THE ROLE OF EXCEPTION MECHANISMS IN SOFTWARE SYSTEMS DESIGN

by

M. STELLA ATKINS

B.Sc. Nottingham University (England), 1966
M.Phil. Warwick University (England), 1976

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

October 1985

22

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 14 October 1985

Abstract

Exception handling is a crucial aspect of practical programming, particularly in systems allowing logical concurrency such as multi-process distributed systems.

First, a survey of existing exception handling mechanisms in operating systems is performed, which shows a diversity of implementations, depending on the process model and the method of inter-process communication. The thesis then develops a model for designing software which exploits the different mechanisms for handling normal and exceptional events. The model is applicable in many multi-process programming environments, and not only preserves modularity, but also enhances efficiency and reliability, while often increasing concurrency.

To derive such a model, exceptions in multi-process software are classified primarily according to the program level at which they are detected and handled. Server-to-client exceptions are of particular interest because of their ubiquity; these are exceptions detected by a server and handled by a client.

The model treats systems programs as event driven, and proposes dividing the events into normal or exceptional, according to the cost and mechanisms for handling them. Techniques are described for designing software according to three criteria: minimising the average run-time, minimising the exception processing time, and incrementally increasing the program's functionality. Many examples are given which illustrate the use of the general model.

Program paradigms in several languages and in several systems are introduced to model features which are system dependent, through illustrative examples for asynchronous i/o multiplexing, and for exception notification from a server to its client or clients.

Finally, some programs which have been implemented according to the rules of the model are described and compared with their more conventional counterparts. These programs illustrate the practicality and usefulness of the model for diverse systems and concurrent environments.

Table of Contents

Abstract ii

List of Tables vii

List of Figures viii

Acknowledgement ix

1 Introduction 1
1.1 Introduction 1
1.2 What is an Exception? 4
1.3 Goals and Scope of the Thesis 7
1.4 Motivation 9
1.5 Synopsis of the Thesis 10

2 Exceptions in Multi-process Operating Systems 12
2.1 Introduction 12
2.2 General Exception Mechanisms in Operating Systems 16
2.2.1 UNIX 16
2.2.1.1 UNIX Processes 16
2.2.1.2 UNIX Signals as Exception Mechanisms 17
2.2.1.3 Intra-process Exception Mechanisms 18
2.2.2 Thoth 19
2.2.2.1 Thoth Processes 19
2.2.2.2 Process Destruction as an Exception Mechanism 20
2.2.2.3 Intra-process Exceptions 20
2.2.3 PILOT 20
2.2.3.1 Processes and Monitors in PILOT 21
2.2.3.2 Intra-process Exceptions in PILOT 21
2.2.3.3 Inter-process Exceptions in PILOT 22
2.2.3.4 Exceptions within Exception Handlers 22
2.2.4 The Cambridge CAP System 23
2.2.4.1 CAP Processes and Protected Procedures 23
2.2.4.2 Intra-process Exceptions in CAP 25
2.2.5 RIG 26
2.2.5.1 Exceptions in RIG. 26

2.2.6 Medusa 28
2.2.6.1 Processes in Medusa 28
2.2.6.2 Exceptions in Medusa 28
2.2.6.3 Internal Exceptions in Medusa 29
2.2.7 Remote Procedure Call 30
2.3 Mechanisms for Interactive Debugging 31
2.4 Exception Handling in Action 34
2.4.1 Inter-process Cooperation 34
2.4.2 Asynchronous Events -- Terminal Interrupt 40
2.5 Summary 44

3 Principles for Improving Programs through Exception Mechanisms 46
3.1 Classification of Exceptions 46
3.1.1 Inter-process Exceptions 49
3.2 Model for Systems Design 52
3.2.1 Objective A: Minimise the Average Run-time 58
3.2.2 Objective B: Minimise the Exception Processing Time 61
3.2.3 Objective C: Increase the Functionality of the System. 63
3.3 Examples Illustrating Minimising the Average Run-time 65
3.3.1 Partition the Event Set into 2 Groups to Reduce Detection Costs 66
3.3.1.1 Spell Example 66
3.3.1.2 Putbyte Example 70
3.3.2 Reduce Handling Costs by Restructuring Programs 72
3.3.2.1 Os Example 72
3.3.2.2 Verex Name-server Example 74
3.3.2.3 Newcastle Connection Example 76
3.3.2.4 The TROFF/TBL/EQN Example 78
3.3.2.5 Point-to-point Network Protocol Example 81
3.3.3 Reduce Context-saving Costs in Normal Events 83
3.3.3.1 Read-back Example 83
3.3.3.2 LNTP Example 84
3.4 Examples Illustrating Increasing the Functionality of the System 87
3.4.1 Using Inter-process Exceptions from Server-client 87
3.4.1.1 ITI <BRK> Example 88
3.4.2 Ignorable Exceptions 88
3.4.2.1 ITI Reconnection Example 88
3.4.2.2 Window-size-change Example 89
3.4.2.3 Hash Table Example 90
3.4.2.4 Slow Graph Plotter Example 90
3.4.2.5 Mesa File-server Example 91

