	60.0
Semaphore	7.7	68.4	Universal	12.1	72.1
Commands	6.3	74.7	SubCatalogue	8.2	80.3
Data	5.3	80.0	Data	6.5	86.8
Catalogue	4.9	85.0	Semaphore	2.6	89.4
SubCatalogue	4.9	90.0	Catalogue	2.1	91.5
SuperFile	2.0	92.0	Directory	1.6	93.1
SoSFile	1.3	93.3	SuperFile	1.5	94.6

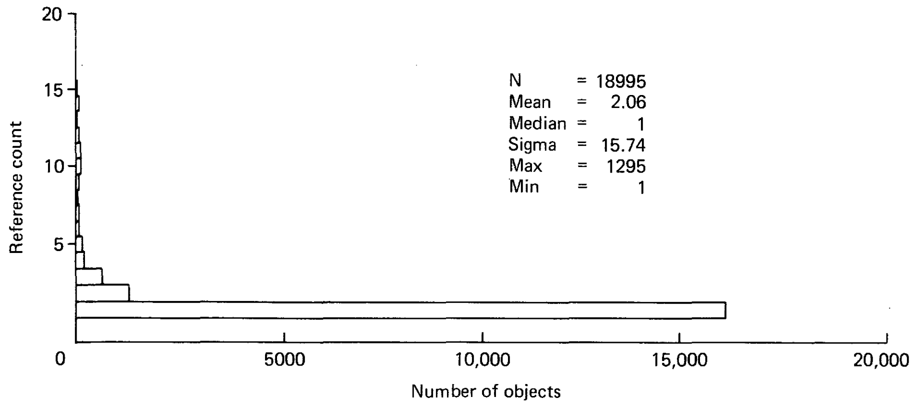

Figure 16-20 Passive GST Summary: total reference counts

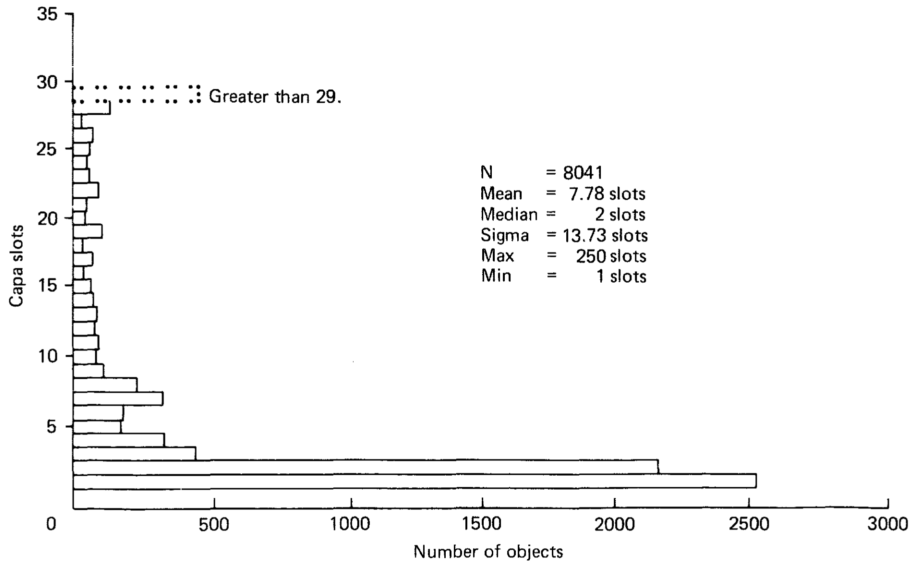

Figure 16-21 Passive GST summary: C-list sizes

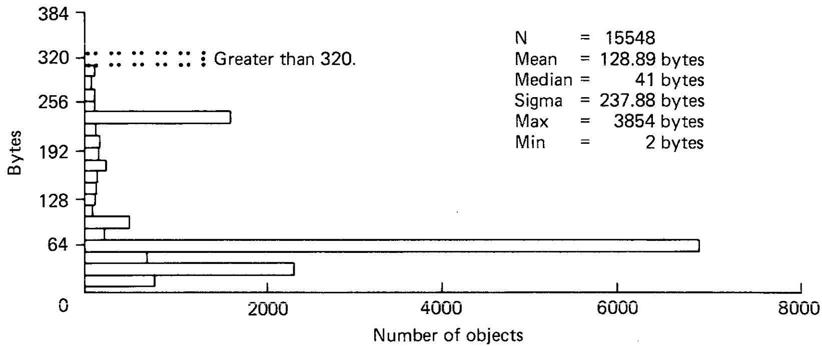

Figure 16-22 Passive GST summary: data-part sizes

Table 16-23 Mean sizes of objects in the Active and Passive GST

Part	Active GST	Passive GST
Fixed-part	64 bytes (20%)	64 bytes (29%)
Data-part	124 bytes (38%)	105 bytes (47%)
C-list	138 bytes (42%)	53 bytes (24%)
Total size	326 bytes	222 bytes

16-3 RETROSPECTIVE

As we noted in the introduction to this chapter, there are many reasons why the various results reperted cannot be compared or combined casually. Caution is necessary. However, it seems fair to conclude the following.

Tools In general, the tools available for measurement have been adequate. In a few cases, such as a better analysis of memory contention, another tool would have been desirable. However, the limitations of the tools have been less important than the benefits of being able to use them interactively during normal user sessions.

Contention Under normal use, contention is low—both for hardware and software resources. We believe that the decisions to build a distributed system, to reject a master/slave hierarchy, to lock data rather than code, and to have a large number of locks were correct and that that belief is supported by the data. There are, however, two exceptions to this conclusion:

- Ports and Policy Semaphores are much too slow. This, we believe, is primarily the result of an improper implementation. Despite the *WaitTime* facility, most of the work of stopping a process and preparing to return it to its PM is done every time the process blocks on one of these objects.

A simpler implementation would have used an ordinary Kernel Semaphore and a demon to handle processes that block longer than their *WaitTime* value. Had we done this, the overhead of blocking on Policy Semaphore objects would have been similar to that of blocking on Kernel Semaphores.

- Pooling resources such as processors and memory is usually the correct approach. However, in a system without caches, and in the presence of shared code pages, we should have provided another mechanism, something that allowed the programmer to indicate that (1) separate copies of the pages should be created, and (2) that these copies should be placed in different memory units.

Synchronization We do not know whether providing the analogs of locks and kernel semaphores to users would have been appropriate. The issue is not technical, but rather a matter of operational policy. Possession of either locks or kernel semaphores allows the individual user to preempt resources that could otherwise be given to another user. Wisely used, of course, both of these facilities could save those resources for all users. Alas, the kind of protection provided by capabilities does not address the (mis)use of a service to which a user has access.

Efficiency As is usually the case, the most important component affecting performance is the user's algorithm, not the operating system or resource contention.

Policy/mechanism separation The concept of policy/mechanism separation still seems a good one to us, but its cost in the Hydra implementation is too high. Moreover, there is no agreement among us about the cause for this or how to correct it in a second iteration. Some of us believe that the level of the kernel-PM interface is too low and that the kernel should be allowed more discretion over paging decisions; in this model, the PM adopts a more advisory role. Others of us believe that precisely the opposite is the problem and that the PM should have more control over (for example) paging decisions independently of scheduling decisions. This latter group argues that the PM, and the kernel/PM interface, are slow because the PM must second-guess too many kernel actions.

The object model The GST should be optimized according to the observed usage patterns. More hardware support, and particularly support for short-lived objects would be a great help in this.

The small address space Oleinick's HARPY data clearly shows the performance penalty paid by processes wishing to address large data segments. It does not show the increased programming burden, although we think that is just as great.

With a large virtual address space, the direct addressing of objects' data-parts becomes feasible and the need for separate PAGE objects may disappear. This would also simplify the programming of many subsystems.

CHAPTER

SEVENTEEN

REFLECTIONS

In this chapter we would like to reflect again upon what we learned from the Hydra/C.mmp experience, and what we hope others can learn from us.

Much of what appears here is simply a rephrasing of the more important points that appeared in the retrospective sections of earlier chapters. There is only a limited amount of organization that can be placed on a chapter like this. It is inherently a list of (only) somewhat related points; so, except for a wrap-up at the end, we shall not try to pretend otherwise.

On multiprocessors Generally, the architectural structure of C.mmp was a complete success. In particular,

- The memory contention that has plagued other multiprocessors was simply not present under normal loads.
- The simple interprocessor interrupt structure proved completely adequate.
- The asymmetry of the I/O structure, the fact that devices are attached to particular processors, posed no problem to creating a symmetric virtual machine at the user level.
- The crosspoint switch, a potential reliability bottleneck of the system, in fact proved to be one of our most reliable components.

If the ground rules were the same again, that is, if we were to be asked to build another symmetric multiprocessor, we would not change these decisions. We would not, for example, want a more powerful interprocessor communication facility. Neither would we want a distributed processor/memory switch.

Issues of technology and cost aside, simplicity and symmetry are among the greatest allies of the system designer/implementor. Cost factors might dictate a hierarchical switch structure, but its existence would simply add another problem to those that the software and users must cope with. The ability to treat all processors as identical, and the ability to assume that the access time from any processor to any memory is identical, both simplified the design. Had I/O been symmetric, it would have simplified the design even more.

The *realization* of C.mmp, as opposed to its design, has a number of failings that we correct in a second iteration:

- The "small address problem" was unanticipated, and most unfortunate. It skews almost any attempt to evaluate the machine and its software. Sixteen-bit computers were relatively new when we started on C.mmp, and we had little choice—but we certainly would not choose a small-address machine again.
- Reliability was a much greater problem than we had anticipated. In large measure this resulted from our naive assumptions about the reliability of the PDP-11. In retrospect, however, our own designs should have been more robust to the problems generated by the 11s. We believe that reliability will be a major issue in the coming decade, and a fundamental attitude which a designer must have is suspicion toward all other components. This is equally true of hardware and software. One cannot allow a malfunction in one component to disable another.
- Initially we thought that it would be possible to partition the system and, for example, do hardware maintenance on one partition while running Hydra on another. This didn't work for several reasons, including the inability to reconfigure devices and the presence of transients when components were powered up or down. We would strive to achieve a partitionable system next time.

In general, we believe that it's possible to make two major mistakes at the outset of a project like C.mmp. One is to design one's own processor; doing so is guaranteed to add two years to the length of the project and, quite possibly, sap the energy of the project staff to the point that nothing beyond the processor ever gets done. The second mistake is to use someone else's processor. Doing so forecloses a number of critical decisions, and thus sufficiently muddies the water that crisp evaluations of the results are difficult. We can offer no advice. We have now made the second mistake[1]—for variety, next time we'd like to make the first! Given the chance, our processor would:

- Be both inherently more reliable and go to extremes not to propagate errors; once an error is detected, it would report that error without further effect on the machine state.
- Provide rapid domain changing; we see no inherent reason that this should require more than, say, a dozen instruction times.
- Provide an adequate address space; actually, rather than a larger number of address bits, we would prefer true capability-based addressing at the instruction level since this leads to a logically infinite address space.

There are other things, of course, but these are the most important.

[1]Twice, in fact. The second multiprocessor project at C-MU, Cm*, also uses the PDP-11.

On multiprocessing The multiprocessing structure of Hydra seems sound to us. A number of other multiprocessor systems have experienced debilitating overheads; most of these systems have been adaptations of operating systems that were initially designed for uniprocessors. Hydra clearly shows that a system engineered from the start for multiprocessing need not suffer these problems. Typically, for example, less than 1% of the processing power is lost to software contention. Critical to this are some of the more fundamental decisions:

- The decision to build a symmetric system, as opposed to a master-slave one, was correct. It is hard to prove that without having done it both ways, of course, but we believe the system is both simpler and more efficient because of this decision. It's simpler because it's more regular. It's more efficient for two reasons: (1) we can fully exploit the parallelism, and (2) a processor can directly perform whatever service is needed—one never has to ask the "master" to do it.
- The decision to lock data rather than code was also correct. This is now the modern theology—but it wasn't when we began in 1971. It seems clear to us that this decision, coupled with the decision to have a large number of locks (alternatively, to have each lock guard a small structure) is the reason there there is so little software contention.
- The decision to provide several levels of synchronization still seems like a good idea to us. Certainly the distinction between locks and Kernel Semaphores was crucial to the system's performance. In a second iteration of the design we would make the semantics of these identical, rather than merely similar. We would also implement Policy Semaphores differently. However, the concept of multiple levels would stay—and might even be expanded to include other levels, say between Kernel Semaphores and Policy Semaphores.
- We believe that having better linguistic constructs, such as monitors, would not improve reliability; experience indicates that making critical sections small causes an increase in the complexity of the algorithms (and more errors) *outside* the synchronization primitives.

The realization of the multiprocessing aspects of Hydra is pretty good, too. Of course there are things we would change, and of course we think we could make it both smaller and faster in a next version. However, we see these changes as fine tuning a generally sound approach. What would we change?

- We would use more processes in Hydra itself. We would not go to the extreme that some recent systems have of making essentially all services into processes—the subroutine model is too natural, and usually more efficient. We would use processes to delay those actions which logically (or naturally) occur asynchronously. Often this can be done with no

visible change to the semantics; at other times a slight change is beneficial.

- The interface between KMPS and Policy Modules needs to be completely rethought. We still believe in principle that an efficient policy/mechanism separation is possible, but we did not achieve it. The benefits of such separation are great, and we would not easily abandon them. As of this writing, however, we do not have a firm idea of how to achieve it efficiently.
- The (in)efficiency of Policy Semaphores is closely, but not completely, associated with the problems of the KMPS-PM interface. Within the context of the current interface, we now believe more efficient Policy Semaphores are possible. If this more efficient implementation had been used, we would feel less guilty about not providing Kernel Semaphores or locks to users.

It is probably worth noting, in addition, that the use of semaphores did not create especially difficult synchronization problems. In a next iteration we would be tempted to use more modern notions of synchronization such as monitors. However, the absence of a more structured mechanism did not result in many, or especially serious problems. To be sure, there were synchronization errors, but it is not clear that simply a better linguistic construct would have avoided these.

On the object model The object-oriented, type-extensible, capability-based structure of Hydra is, in some ways, its most interesting contribution. We think there are a lot of things right with it, a few things that are wrong, and some things we would just like to change.

First, at the top level, the concept of providing a kernel that supports type-extension is *exactly* right. The addition of a protection mechanism that smoothly extends to allow dynamic type creation is also correct.

- The approach allows for, indeed caters for, the inevitable evolution and adaptation that all real systems experience.
- The approach is non-preemptive. No user must suffer with inappropriate facilities *if* he has the energy and gumption to define the appropriate ones for himself. Much the same comment applies to the protection facilities. Security and protection are related, but distinct, concepts. For the "typical" user, the base protection facilities of Hydra will be adequate for the kinds of security he needs. But, for applications needing greater (or simply *different*) security policies, they are definable within user-level code.
- The intrinsic costs of the approach are low. Hydra's overheads (especially for Call) are too high, but these costs are not endemic to the approach—as we could have proven had we had the opportunity for a second iteration.