4 Program Paradigms for Implementing System Dependent Features 92
4.1 Program Paradigms for Inter-process Exception Notification 93
4.1.1 The KEYBOARD/MOUSE Example 93
4.1.2 Exception Notification in a Synchronous Message-passing System 94
4.1.3 Exception Notification in Procedure-oriented Concurrent Languages 97
4.1.4 Exception Notification in Message-oriented Concurrent Languages 99
4.1.5 Exception Notification in Operation-oriented Concurrent Languages 100
4.1.6 Exception Notification in Distributed Systems 101
4.2 Program Paradigms for Ignorable 1:many Exceptions 101
4.2.1 The Storage Allocator Example 101
4.2.2 Storage Allocator Using Unreliable Broadcast Notification 107
4.2.3 Storage Allocator in Mesa 110
4.2.4 Storage Allocator in Medusa 112
4.2.5 Storage Allocator in V-kernel 112
4.3 Exception Handlers as Processes: Mechanisms for Mutual Exclusion 117

5 Programs Illustrating the Model 123
5.1 Exceptions and Protocol Implementations -- a New Approach 125
5.1.1 The ITI Protocol 125
5.1.2 The Interrupt Protocol 127
5.1.3 The ITI Implementation in a Thoth-like Operating System 129
5.1.3.1 Notification of Inter-process Exception Using Process Destruction 130
5.1.3.2 Notification of Inter-process Exception Using a Toggle Switch 131
5.1.3.3 ITI as a Filter and an Exception Process 133
5.1.4 Problems with Synchronisation 136
5.1.5 Recovery after network failure 140
5.1.6 Summary 141
5.2 A Nested Transaction Mechanism for Atomic Data Updates 143
5.2.1 Introduction 143
5.2.2 Overview of the ARGUS Model 144
5.2.2.1 Atomic Objects 144
5.2.2.2 Nested Actions 145
5.2.2.3 Guardians 145
5.2.3 Implementation Considerations 147
5.2.3.1 Introduction 147
5.2.3.2 Requirements for correct implementation 148

5.2.4 Features of the Implementation 150
5.2.4.1 Transaction Invocation 151
5.2.4.2 Atomic Data Access 152
5.2.4.3 Subaction Commit 153
5.2.4.4 Top-level Commit 153
5.2.4.5 Transaction Aborts 154
5.2.4.6 Transaction Read/Write Conflicts 154
5.2.4.7 The Group Leader's Role 156
5.2.4.8 Group-Leader's TOP-COMMIT Algorithms. 157
5.2.4.9 Group Leader's ABORT Algorithms 161
5.2.5 Efficiency of Mechanisms 162

6 Conclusions 165
6.1 Summary of results 165
6.1.1 Characterize the Nature of Exceptions 166
6.1.2 Model for Systems Design 167
6.1.3 Program Paradigms for Systems Design 168
6.1.4 Program Examples 169
6.2 Further Work 169

References 172

Appendix A. The putbyte program 176

Appendix B. The os program 181

Appendix C. The TROFF/TBL/EQN system 187

List of Tables

3.1 Execution Costs for a General Event Manager 67

5.1 Number of SendReceiveReply Messages on <BRK> 131
5.2 Comparison of Locus and V-kernel Nested Atomic Actions 162

A.1 Execution Costs for the Putbyte Routine 177
B.1 Execution Costs for the Os Utility 183

List of Figures

2.1 Thoth Server-client Notification using a Victim Process 37

3.1 Level 1 Classification of Exceptions 46
3.2 A Classification of Inter-process Exceptions 49
3.3 Level 1 Model for Systems Design 55
3.4 Level 2 and 3 Model to Minimise the Average Run-time 59
3.5 Model to Minimise the Exception Processing Time 62
3.6 Model to Increase the Program's Functionality 64
3.7 Expected Execution Costs for Different Values of np and n 70
3.8 Restructuring the Verex Name-server to Reduce Ipc in the Normal Case 75
3.9 Restructuring Newcastle Connection to Reduce Ipc in the Normal Case 77