Second, capabilities are the right basis for providing this kind of object-oriented, extensible environment. We would, however, change several things about our design and implementation of capabilities and objects. Aside from preferring the "right" hardware, we would:

- Optimize the system toward small, short-lived objects. Where before our mental model for the size and creation rate of objects was that of "files" in a traditional system, our model now would be that of "records" in a programming language.
- Make the notion of "subsystem," or "type definition," actually, more central and explicit. Several things are involved here. First, we now believe that *TypeCall* is more fundamental than *Call.* Second, we believe that the full generality of rights amplification is not necessary for the vast majority of applications. Finally, we believe that the collection of procedures that define a type should be a first-class concept in the system; this collection is really a more important notion than that of the individual procedures. Thus, in a next iteration we would probably have a notion of type-defining-module with multiple entry points and complete amplification of objects of the type being defined.[2]

Third, we would reconsider the "one-level store" decision. We believe that from a programming point of view, this abstraction is a good idea. We would strive to keep it—and, indeed, make it complete by eliding the *Update* Kall. However, supporting this abstraction adds substantial complexity and size to the kernel. It would be nice to find a way to remove this complexity from the portion of the system that operates unprotected. One possibility, for example, is to make Passive GST management, like Policy Modules, a user-level subsystem. Seeing how to do this in a secure way is an interesting problem that we leave to our readers (see also [Stu74]).

Finally, there are a collection of problems that we didn't address:

Accounting. No one "owns" an object in the Hydra scheme of things; thus it's very hard to know to whom the cost of maintaining it should be charged.

Incremental backup. In conventional file systems it's possible to incrementally dump the updates to a file and thus recover it if necessary. The corresponding operation for the GST is much more complex because of its graph structure. Indeed, in the presence of sharing, it's not at all clear what the intended effect of a back-up should be.

Revocation. Revoking privilege in a capability system is more difficult than in an "access list" system. Roger Needham, however, once observed that "this, like all other problems in Computer Science, can be solved by one more level of indirection." Schemes for adding this level of indirection

[2]The designers of the CAL system [Lam76] foresaw the importance of both short-lived objects and subsystems and designed them into their system.

are sometimes called "aliases" and are now reasonably well known [Red74]. We would like to have seen such a scheme implemented and used.

For the few who are still skeptical about the possibility of using higher-level languages for operating systems, we must note that all but a very small fraction of Hydra is written in Bliss/11. About the only assembly language code appears in code which is so machine-specific that it is actually clearer in assembly language. This includes the context-switching mechanism, the error detection mechanisms, and the processor diagnostics. It does not include any of the interrupt routines or I/O drivers, which are all written in Bliss.

We should also note that the Bliss/11 compiler produces exceptionally fine object code. Although we cannot prove it, our feeling is that Hydra would have been *less* efficient if it had been coded in assembly language. It certainly would have been nearly impossible to maintain.

In the next iteration we would certainly choose a more modern implementation language, particularly one that provides type definition and checking (and preferably full data abstraction facilities). To be acceptable, it would have to produce code comparable to that which we are accustomed to. Under no circumstance, however, would we revert to assembly language.

On managing a research project One cannot avoid making a few comments about the management and research strategy of the project. In general, the management was loose. There was one faculty member (Wulf) nominally in charge of all aspects of the project. There was one full time engineer from the start, and two full time software people were added at later stages. The remainder of the project consisted of other faculty, interested in their own research aspects of the project, and graduate students. In practice, the subprojects functioned as autonomous groups, meeting as necessary to resolve problems.

In an industrial context, and even in some universities, such an informal organization would not have worked. Indeed, several of Hydra/C.mmp's failings can be traced to the management. However, it was a style that suited the particular individuals—and perhaps that is more important than adherence to some preconceived notions of management structure. There are, however, a number of management decisions that we would make differently given the opportunity:

- In at least one dimension, we were too ambitious. We set out to build a full, general-purpose, time-sharing system. In retrospect, there was no chance that we could construct all the software necessary to achieve that goal—editors, compilers, debuggers, etc., in addition to the operating system. We should have chosen a narrower goal and supported it extremely well. Had we done so, we would have attracted more users,

who in turn would have developed more software and hence attracted more users, and so on.

- Reliability is not an add-on feature. We should have designed the error detection and recovery facilities into the base system. Patched on, as they were, they worked reasonably well—but not nearly as well as they might have otherwise. In this regard, it's worth noting the power of the notion of "hitting the leading term," that is, covering the most common errors. Our experience is that, at any given stage, most errors arise from one of a small number of sources. Providing the software to detect and recover from those errors produced dramatic improvements in reliability.
- Generally, we invested too little, too late in tools. The one exception to this was Bliss/11, which was a big "win." For the most part, however, we developed debuggers, hardware diagnostics, etc., only after beating our heads against the wall.
- One always tends to focus on the new, exciting aspects of a problem. We were no exception. We focused on the contention problems and on the capability/object mechanism. We neglected much of the user-visible interface until too late.

17-1 REFLECTION ON THE REFLECTIONS

Our goal in writing this monograph was to describe, as best we could, the design decisions in Hydra/C.mmp and the consequences of those decisions. Our hope is that by doing so, our colleagues who also design and implement operating systems will profit from our experiences. To that end, we felt that the "reflections" section of each chapter was an essential, if subjective, part of the book.

For a number of reasons, our reflections tend to focus more on the things that are wrong with Hydra/C.mmp than on the things that are right with it. The performance bottlenecks, the awkward places, the inconsistencies and missing parts all present obstacles to using the system. They get in one's way and hence are much more noticeable than the things that are well designed, efficient, and "smooth." The fact remains, however, that overall we feel the system was a great success—and much of it ought to be emulated in future systems.

Hydra/C.mmp routinely runs with from 8 to 16 processors, depending upon how many are functioning at a given moment. Configuration is automatic, and memory contention is negligible. It normally runs without an operator and automatically detects and recovers from nearly all hardware and software failures; the period between (necessary) manual reloads is more than an order of magnitude larger than the time between failures.

Users routinely spawn a large number of processes, sometimes 50 or more per job, and the overhead for managing these is comparable to, or less

than, those in other systems. When needed, essentially the full processing power of the machine can be applied to a single problem. In more typical multi-user situations, response is comparable to that observed in conventional systems.

Definition of traditional operating system facilities by user-level subsystems is the norm. Several command languages, directory (catalogue) systems, file systems, and schedulers (Policy Modules) can and do coexist. Through use of *TypeCall*, facilities such as command interpreters and compilers need not know which file system, for example, they are using.

Hydra/C.mmp works extremely well. Yes, there are rough edges, but its remaining problems are those of any ambitious new system and provide the fodder for further research. Hydra addresses problems that we feel will become extremely important as computing becomes ever more ubiquitous. We feel that the approaches and solutions it provides are portents of the systems of the future.

REFERENCES

[Alm77] Almes, G., and G. Robertson. *An Extensible File System for Hydra.* Technical Report, Carnegie-Mellon University, Computer Science Department, 1977.

[Alm80] Almes, G. T. *Garbage Collection in an Object Oriented System.* PhD thesis, Carnegie-Mellon University, Computer Science Department, June, 1980.

[Amd64] Amdahl, G., G. Blaauw, and F. Brooks. Architecture of the IBM System/360. *IBM Journal of Research and Development* 8:87-101, April, 1964.

[Bal76] Ball, J. E., J. A. Feldman, J. R. Low, R. F. Rashid, and P. D. Rovner. RIG, Rochester's Intelligent Gateway: System overview. *IEEE Transactions on Software Engineering* 2(4):321-328, December, 1976.

[Bau78] Baudet, G. M. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Carnegie-Mellon University, Computer Science Department, April, 1978.