4.1 Thoth Server Notification Using Two Worker Processes 95
4.2 Thoth Switch Notification 96
4.3 Levin's Storage Allocator 103
4.4 A User of Levin's Storage Allocator 105
4.5 Storage Allocator Using a Monitor and Broadcast Exceptions 109
4.6 Pool-server in V-kernel 114
4.7 A User and Exception Handler of the V-kernel Storage Allocator 117

5.1 ITI Implementation as a Server 126
5.2 Network Packets to be Exchanged on an Interrupt 128
5.3 Process Switches to Handle an Interrupt Using Process Destruction 130
5.4 ITI as an I/O Filter 134
5.5 Process Switches on <BRK> with a Separate Exception Process 135
5.6 Formation of a Version Stack with Nested Transactions 155
5.7 Group Leader Communication Structure 156

B.1 The Initial Version of Os 182

C.1 Pipe Worker Processes 187
C.2 Message Traffic for Handling an Equation 189

Acknowledgements

Thanks to Son Vuong, my supervisor, for his support during the last few years, and to my thesis committee of Sam Chanson, Doug Dyment, Paul Gilmore and Mabo Ito, for their active encouragement and advice on the presentation of this thesis, and to Roy Levin for his constructive comments and criticisms.

Thanks to David Cheriton who started my investigation into exceptions, and who gave continued inspiration and advice in the first few years while he was at the University of British Columbia.

Thanks to the many graduate students and staff in the department who have helped me through discussions and criticisms, especially to Ravi Ravindran, Steve Deering and Vincent Manis, and to my colleagues Rachel Gelbart and Lisa Higham for their continued support.

Many thanks to Roger Needham and the staff and students at the Cambridge Computer Laboratory for their friendly support and use of their facilities, and thanks also to the staff at Simon Fraser University for letting me finish. Finally, thanks to Derek my husband, and to my children, for their patience.

CHAPTER 1

Introduction

1.1. Introduction

A crucial aspect of practical programming, particularly in systems allowing logical concurrency such as multi-process distributed systems, is exception handling. Over the last few years some operating system and programming language structures have been developed to aid in the handling of exceptional events. The developments are interrelated because of the recursive nature of computer systems: the operating system must be written in a programming language, and the program must be run by an operating system.

The various exception handling mechanisms that have been proposed for programming languages all aim to separate the common from the uncommon case and thus to help programmers separate their concerns, by preserving modularity. In general this is achieved by handling an exception at a higher level of abstraction from where it is detected. This is reflected in most embedded programming language mechanisms, where handlers for exceptions are searched in a path which is directed up the so-called *procedure calls hierarchy.* Of course, the burden of having to write the code for the uncommon case cannot be eliminated. The claimed methodological advantage of embedded exception handling mechanisms is the separation of such code from the main line program both in the time of its writing and in its location in the program text.

Similarly, in operating systems software, the facilities for handling exceptional events have been developed for modularity in programming. Many modern operating systems are

structured as a hierarchy of processes, where it is possible for a process to control subordinate processes while itself being subject to the control of another. For modularity it is desirable to provide the same exception handling facilities at every level in the hierarchy. However, in operating systems, performance and reliability are additionally crucial to success, and the exception mechanisms must also be designed to enhance the overall system performance and reliability. For example, both the user and the operating system require mechanisms for handling errors -- otherwise an erroneous program which never halted could only be stopped by switching off the power supply! In modern concurrent operating systems where a program may consist of several processes which may be running on separate processors, the need for such mechanisms is even more acute. As Nelson remarks in his thesis on Remote Procedure Calls [Nelson 81]: " Machinery for aborting processes and forcing asynchronous exceptions may seem somewhat violent to the gentle programmer of sequential abstractions. But realistic remote procedure environments and, indeed, realistic applications will in fact require this machinery to provide robust services and exploit the full performance of distributed systems."

The user requires a mechanism to allow him to terminate program execution arbitrarily, and such a mechanism is typically invoked by hitting the terminal <BRK> key. The operating system must respond to such an event by terminating the offending program, reclaiming the resources that were allocated to it, and generally attempting to return to the state which existed before the program was run. Such activity may be considered as an example of exception handling in operating systems.