[Bha73] Bhandarkar, D., and S. H. Fuller. Markov Chain Models for Analyzing Memory Interference in Multiprocessor Systems. In *First Annual ACM/IEEE Symposium on Computer Architecture.* ACM, December, 1973.

[Bob72] Bobrow, D. G., et al. TENEX, a Paged Time Sharing System for the PDP-10. *Communications of the ACM* 15(3):135-143, March, 1972.

[Bri70] Brinch Hansen, P. The Nucleus of a Multiprogramming System. *Communications of the ACM* 13(4):238ff, April, 1970.

[Bri71] Brinch Hansen, P. *RC4000 Software Multiprogramming System.* A/S Regnecentralen, Copenhagen, 1971. Second Edition.

[Bri75] Brinch Hansen, P. A Programming Methodology for Operating System Design. In *Proceedings of the 1975 International Conference on Reliable Software.* IEEE, 1975.

[Bri78] Brinch Hansen, P. *Concurrent Pascal.* Prentice-Hall, New York, 1978.

[Bro75] Brooks, F. P. Jr. *The Mythical Man-Month.* Addison-Wesley Publishing Co., 1975.

[Cai72] Bell, C. G., and P. Freeman. C.ai—A Computer Architecture for AI Research. In *1972 Fall Joint Computer Conference,* pages 779-790. AFIPS Press, 1972.

[Che79] Cheriton, D. R., et al. Thoth, a Portable Real-Time Operating System. *Communications of the ACM* 22(2):105-15, February, 1979.

[Coh75] Cohen, E. and D. Jefferson. Protection in the Hydra Operating System. In *Proceedings of the 5th Symposium on Operating System Principles,* pages 141-160. November, 1975.

[Dah66] Dahl, O.-J., and K. Nygaard. Simula—An Algol-Based Simulation Language. *Communications of the ACM* 9(9):671ff, September, 1966.

[Dah68] Dahl, O.-J. *Simula 67 Common Base Language.* Technical Report, Norwegian Computing Center, Oslo, 1968.

[DDH74] Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, London, 1974.

[DEC73] *PDP-11/05/10/35/40 Processor Handbook* Digital Equipment Corporation, Maynard, MA, 1973.

[Den66] Dennis, J., and E. Van Horn. Programming Semantics for Multiprogrammed Systems. *Communications of the ACM* 9(3):143ff, March, 1966.

[Den70] Denning, P. J. Virtual Memory. *Computing Surveys* 2(3), September, 1970.

[Dij68] Dijkstra, E. W. Cooperating Sequential Processes. In F. Genuys (editor), *Programming Languages*, . Academic Press, New York, 1968.

[Eng74] England, D. M. Capability Concept Mechanisms and Structure in System 250. In *IRIA Workshop: Protection in Operating Systems.* August, 1974.

[Ens77] Enslow, P. H. Multiprocessor Organization—A Survey. *Computing Surveys* 9(1):103-129, March, 1977.

[Fab74] Fabry, R. S. Capability-based Addressing. *Communications of the ACM* 17(7):403-412, July, 1974.

[Fer74] Ferrie, J., et al. An Extensible Structure for Protected Systems' Design. In *IRIA Workshop: Protection in Operating Systems.* August, 1974.

[Ful73] Fuller, S. H., R. Swan, and W. A. Wulf. The Instrumentation of C.mmp: A Multi-mini Processor. *IEEE Compcon* , 1973.

[Ful78] Fuller, S. H., and S. P. Harbison. *The C.mmp Multiprocessor.* Technical Report, Carnegie-Mellon University, Computer Science Department, 1978.

[Gra72] Graham, G. S., and P. J. Denning. Protection—principles and practice. In *1972 Spring Joint Computer Conference*, pages 417-424. AFIPS Press, 1972.

[Gum78] Gumpertz, R. *The Hydra Users Library.* Technical Report, Carnegie-Mellon University, Computer Science Department, 1978.

[Gut78] Guttag, J. V., E. Horowitz, and D. R. Musser. The Design of Data Type Specifications. In R. T. Yeh (editor), *Current Trends in Programming Methodology*, pages 60-79. Prentice-Hall, 1978.

[Gut80] Guttag, J. V. Notes on Type Abstraction (Version 2). *IEEE Transactions on Software Engineering* 6(1):13, January, 1980.

[Hea73] Heart, F. E., et al. A New Minicomputer/Multiprocessor for the ARPA Network. In *1973 National Computer Conference.* AFIPS Press, 1973. Volume 42.

[Hea75] Heart, F. E. The ARPA Network. In Grinsdale, R. L. and F. E. Kuo (editors), *Computer Communication Networks*, pages 19-33. Noordhoff Int. Publ., 1975.

[Hoa74] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM* 17(10):549ff, October, 1974.

[Ich79] Ichbiah, J. D., et al. Preliminary ADA Reference Manual. *ACM SIGPLAN Notices* 14(6A), June, 1979.

[Jai79] Jain, N. *Performance Study of Synchronization Mechanisms in a Multiprocessor.* PhD thesis, Carnegie-Mellon University, Computer Science Department, 1979.

[Jen76] Jensen, K., and N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, New York, 1976. *Lecture Notes in Computer Science*, Vol. 18.

[JL75] Jones, A. K., and R. Lipton. The Enforcement of Security Policies for Computation. In *Proceedings of the 5th Symposium on Operating System Principles.* ACM, November, 1975.

[Jon73] Jones, A. K. *Protection in Programmed Systems.* PhD thesis, Carnegie-Mellon University, Computer Science Department, June, 1973.

[Jon75] Jones, A. K., and W. A. Wulf. Toward the design of secure systems. *Software: Practice and Experience* 5:321-333, 1975.

[Jon76] Jones, A. K. and B. H. Liskov. A Language Extension for Controlling Access to Shared Data. *IEEE Transactions on Software Engineering* SE-2(4):277-285, December, 1976.

[Jon79] Jones, A. K., et al. STAROS—A Multiprocessor Operating System for Implementing Task Forces. In *Proceedings of the 7th Symposium on Operating System Principles.* ACM-SIGOPS, Pacific Grove, CA, 1979.

[Kat78] Katsuki, D. et al. Pluribus—An Operational Fault-Tolerant Multiprocessor. *Proceedings IEEE* 66(10), October, 1978.

[Kun76] Kung, H. T. Synchronized and Asynchronous Parellel Algorithms for Multiprocessors. In J.F. Traub (editor), *Algorithms and Complexity: Recent Results and New Directions,* . Addison-Wesley, 1976.

[Lam69] Lampson, B. W. Dynamic Protection Structures. In *1969 Fall Joint Computer Conference,* pages 27-38. AFIPS Press, 1969.

[Lam73] Lampson, B. W. A Note on the Confinement Problem. *Communications of the ACM* 16(10):613ff, October, 1973.

[Lam74] Lampson, B. W. Protection. *Operating Systems Review* 8(1), January, 1974.

[Lam76] Lampson, B. W., and H. E. Sturgis. Reflections on an Operating System Design. *Communications of the ACM* 19(5):251-265, May, 1976.

[Lam80] Lampson, B., and Sturgis, H. E. Crash Recovery in a Distributed Data Storage System. *Communications of the ACM.* To appear.

[Lau79] Lauer, H. C., and R. Needham. On the duality of operating system structures. In D. Lanciaux (editor), *Operating Systems,* North Holland Publishing Co., Amsterdam, 1979. Reprinted in *Operating Systems Review* 13(2), April 1979.

[Lin76] Linden, T. A. Operating System Structures to Support Security and Reliable Software. *Computing Surveys* 8(4):409-445, December, 1976.

[Lip75] Lipner, S. B. A comment on the confinement problem. *Operating Systems Review* 6(5):192-196, November, 1975.