This thesis studies not only the problems which designers of concurrent operating systems have to overcome in handling exceptional events such as the <BRK> described above, but also the problems of systems design exploiting exception mechanisms in general, in a

multi-process environment. The high level language exception mechanisms are usually engineered so that their use is economic only when they are rarely exercised; the very circumstance in which an error of logic is likely to remain undetected [Black 82]. Furthermore none of the high level language mechanisms extend to multiple processes (except for an exception ABORT); they are concerned with exceptions detected and handled *within* a process. The structures which *are* available for handling exceptional events in a multiple-process environment are specific to the particular environment; most are not portable to other systems, and they do not pretend to be of general use.

This research extends the development of linguistic exception mechanisms which deal with intra-process exceptions, to mechanisms in a multi-process environment; in particular to *event driven* programs used in operating systems. An event driven program is one which responds to events which may be generated externally to the computer system (such as by the user hitting the terminal <BRK> key), or by other processes within the computer system (such as an i/o completion event or a timer firing). Peterson and Silberschatz state that operating systems are event driven programs [Peterson 83], in that if there are no jobs to execute, no i/o devices to service, and no users to respond to, an operating system will sit quietly, waiting for something to happen.

We use the term *event driven* more widely, to include system servers (or monitors) which are structured hierarchically, with a client, or clients, making requests of a server below, which in turn may use the resources of another server (or may make a nested monitor call). We also view some data-driven programs as event driven programs -- in these cases, the events correspond to the value of the data received. The program units which respond to these events are called *event managers*. In some cases, the events occur randomly, with unk-

nown probabilities; in others, the probability of an event is deterministic. In event driven programs where the relative probabilities of the events are known, we can say that the event managers execute some small percentage of the code to handle the *normal* or *most probable* events, and the rest of the code is executed to handle the *unusual* events. Now it is computer folklore that 90% of most program code is executed 10% of the time. It is therefore important to structure such event managers so that the *normal* case is clearly differentiated from the *unusual* cases, so the programmer can separate his concerns, to writing efficient code for the normal cases which are executed most of the time[1].

This thesis develops principles for writing efficient event managers; principles which exploit different mechanisms for handling normal and exceptional events. The principles are used to derive a model for designing software, applicable in multi-process programming environments, which not only preserves modularity, but also enhances efficiency and reliability, and often increases concurrency.

1.2. What is an Exception?

Exceptions have often been considered as *errors*, although most designers of exception handling mechanisms claim that their mechanisms, while suitable for dealing with errors, are also valuable for other events. Unfortunately this guideline is of little use because the term *error* has as many interpretations as *exception*.

Black devotes a Chapter in his thesis *Exception Handling -- the case against* [Black 82], to the topic *What are Exceptions?* He tries out phrases like *an exception occurs when a routine cannot complete the task it is designed to perform* (from the CLU reference manual

[1]But note that in systems where the probability of an event is not known *a priori*, such structuring may not be possible.

[Liskov 78]), *inconvenient results of an operation* (from [Goodenough 75]), *errors that must be detected at run-time,* (from the ADA definition [US 83]), and *means of indicating implementation insufficiency* [Black 82] and finally observes that none of these definitions describe an abstract object called *exception,* which can be neatly formalised.

In his thesis *Program Structures for Exceptional Condition Handling,* Levin [Levin 77] rejects any connotation of *error* by *exception,* but he also rejects the common guideline that an exception is a rarely occurring event, as the frequency of many events depends on context. For example, looking up a name in a compiler's symbol table may yield two results: name present or name absent. Which of these is more frequent depends on context. Why should one result be considered exceptional, when the other is not? Levin's solution to the dilemma is to treat both results as an exception.

The dictionary defines the word **exception** in a manner which captures the essence of the notion of unusual case: i.e. " exception - a person, thing, or case to which the general rule is not applicable" -- and this is the spirit in which we use it, strenuously avoiding the connotations it has as a synonym for *error,* observing that *errors* are a strict subset of *exceptions.* (At least, we hope that errors are *unusual events*!). We therefore define an exception as a rare event which is treated differently to a normal event.

This thesis claims that if an event can be defined as being unusual in most contexts, then it can be considered an exceptional event. This definition agrees closely with Levin's for the problems presented in this thesis. However, it is accepted that in some problems where the context changes, the exception events could become the norm. For example, a system resource-manager may receive one request for almost all its resources, and for the next week it may be running in the *exceptional mode* of being short of resources. If this *situation* is judged

to be exceptional, then it can still be claimed that there are enough resources for most requests in the *normal mode* of operation for that resource-manager. If however, the probabilities of the events are unknown or very variable, then the best solution may be to treat all events in the same way, and many of the design proposals in this thesis are not applicable in such situations.