[Lis75] Liskov, B. H. and S. N. Zilles. Specification Techniques for Data Abstractions. *IEEE Transactions on Software Engineering* SE-1, March, 1975.

[Lis77] Liskov, B. H., et al. Abstraction Mechanisms in CLU. *Communications of the ACM* 20(8):564-576, August, 1977.

[Lis79] Liskov, B. H., et al. *CLU Reference Manual.* Technical Report TR-225, MIT Laboratory for Computer Science, October, 1979.

[Lon75] London, R. L. A View of Program Verification. In *Proceedings of the International Conference on Reliable Software,* pages 534-545. IEEE Computer Society, April, 1975.

[Low77] Lowerre, B. *The HARPY Speech Recognition System.* PhD thesis, Carnegie-Mellon University, Computer Science Department, 1977.

[Mar77] Marathe, M., and S. H. Fuller. A study of multiprocessor contention for shared data in C.mmp. ACM SIGMETRICS Conference. Washington, D.C., December, 1977.

[McC73] McCredie, J. W. *Analytic Models of Time-shared Computer Systems: New Results, Validations, and Uses.* PhD thesis, Carnegie-Mellon University, Computer Science Department, 1973.

[McG80] McGehearty, P. *Performance Evaluation of a Multiprocessor Under Interactive Workload.* PhD thesis, Carnegie-Mellon University, Computer Science Department, To appear.

[Mor73] Morris, J. H. Protection in Programming Languages. *Communications of the ACM* 16(1):15-21, January, 1973.

[New77] Newcomer, J., et al. *Hydra: Basic Kernel Reference Manual.* Technical Report, Carnegie-Mellon University, Computer Science Department, 1976.

[Ole77] Oleinick, P. H., and S. H. Fuller. *The Implementation and Evaluation of a Parallel Algorithm on C.mmp.* Technical Report, Carnegie-Mellon University, Computer Science Department, December, 1977.

[Ole78] Oleinick, P. H. *The Implementation and Evaluation of Parallel Algorithms on C.mmp.* PhD thesis, Carnegie-Mellon University, Computer Science Depart-

ment, November, 1978.

[Orn75] Ornstein, et al. Pluribus: A reliable multiprocessor. In *1975 National Computer Conference.* AFIPS Press, 1975.

[Par71] Parnas, D. L. Information Distribution Aspects of Design Methodology. In *Proceedings of IFIP Congress,* pages 26-30. IFIP, 1971. Booklet TA-3.

[Par72a] Parnas, D. L. A Technique for Software Module Specification with Examples. *Communications of the ACM* 15:330-336, May, 1972.

[Par72b] Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM* 15(12), December, 1972.

[Red74] Redell, D., and R. Fabry. Selective Revocation of Capabilities. International Workshop on Protection in Operating Systems. IRIA, 1974.

[Rei75] Reid, B. K. and Newcomer, J. (ed). *The Hydra Songbook—A Vigilante User's Manual.* Technical Report, Carnegie-Mellon University, Computer Science Department, 1975.

[Rit74] Ritchie, D.M. and K. Thompson. The UNIX Time Sharing System. *Communications of the ACM* 17(7):365-75, July, 1974.

[Rot73] Rotenberg, L. *Making computers keep secrets.* Technical Report TR-115, MIT Project MAC, 1973.

[Sal74] Saltzer, J. H. Protection and the Control of Information Sharing in Multics. *Communications of the ACM* 17(7):388-402, July, 1974.

[Sal75] Saltzer, J. H., and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings IEEE,* pages 1238-1308. September, 1975.

[Sch72] Schroeder, M. D. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility.* PhD thesis, Massachusetts Institute of Technology, September, 1972. MAC TR-104.

[Sno80] Snodgrass, R. *COLA—A Command Language for Hydra.* Technical Report, Carnegie-Mellon University, Computer Science Department, 1980.

[Str70] Strecker, W. *Analysis of the Instruction Execution Rate in Certain Computer Systems.* PhD thesis, Carnegie-Mellon University, Computer Science Department, 1970.

[Stu74] Sturgis, H. E. *A Post-Mortem for a Time Sharing System.* Technical Report CSL 74-1, Xerox Palo Alto Research Center, January, 1974.

[Swa77] Swan, R., S. H. Fuller, and D. Siewiorek. CM*—A modular, multi-microcomputer. In *1977 National Computer Conference,* pages 637-644. AFIPS Press, 1977.

[Wal79] Walker, B., R. Kemmerer, and G. Popek. Specification and Verification of the UCLA Unix Security Kernel. In *Proceedings of the Seventh Symposium on Operating System Principles.* ACM, December, 1979.

[Wei76] Weinstock, C. B. *Dynamic Storage Allocation Techniques.* PhD thesis, Carnegie-Mellon University, Computer Science Department, April, 1976.

[Wil79] Wilkes, M. V., and R. M. Needham. *The Cambridge CAP Computer and its Operating System.* Elsevier/North-Holland Publishing Co., New York, 1979.

[Wir76] Wirth, N. *Modula—A Language for Modular Programs.* Technical Report, Technische Hochschule Zurich, March, 1976.

[Wul71] Wulf, W. A., et al. Bliss: A Language for Systems Programming. *Communications of the ACM* 14(12):780ff, December, 1971.

[Wul74] Wulf, W. A. *Alphard: Toward a Language to Support Structured Programs.* Technical Report, Carnegie-Mellon University, Computer Science Department, 1974.

[Wul75] Wulf, W. A. et al. Overview of the Hydra Operating System. In *Proceedings of the 5th Symposium on Operating System Principles,* pages 122-131. ACM, November, 1975. Austin, Texas.

[Wul76] Wulf, W. A., R. L. London, and M. Shaw. An Introduction to the Construction and Verification of Alphard Programs. *IEEE Transactions on Software Engineering* SE-2(4), December, 1976.

[Wul78] Wulf, W. A., and S. P. Harbison. Reflections in a Pool of Processors: An Experience Report on C.mmp. In *1978 National Computer Conference.* AFIPS Press, 1978.

INDEX

$Type$, 45, 73, 78

Abstract data types, 37–38, 43, 52, 174, 184
Abstraction, 37–38
Access control lists, 61
Access rights, 52–53
  allowed-rights, 46
  amplification, 53, 107
  AppendCapaRts, 66
  AppendDataRts, 66
  auxiliary rights, 46, 65
  capability rights, 65, 66
  ChangeTypeRts, 78
  C-list rights, 66
  ConnectRts, 96
  CopyRts, 66, 73
  CreateRts, 66, 73
  data-part rights, 66
  DeleteRts, 66, 108
  EnvRts, 66, 108, 112, 115
  generic, 46, 55
  GetCapaRts, 66, 108
  GetDataRts, 66
  implementation, 55–58, 81
  kernel rights, 46, 65, 108–109
  KillRts, 66
  ModifyRts, 66, 108, 110, 124, 190
  new-rights, 53
  object rights, 65
  propagation, 107
  PutCapaRts, 66, 110
  PutDataRts, 66, 110
  required-rights, 53
  restriction rights, 66
  retrospective, 88, 116
  TemplateRts, 78
  UncfRts, 66, 108, 113, 115, 125
  WriteRts, 190
Accounting, 161, 279
Active fixed-part directory, 154
Active GST, 153
Ada (programming language), 42
  for I/O, 200
  processor modifications for, 13
  (*see also* Relocation)
Address space, 13
Algol'68, 8
Allowed-rights, 46
Almes, G., 263, 268
Alphard (programming language), 42
Amplification, 53, 107
  restrictions, 110, 112
  retrospective, 59
  template, 53
AppendCapa, 72
AppendCapaRts, 66
AppendData, 71
AppendDataRts, 66
ARPANET, 17, 103, 129–135, 264
  connections, 130, 134–135
  HOST-HOST communication, 133–134
  IMP-HOST communication, 131–133
  links, 130
  NCP, 129–135
  protocols, 129–131
  sockets, 130
ASP (IBM), 3
Asynchronous Gauss-Sidel method (for PDE's), 236
Asynchronous Jacobi method (for PDE's), 235
AttachPolicy, 168
Authenticate, 142–144
Authority-based protection, 61
Autorestart, 212–213
Auxiliary rights, 46, 65
  ChangeTypeRts, 78
  ConnectRts, 96
  TemplateRts, 78
  WriteRts, 190

Backup (of GST), 161, 279
Baudet, G., 234

BBN Pluribus, 9, 19
Bhandakar, D., 239
Bliss, 7, 80, 101, 256, 260
  retrospective, 280
Box subsystem (example), 79–88
Brinch Hansen, P., 33
Brooks, F., xv
Burroughs D825, 3, 9

C (programming language), 8
C.ai, 3
Cache, 23–24, 241
Cacheable bit, 16, 24
Call, 44, 51, 74, 107
  definition, 55
  performance, 255–260
Call mechanism (*see Call*)
  retrospective, 61, 89, 278
Capability, 43, 46–47
  active form, 154
  capability rights, 65, 66
  passive form, 154
  retrospective, 278
Capability list (*see* C-list)
Catalogue System, 8, 48
CLength, 70
C-list, 47, 154
  rights, 66
Clock, time of day, 16, 17
Close, 123
CLU (programming language), 42
Cm*, 4, 9
Command language, 8, 80, 139, 143, 144
Compare, 71
Confinement problem, 108, 113–115, 120
Connect, 96
  ARPANET , 130, 134–135
  Message System, 92
Conservation problem, 108, 112–113
Contention, 4, 24, 202, 232, 239–243, 271, 275
  on locks, 271
  retrospective, 271
CONTINUE control function, 17
  cacheable, 16, 24
  dirty, 16, 193
  NXM, 16, 190
  write-protect, 16
Copy, 73
CopyRts, 66, 73
Cost of C.mmp, 25
CPSLoad, 189
CPSUnLoad, 189
Create, 45, 73
CreateMsg, 98
CreateRts, 66, 73
Creation template, 53, 73, 82
Crosspoint switch, 9, 14, 19–22, 275
  address parity, 20, 23
  components, 19
  contention, 4, 24, 202, 232, 239–243, 271, 275
  cost, 25
  implementation, 20
  priority resolution, 20, 241
  priority resolution performance, 242
  retrospective, 28

D825 (Burroughs), 3, 9
DARPA, 3, xvii
Data abstraction, 37–38, 43, 52, 174, 184
  retrospective, 184
DATA object, 78
Data-part, 47, 154
  rights, 66
DEC, 10
  PDP-10, 197, 261, 264
  PDP-10 performance comparison, 237, 238
Definition
  AppendCapa, 72
  AppendCapaRts, 66
  AppendData, 71
  AppendDataRts, 66
  AttachPolicy, 168
  Call, 44, 51, 55, 74
  capability, 43, 46–47
  CLength, 70
  Compare, 71
  Connect, 96
  Copy, 73
  CopyRts, 66, 73
  CPSLoad, 189
  CPSUnLoad, 189
  Create, 73
  CreateMsg, 98
  CreateRts, 66, 73
  Delete, 72
  DeleteRts, 66
  Disconnect, 96
  DLength, 70
  EnvRts, 66
  GetCapa, 71
  GetCapaRts, 66
  GetData, 71
  GetDataRts, 66
  GetMsgCapa, 98
  glitch, 27
  Hydra (object) types, 43, 45–46
  InterchangeCapa, 72
  KillRts, 66
  LNS, 44, 49–51

LNSLength, 70
MakeAmplificationTemplate, 76
MakeCreationTemplate, 73
MakePort, 96
MakeProcess, 168
MakeSimpleTemplate, 75
mechanism, 35
Merge, 53–55, 74
ModifyRts, 66
object, 43, 44–45
ObjInfo, 71
PassCapa, 72
path, 65
policy, 35
Policy object, 167–171
procedure, 44, 51–52
Process, 52
PutCapa, 71
PutCapaRts, 66
PutData, 71
PutDataRts, 66
PutMsgCapa, 98
ReadMsg, 98
ReceiveMsg, 99
ReceivePolicy, 169
ReplyMsg, 99
RequeueMsg, 100
Restrict, 72
Return, 51, 75
RPSLoad, 189
RSVPMsg, 99
RunTime, 170
SetSchedParms, 168
Start, 169
Stop, 169
TakeCapa, 72
template, 44
TypeCall, 76
UncfRts, 66
Update, 78
WriteMsg, 98
Delete, 72
DeleteRts, 66, 108
Demand paging, 187
GST, 157
paging, 193
reliability watchdogs, 212
retrospective, 185, 194
Dennis, J., 60
DESCAL (speech recognition task), 237
Device Allocation System, 148
DEVICE object, 131, 141, 196
Differential equations (parallel algorithms), 234
Digital Equipment Corporation (*see* DEC)
of active fixed-parts, 154
example, 48
of passive fixed-parts, 156
System, 148
(*see also* Catalogue System)
Dirty bit, 16, 193
Disconnect, 96
Disk (paging), 24
DLength, 70
DMA I/O, 200–201
Domain, 49
Duration (of kernel critical sections), 249

EnvRts, 66, 108, 112, 115
hardware, 23
retrospective, 213
software, 207, 209–210
tracking registers, 23
accumulator, 221
action, 221
definition, 221
Exception handling, 101

Fault tolerance, 87, 207, 208–209
File System, 8, 119–126
operations, 122–124
protection, 124–126
representation, 121
retrospective, 126
subfile operations, 123
subfiles, 120–122
synchronization, 121
Fixed-part, 154
Fork System, 148
Fortran, 8
Front End processor, 141

Garbage collection, 157
parallel, 158
Generic operations and rights, 46, 55
GetCapa, 71
GetCapaRts, 66, 108
GetData, 71
GetDataRts, 66
GetMsgCapa, 98
Glitch, 27
Global Symbol Table (*see* GST)
of Hydra, 31–41
retrospective, 58
of the Message System, 91
Graph structure of objects, 47
GST, 7, 153–160
active, 153
active GST performance, 264
demon, 157
fixed-part, 154

implementation, 153–160
lack of a backup mechanism, 161, 279
object creation rate, 265
passive, 153
passive GST performance, 268–271
performance, 160, 263–268
reliability, 159–160, 161
retrospective, 160

HALT control function, 17
cost, 25
error detection, 23
performance, 25–27
performance factors, 231
performance monitor, 220, 221
reliability, 27, 276
research issues, 4
retrospective, 27
technology of C.mmp, 25
HARPY, 237, 255
Hercules (script driver), 221, 229–230
History of Hydra/C.mmp, 6
Honeywell 645 (Multics), 9
HOST-HOST communication, 133–134
ASP, 3
OS/360, 61
System/360, 9, 31, 32, 197, 253
IMP-HOST communication, 131–133
of access rights, 55–58, 81
of I/O System, 198–202
of KMPS, 174–184
of objects, 154
of the GST, 153–160
of the Paging System, 191
Initialization problem, 108, 115–116
Input channel, 95
InterchangeCapa, 72
Interprocess communication, 91–104, 196
Interprocessor bus, 9, 16–17, 24
control functions, 17
interrupts, 16, 200, 275
interval timer, 16
time of day clock, 16, 17
Interprocessor interrupt, 16, 200, 275
Interval timer, 16
address mapping, 200
configuration tables, 198
direct memory access, 200–201
hardware, 195–196, 275
implementation, 198–202
kernel i/o, 202–203
reliability, 201–202
retrospective, 203
System, 195–203
user-level, 196–198
via the Message System, 93
IP bus (*see* Interprocessor bus)
IPI (*see* Interprocessor interrupt)