It transpires that there are several cases encountered in computing practices, particularly in operating systems, which are not generally considered as exceptions, but which also fall into this category under the dictionary definition. For example, program initialisation code which is executed only once and hence usually does not follow the general rule. Similarly, opening a file before it can be read or written is an essential part of the program execution, but the execution of this event occurs with a low frequency compared with execution of other events on the file, such as reading and writing to the opened file.

An *exception mechanism* is composed of three parts: *detection* (which may be by hardware or software), *notification* and *handling*. *Notification* is the act of signalling the occurrence of an exception to interested parties; in a multi-process system one or more processes may be involved. Together, the detection and notification may be considered as drawing attention to the exception event. *Exception handling* is defined as the execution of some actions in response to the exception notification. All these components of exception mechanisms are discussed in detail in the thesis.

When, then, should we use exceptions? For example, when the End-of-File has been read, is it treated as an exception or as a normal, but relatively infrequent event? There is no clear demarcation between these viewpoints, but they may require different programming styles, and may be implemented by different mechanisms which may incur different overheads

at compile-time and run-time. Our research addresses these issues in detail, and shows that generally an exception-handling mechanism should be used when a programmer wishes to improve clarity of the normal-case computation, provided it does not incur unreasonable costs or impair reliability. By using an exception mechanism we attempt to reduce cost by making the normal case simpler and faster[2].

1.3. Goals and Scope of the Thesis

The thesis of this dissertation is that designing systems software to exploit the use of exception mechanisms leads to efficient modular programs which are also easy to write and to understand. Our goals are to show how to design efficient and modular systems programs exploiting exception mechanisms.

This thesis does *not* include discussion of hardware reliability; this topic has been adequately reviewed in another survey paper [Randell 78]. We do not discuss in detail the exception mechanisms embedded in high level languages, as Black has covered this in his thesis, although these mechanisms are referred to throughout the text. Issues of proof of program correctness are also considered beyond the scope of this thesis, although correctness issues are referenced whenever applicable.

To achieve the goals some general principles for exception mechanisms are needed -- principles which have universal application across languages and systems. The systems we consider are event driven, where the relative probabilities of the events are known *a priori*. In such systems it is possible to define what is an exceptional event. But context is not the only factor in designating what is an exceptional event -- the language/system in which

[2]again with due regard to the fact that should the operating context change, the previous exception events may become normal case with corresponding changes (sometimes a dramatic increase) in run-time.

computations are expressed also influences what is an exception. Therefore the languages/systems in which exceptional events occur must also be considered. Only then can generalities be made about what constitutes an exceptional event, and how best to handle it. Thus the first goal is to *characterize the nature of exceptions.* We do this by considering a wide range of systems problems from different areas of systems design, in particular, event managers such as system servers, communication protocols and data-driven system programs[3]. A general classification of exceptions is then derived.

The next goal is to provide a model for designing efficient and modular software in event driven programs in a multi-process environment, by exploiting exception mechanisms. This goal is achieved by considering three different criteria for program design: minimising the average run-time, minimising the exception-handling time, or increasing the program's functionality in a modular way. A model is derived which provides system design guidelines for these three criteria.

Because of the diversity of languages/systems, the final goal is to provide program paradigms which employ the design principles propounded in the general model, in particular for the system dependent features of the model. The program paradigms may then be used as models for implementation in different environments. To fulfill this goal, several programs are described in detail, and two of the design models which have been implemented together with their exception-handling mechanisms are described to show their applicability.

The operating system Verex [Lockhart 79], a derivative of Thoth [Cheriton 79a] was used to perform some of the research, and also TRIPOS [Richards 79b], implemented on the Cambridge Ring local area network [Wilkes 80]. Later, the UNIX 4.2BSD operating system

[3]Note that we do not discuss in detail the inner kernel features which are driven by hardware interrupts; our research is concerned at a higher level of abstraction than that provided by a raw kernel.

[Joy 83] was used, implemented on VAX computers and SUN workstations connected over an Ethernet local area network [Metcalfe 76]. Thus the designs were tested on different real systems, to show how requirements of adequacy and practicality were met.

1.4. Motivation

The major motivation is that problems are encountered in managing the numbers and types of special cases and unusual circumstances when writing software for concurrent systems, and there are few, if any, answers to these problems.