Jain, N., 245
JMON, 141
JOB System, 146–147
Jones, A., 7

K.mon, 220, 221
event accumulator, 221
event action, 221
event definition, 221
Kall (*see* Kernel Kalls)
Kernel I/O, 202–203
AppendCapa, 72
AppendData, 71
AttachPolicy, 168
Call, 44, 51, 74
CLength, 70
Compare, 71
Connect, 96
Copy, 73
CPSLoad, 189
CPSUnLoad, 189
Create, 73
CreateMsg, 98
Delete, 72
Disconnect, 96
DLength, 70
GetCapa, 71
GetData, 71
GetMsgCapa, 98
InterchangeCapa, 72
LNSLength, 70
MakeAmplificationTemplate, 76
MakeCreationTemplate, 73
MakePort, 96
MakeProcess, 168
MakeSimpleTemplate, 75
Merge, 53–55, 74
notation, 68
ObjInfo, 71
PassCapa, 72
performance, 255–260
PutCapa, 71
PutData, 71
PutMsgCapa, 98
ReadMsg, 98
ReceiveMsg, 99
ReceivePolicy, 169
ReplyMsg, 99
RequeueMsg, 100

Restrict, 72
Return, 51, 75
RPSLoad, 189
RSVPMsg, 99
RunTime, 170
SetSchedParms, 168
Start, 169
Stop, 169
TakeCapa, 72
TypeCall, 76
Update, 78
WriteMsg, 98
Kernel Multiprogramming System (*see* KMPS)
Kernel object types, 78–79
PORT, 78
CPS, 78, 188–190
DATA, 78
DEVICE, 78
LNS, 44, 49–51
PAGE, 78, 188–190
POLICY, 78, 167–171
PROCESS, 78, 167–171
RPS, 78, 188–190
SEMAPHORE, 78, 172, 173
TYPE, 45, 73, 78
UNIVERSAL, 78
Kernel properties, 32, 33, 41
Kernel rights, 46, 65, 108–109
AppendCapaRts, 66
AppendDataRts, 66
CopyRts, 66, 73
CreateRts, 66, 73
DeleteRts, 66, 108
EnvRts, 66, 108, 112, 115
GetCapaRts, 66, 108
GetDataRts, 66
KillRts, 66
ModifyRts, 66, 108, 110, 124, 190
PutCapaRts, 66, 110
PutDataRts, 66, 110
retrospective, 88, 116
UncfRts, 66, 108, 113, 115, 125
Kernel semaphores, 172
performance, 246–249
timing data, 246
Kernel tracer, 220, 223–228
example output, 225
performance, 224
KillRts, 66
KMPS, 7, 36, 163–184
implementation, 174–184
interaction with paging, 190–191, 194
kernel semaphores, 172
Lock operation, 172
LockModule, 175
locks, 172
number of slices, 164, 165
P operation, 173
parameters, 164–167
priority, 164, 165
ProcessModule, 175
processor mask, 164, 166
ProcessorModule, 175
QueueModule, 175
retrospective, 184
SchedulerModule, 176
SEMAPHORE objects, 172
SemaphoreModule, 176
stop message, 169
synchronization mechanisms, 171–174
time slice limit, 164, 165
Unlock operation, 172
V operation, 173
wait time, 164, 166, 174, 233, 249
working set limit, 164, 190

L*, 8, 256
Languages
Ada, 42
Algol 68, 8
Alphard, 42
Bliss, 7, 80, 101, 256, 260
C, 8
CLU, 42
command language, 8, 80, 139, 143, 144
Fortran, 8
L*, 8, 256
notion of type, 37
Pascal, 80, 174
PL/1, 101
Simula, 37, 42
Links, 130
LNS, 44, 49–51, 55
LNSLength, 70
Local name space, 44, 49–51
Lock operation, 172
LockModule, 175
Locks, 172
contention, 271
number used, 244
performance, 243–245
retrospective, 277
timing data, 277
Logging in, 142–144
Logging out, 144–145

MakeAmplificationTemplate, 76
MakeCreationTemplate, 73

MakePort, 96
MakeProcess, 168
MakeSimpleTemplate, 75
Marathe, M., 239, 243, 253
McCredie, J., 239
McGehearty, P., 241, 255, 262
Memory contention, 4, 24, 202, 232, 239–243, 271, 275
Memory-processor switch (*see* Crosspoint switch)
Merge, 53–55, 74
Message System, 91–104, 119, 131, 196
  connections, 92
  goals, 91
  I/O, 93
  message types, 92, 97
  messages, 92
  replies, 93
  reply mask, 97
Message type, 92, 97
Modification problem, 108, 110, 120
ModifyRts, 66, 108, 110, 124, 190
Monitor (*see* K.mon)
Multics, 9, 60
Mutual suspicion problem, 108, 109–110

NCP, 129–135
  ARPANET protocols, 129–131
  retrospective, 135
Needham, R., 279
Network Control Program (*see* NCP)
New-rights, 53
Non-hierarchical protection, 109
  for kernel Kalls, 68
  for object graphs, 63–65
  paths, 65
NXM, 16, 190
  control bits, 16, 190
  trap, 23

Objects, 43, 44–45
  C-list, 47, 154
  creation rate, 265
  data-part, 47, 154
  definition, 43, 44–45
  fixed-part, 154
  graph structure, 47
  implementation, 154
  locking performance, 265
  number in system, 265, 268
  number of types, 265, 268
  observed reference counts, 265, 269
  representation, 45, 47–49, 154
  retrospective, 272, 278
  rights, 65
  sizes, 265
  type, 45
  (*see also* GST)
  (*see also* Kernel object types)
ObjInfo, 71
Oleinick, P., 230, 237
ONR, xvii
OpenForRead, 122
OpenForWrite, 122
Output channel, 95

P operation, 173
Page frame, 188–190
  disk (drum), 24
Paging System, 187–194
  implementation, 191
  interaction with scheduling, 190–191, 194
  paging demon, 193
  replacement policy, 193–194
  retrospective, 194
  state of a page, 191
  asynchronous Gauss-Sidel method, 236
  asynchronous Jacobi method, 235
  HARPY, 237, 255
  partial differential equations, 234
  purely asynchronous method, 236
  RootFinder, 230
Parallel garbage collection, 158
  in Hydra kernel, 40–41, 163–184
  in user programs, 40–41, 91
  (*see also* Parallel algorithms)
Parameter template, 54
  (*see also* amplification template)
  (*see also* simple template)
  address, 20, 23
  memory, 23
Pascal, 80, 174
PassCapa, 72
Passive fixed-part directory, 156
Passive GST, 153
Path, 49, 65
  definition, 65
  notation, 65
  pretarget, 65
  steps, 65
  target, 65
PDP-10, 197, 261, 264
  performance comparison, 237, 238
PDP-11/40E, 8, 9, 10
Performance, 219–273
  active GST, 264
  basic hardware, 25–27
  crosspoint switch priority resolution, 242
  duration of kernel critical sections, 249
  error detection/recovery, 214

experiments, 230–271
GST, 160, 263–268
hardware factors, 231
hardware monitor, 220, 221
inter-communication intervals, 252
inter-synchronization intervals, 249
Kall frequency, 255–260
Kall timings, 255–260
kernel locks, 243–245
kernel semaphores, 246–249
kernel size, 260–262
kernel tracer, 220, 223–228
lock timing data, 277
number of objects, 265, 268
number of types, 265, 268
object creation rate, 265
object locking, 265
object sizes, 265
passive GST, 268–271
Port timing data, 250
Ports, 245–253, 271
RootFinder, 231, 233
script driver, 221, 229–230
SEMAPHORES, 245–253
small-address effect, 253–255
snapshot taker, 220, 228–229
stretch factor, 262–263
synchronization mechanisms, 233, 243–253, 272
timing of kernel semaphores, 246
tools, 220–230
variations, 231
PL/1, 101
Pluribus (BBN), 9, 19
PM (*see* Policy Module)
Policy Module, 8, 36, 144, 163
retrospective, 59, 272
stop message, 169
(*see also* KMPS)
Policy object, 167–171
Policy/mechanism separation, 33, 35, 41, 163, 195, 272
mechanism definition, 35
policy definition, 35
retrospective, 59, 272
Port System (*see* Message System)
Ports, 91–104
input channel, 95
message slots, 95
output channel, 95
performance, 245–253, 271
PORT objects, 92
timing data, 250
Pretarget (in a path), 65
Procedure, 44, 51–52, 55
protection, 52–53, 107
Process, 52
ProcessModule, 175
Processor modifications, 13, 22–23
address mapping, 13
stack protection, 13
ProcessorModule, 175
Productivity, 126
Propagation of access rights, 107
Properties of a kernel, 33, 41
at Procedure invocation, 52–53, 107
Hydra's philosophy, 34–35
non-hierarchical, 109
relation to abstract data types, 40, 52–53
retrospective, 116–117
and security, 58
in the File System, 124–126
use of the Hydra mechanisms, 107–117
what to protect, 39
Protection domain, 49
confinement, 108, 113–115, 120
conservation, 108, 112–113
initialization, 108, 115–116
modification, 108, 110, 120
mutual suspicion, 108, 109–110
other, 117
Purely asynchronous method (for PDE's), 236
PutCapa, 71
PutCapaRts, 66, 110
PutData, 71
PutDataRts, 66, 110
PutMsgCapa, 98

QueueModule, 175

ReadMsg, 98
ReceiveMsg, 99
ReceivePolicy, 169
Reference counts, 265, 269
address parity, 27, 276
autorestart, 212–213
crosspoint switch, 27
GST, 159–160, 161
hardware, 27, 276
I/O, 201–202
performance, 214
retrospective, 27, 161, 204, 213, 281
software, 207–213
software diagnostic mechanisms, 207, 210–211
software error detection, 207, 209–210
software fault tolerance, 207, 208–209
software recovery mechanisms, 207, 211–212
software techniques, 159–160
software validation, 207, 208

suspect-monitor, 212
watchdogs, 212
Relocation, 14, 15
cacheable bit, 16, 24
control bits, 16
dirty bit, 16, 193
for I/O, 200
NXM, 16, 190
write-protect, 16
Replacement policy, 193–194
Reply, 93
Reply mask, 97
ReplyMsg, 99
C-list, 47, 154
data-part, 47, 154
files, 121
Hydra objects, 45, 47–49, 154
RequeueMsg, 100
Required-rights, 53
hardware, 4
software, 4, 5
Restrict, 72
Restriction rights, 66
Retrospective
access rights, 88, 116
amplification, 59
Bliss, 280
capabilities, 61, 89, 278
contention, 271
crosspoint switch, 28
data abstraction, 184
demons, 185, 194
error detection, 213
File System, 126
GST, 160
hardware, 27
Hydra protection mechanism, 116–117
Hydra's goals, 58
implementation language, 277
I/O, 203
kernel facilities, 88–89
KMPS, 184
KMPS-PM interaction, 184, 194, 278
locks, 277
management, 280
Message System, 102–104
mistakes, 60, 149, 276
multiple synchronization mechanisms, 277
NCP, 135
object model, 278
objects, 272, 278
on the Retrospective, 281
paging, 194
policy modules, 59, 272
policy/mechanism separation, 59, 272
reliability, 27, 161, 213, 281
revocation, 279
scheduling, 185
small address problem, 27, 273
software reliability, 204
subsystems, 149
symmetric system, 277
synchronization, 272
tools, 271
type mechanism, 278
TypeCall, 89
use of processes, 277
user-level operating system, 149
Return, 51, 75
Revocation, 279
Rights (*see* Access Rights)
RootFinder, 230, 240, 244, 254
expected performance pattern, 230
performance, 231, 233
RPSLoad, 189
RSVPMsg, 99
RunTime, 170

SAP (*see* Small address problem)
SchedulerModule, 176
implementation, 180
Scheduling, 163–184
interaction with paging, 190–191, 194
medium-term, 163
parameters, 164–167
retrospective, 185
short-term, 163, 164
(*see also* KMPS)
Scheduling parameters, 164–167
number of slices, 164, 165
priority, 164, 165
processor mask, 164, 166
time slice limit, 164, 165
wait time, 164, 166, 174, 233, 249
working set limit, 164, 190
Script driver, 221, 229–230
Security, 58
SemaphoreModule, 176
implementation, 178
Kernel, 172
object type, 172
performance (kernel semaphores), 246–249
performance (SEMAPHORE objects), 245–253
SetSchedParms, 168
Simula, 37, 42
SIX12 (debugger), 8
Size of Hydra, 260–262
Small address problem, 14, 27, 187, 253–255, 261, 273, 275
retrospective, 27, 273

Snapshot taker, 220, 228–229
Socket, 130
Software research issues, 4, 5
SOS, 256
  DESCAL task, 237
  HARPY System, 237, 255
  information retrieval task, 237
Stack protection, 13
Start, 169
START control function, 17
State of a page object, 191
Steps (in a path), 65
Stop, 169
Stop message, 169
Strecker, W., 239
Stretch factor, 262–263
Structured programming, 37
SubfileClose, 123
SubfileRead, 123
Subfiles, 120–122
SubfileWrite, 123
Subsystems, 39, 41
  Box (example), 79–88
  Catalogue System, 8, 48
  device allocation, 148
  directories, 148
  example, 79–88
  File System, 8, 119–126
  Fork, 148
  GST, 7, 153–160
  I/O, 195–203
  JMON, 141
  JOB, 146–147
  KMPS, 7, 36, 163–184
  Message System, 91–104, 119, 131, 196
  Paging, 187–194
  Policy Modules, 8, 36, 144, 163
  retrospective, 149
  SOS, 256
  SYSMON, 149
  TECO, 256
  TMUX, 141
  UserToken, 142–144
Suspect-Monitor mode, 212
  examples, 87, 124
  mechanisms, 171–174
  object locking performance, 265
  performance, 233, 243–253, 272
  retrospective, 272, 277
SYSMON, 149

TakeCapa, 72
Target (in a path), 65
TECO, 256
Templates, 44, 55
  amplification, 53
  creation, 53, 73, 82
  parameter, 54
  simple, 53
  use with *Merge*, 53–55
Terminal multiplexor, 141
Terminal session (example), 139
Time-sharing performance (stretch factor), 262–263
TMUX, 141
Tools for performance measurement, 220–230
Tracking registers, 23
Traps (NXM), 23
Type
  abstract data types, 37–38, 43, 52, 174, 184
  definition facilities, 37
  Hydra definition mechanism, 73
  Hydra objects, 43, 45–46
  of messages, 92, 97
  number of types, 265, 268
  object sizes as a function of type, 265
  in programming languages, 37
  relation to protection, 40, 52–53
  retrospective on the Hydra mechanism, 278
  TYPE objects, 45, 73, 78
TypeCall, 76, 119, 144
  retrospective, 89

UncfRts, 66, 108, 113, 115, 125
Unique-name, 44, 46
UNIVERSAL object, 78
UNIX performance comparison, 237
Unlock operation, 172
Update, 78, 87

V operation, 173
Van Horn, E., 60
Virtual machine, 33