As programs grow in size and complexity, special cases and unusual circumstances are compounded. Controlling and checking for exceptions takes more and more effort. With the advent of parallel programming, distributed systems and computer networks, not only the *number* of exceptions increases, but also the *type* of possible exceptions. Exceptions can be within sequentially executed programs, between tasks sharing memory but operating in parallel, and between tasks executing in parallel which do not share any memory.

Research on this subject is needed because there are very few, if any, exception-handling techniques implemented for multi-process programs that preserve structural clarity, and that have simply defined semantics so that the techniques are easy to use and understand. In support of this claim, the authors of one multi-process system, PILOT [Lampson 80] written in the high level language Mesa [Mitchell 79] comment that:

> Aside from the general problems of exception-handling in a concurrent environment, we have experienced some difficulties due to the specific interactions of Mesa signals with processes and monitors. In particular, the reasonable and consistent handling of signals (including UNWINDS) in entry procedures represents a considerable increase in the mental overhead involved in designing a new monitor or understanding an existing one.

Motivation from my own experiences came when writing software for Verex, a multiple-process operating system. It was desirable to trap errors in programs so that they could then

be conveniently debugged instead of immediately destroyed. An error-handling mechanism which optionally diverted program termination by handing control to the interactive debugger was implemented by the author. It transpired that the mechanism could be used to handle more than erroneous situations. In considering just how and where the mechanism would be used, many problems of exception-handling had to be addressed. For example, should the program unit raising the exception be allowed to resume from the place where the exception was raised, or should the exception handler terminate execution of the current program unit? Is it desirable, or even feasible, to use the same mechanism to handle exceptions in sequential code, exceptions between parallel tasks sharing memory, and between parallel tasks in a distributed system which have no shared memory? How does interactive debugging fit into the scheme? It is the aim of this thesis to answer some of these important issues.

The main contribution of my thesis is in characterisation of exceptions in the context of multi-process systems software, and in the development of a model and program paradigms which improve system programs by exploitation of exception-handling mechanisms -- either by making the programs more efficient at run-time, and/or by making the program code more modular and reliable. My work also considers the use of exception mechanisms in high level languages in developing systems software and application programs, and attempts to unify the different approaches taken by message-based and procedure-based systems.

1.5. Synopsis of the Thesis

In Chapter 2 exception handling mechanisms in several recent multi-process operating systems are presented. Most of these operating systems use different constructs for exceptional events within a process and between processes. These in turn depend on the character of the process which the operating system supports and the inter-process communication

mechanism. Thus the discussion of exception mechanisms in operating systems includes comparisons of various process and inter-process communication concepts, and also includes a discussion of the various mechanisms available for interactive debugging, as these are often claimed to be useful for exception handling. Two typical problems in operating systems design are then presented, and current solutions given for which exception mechanisms have been used.

In Chapter 3 a taxonomy of exceptions is derived from these examples, including a new class of *ignorable* exceptions. A model for systems design is then propounded, based on the classification. Many examples are given to illustrate the model.

In Chapter 4 program paradigms in different languages and systems are described for example problems. These paradigms are based on the general model, but they employ system dependent features such as the server-client inter-process exception notification mechanism.

Chapter 5 describes implementations of two systems programs which were designed according to the model, showing the suitability and practicality of the approach.

Chapter 6 concludes with a summary of the thesis, and considers further areas for systems design methodology which use exception mechanisms.

CHAPTER 2

Exceptions in Multi-process Operating Systems

2.1. Introduction

The history of programmed exception-handling has been reviewed by Goodenough [Goodenough 75] and Levin [Levin 77], both of whom made new proposals for exception-handling in high level language design. Since 1977, there has been more emphasis on formal exception-handling in both high level language and in multi-process operating system design. It is worthwhile to make a review of multi-process operating systems, because the nature of their exception mechanisms depends on the process model and on the inter-process communication mechanism supported by the operating system.

A multi-process operating system is defined as a system which provides facilities for multiple concurrent user processes which behave as the active entities of the system. It is possible for one process to create another process dynamically. A process may acquire, release and share resources, and may interact, cooperate or conflict with other processes. In multi-process operating systems the designers have to deal with the problems of synchronisation and deadlock. Synchronisation is needed, both for mutual exclusion (because of sharing) and for sequencing (to impose event-ordering for cooperation among processes).

The survey article *Concepts and Notations for Concurrent Programming* by Andrews and Schneider [Andrews 83] is taken as a guide to the review. However, Andrews and Schneider confine their discussion to concurrent high level languages whereas we are also concerned with concurrent operating systems. They divide the synchronization primitives into

two groups; those which rely on shared memory, and those based on message passing.

The operating systems discussed fall into several different classes, including systems from both the above groups.

(1) The UNIX time-sharing operating system [Ritchie 74] was developed at Bell Telephone Laboratories. It is a multi-user system which allows each user to create multiple processes to run in parallel, and contains a number of novel features such as *pipes* which have made it very popular. It is written in the high level language C [Ritchie 78]. UNIX-style operating systems have a large kernel supporting user programs through use of system calls. A null system call typically takes .6 millisecs on a VAX 11/750; a read or write system call takes at least 2.5 msecs. Processes are large and there is a high overhead on process creation. We also consider distributed versions of UNIX such as LOCUS [Popek 81] and Newcastle Connection [Brownbridge 82], where several UNIX hosts are connected over a local area network.

(2) Thoth was designed as a portable multi-process real-time operating system at the University of Waterloo [Cheriton 79a]. The aims of Thoth [Cheriton 79b] were to achieve structured programming through use of many processes, both for concurrent and sequential tasks. Various facilities are provided to make this structuring attractive; small, inexpensive processes, efficient inter-process communication by messages or shared memory, and dynamic process creation and destruction. Thus, the Thoth style is to split certain sequential programs into many cooperating processes. Process switching typically takes 0.5 msec on a SUN workstation. Derivatives of Thoth are Verex [Lockhart 79], [Cheriton 81], and PORT [Malcolm 83]. There are distributed versions; V-system [Cheriton 83a, 83b] where an identical kernel resides on hosts connected over a

local area network, and Harmony [Gentleman 83], where multiple 68000 microprocessors are connected by a Multibus.

(3) Object-oriented operating systems such as PILOT, designed at Xerox PARC for the personal computer environment [Redell 80] written in the language Mesa, rely on shared memory, and use monitors for synchronisation of processes. The overheads on monitor calls are only a little more than for procedure calls, but there is no direct mechanism for general inter-process communication between arbitrary processes. PILOT is a single-language, single-user system, with only limited features for protection and resource allocation. PILOT and Mesa are mutually dependent; PILOT is written in Mesa and Mesa depends on PILOT for much of its run-time support. Thus Mesa monitors were chosen as the basic inter-process communication paradigm for the processes of PILOT, as described by Lampson and Redell in [Lampson 80]. For distributed object-oriented operating systems, a remote procedure call mechanism is appropriate [Nelson 81], [Spector 83].

(4) CAP [Herbert 78] is a multi-user single processor machine built at the University of Cambridge for research into capability-based memory protection [Wilkes 79]. The machine is controlled by a microprogram which provides the user with a fairly conventional instruction set, together with the support of a capability-based addressing and protection scheme. In this protection scheme, the fundamental notions are *capability* and *object*. A capability identifies (i.e. names) some object, and possession of a capability allows some access to the object it names. If the object contains executable code it is called a *protection domain*. The programs of CAP are protected from each other; they are mutually suspicious subsystems (in contrast to those of PILOT). Most of the CAP

operating system is written in ALGOL68C [Bourne 75]. Users may not perform much multi-tasking in the CAP system, but these protected systems present peculiar problems with respect to process creation and destruction, which warrants their inclusion in this discussion.

(5) RIG [Lantz 80] is a distributed message-based multi-process operating system developed at Rochester University. It differs from Thoth-like systems in that it has no shared memory between processes. RIG uses special *Emergency messages* for exception handling.

(6) Medusa [Ousterhout 80], [Sindhu 84] is a unique operating system designed for a particular multiprocessor hardware configuration, Cm*, at Carnegie-Mellon University. Like RIG, it introduces several novel approaches to exception management and is thus included here.

(7) Remote Procedure Call (RPC) is also discussed briefly, as Nelson in his thesis *Remote Procedure Call* claims it can be used as a programming language primitive for constructing distributed systems, although it has no special exception mechanisms.

General exception mechanisms are discussed for all these types of multi-process operating systems.

Now interactive debugging is often cited as a use for exception mechanisms, so the various mechanisms available for achieving interactive debugging are described next.

Exceptions occurring in two typical example problem situations in operating systems are then described. In many operating systems supporting multiple processes, interaction between processes is often of the *server-client* relationship. The server process typically manages a shared resource, which is requested by client processes, either locally or remotely.

To the client, the request for service appears like a synchronous procedure call; the call returns when the request is satisfied, and the results may be passed as parameters. For example, in most message-based operating systems, the host's files are managed by a *file-server* process. If the client processes are viewed as cooperating together, as is the case for a single-user workstation environment, special communication may be required to enhance the performance. The first example discusses the management of files by cooperating processes.

When the server provides access to input-output devices (i/o devices), the desired communication may be more complex than that provided by a synchronous procedure call. For example, the terminal server may wish to communicate to the client that the user has pressed the interrupt key (<BRK>). But the client may be outputting to the terminal at that moment, and may not be reading from the device. Such *asynchronous events* do not fit into the procedure call paradigm, and alternative solutions to handle such events must be found. The second example describes how different operating systems allow the user to handle such asynchronous events.

2.2. General Exception Mechanisms in Operating Systems

2.2.1. UNIX

2.2.1.1. UNIX Processes

UNIX has a structure similar to the classical 2-level operating systems where the kernel contains nearly all of the operating system code. A user process executes either in *kernel mode* or in *user mode*. A UNIX user may have many processes which do not share data with one another.

In kernel mode, a process uses the kernel stack and data space, which is common to all processes operating in kernel mode. Thus for execution of i/o, a user process will switch modes via a system call from user to kernel, allowing access to shared system data. Synchronisation is required between these so-called *kernel processes* to avoid conflict. This is achieved by various mechanisms such as raising the priority of a kernel process executing critical code to avoid interrupts till it has tidied up, or by setting explicit locks on shared resources. This simple way for achieving mutual exclusion using a non-preemptive scheduler works for a single processor system but would be inappropriate in a multi-processor system.

Kernel processes communicate through the shared data of the kernel. In UNIX Version 7 user processes may communicate by creating a pipe, which provides a buffered stream of characters. If one process attempts to read from an empty pipe it is suspended. Thus pipes are not suitable for reporting asynchronous exceptions such as user interrupts, because of their blocking characteristics. In UNIX 4.2BSD [Joy 83], there are facilities for inter-process communication through use of sockets[1], although there is a high overhead on this communication. For example, a Send-Receive-Reply from one process to another on the same machine through a socket takes 10 msecs on a VAX 11/750.

2.2.1.2. UNIX Signals as Exception Mechanisms

A user process may receive notification of an exceptional event by a *signal.*[2] A signal is a binary flag, which works like a *software interrupt.* A process that has been sent a signal will start to execute code corresponding to the signal when it is next activated by the operating system. The default action of all signals is to stop the receiving process.

---

[1]in UNIX Version 7 sockets are not available though pipes are available in both UNIX Version 7 and in UNIX 4.2BSD

[2]the only direct inter-process communication mechanism available in UNIX Version 7

The signal mechanism is very simple: each process has a fixed number (16) of *ports* at which a signal may be received. The user can specify different treatment of a signal by associating a port with a procedure (i.e. a signal handler). A signal is usually generated by some abnormal event, initiated either by a user at a terminal, by a program error, or from another process. One signal is SIGINT which is usually sent to the process running a command when the user presses <BRK> on the terminal. The user can chose to ignore the signal, or execute some clear-up code in the signal handler before terminating. One signal, SIGKILL, cannot be ignored or handled. Thus a process receiving SIGKILL is forced to terminate immediately, with no chance to tidy up at all. To prevent a user from arbitrarily killing processes belonging to a different user, the user-id of the process issuing the SIGKILL signal must be the same as the target process.

Although the signal mechanism is simple to implement and is very powerful in its generality, it has the disadvantage that, like a hardware interrupt, a signal can occur whenever a process is active, leading to non-deterministic execution. The common way of dealing with this asynchronous event is for the signal handler to set a global flag and then to resume execution at the point of interruption. The process inspects the flag when convenient. This involves two stages in the signal handling, which can lead to errors while handling multiple calls of the signal. Furthermore, signals have the same priority, so multiple signals are processed non-deterministically. Finally, a process receiving a signal cannot determine which process sent it, so their use is necessarily restricted to well-defined cases.

2.2.1.3. Intra-process Exception Mechanisms

When a process has received a signal, the signal handler may wish to restore the process to some previously known state. A UNIX process may use the C language library functions

setjmp and **longjmp** for providing a very basic method for passing over dynamic program levels, equivalent to a non-local GOTO. **setjmp(env)** saves its stack environment in **env** for later use by **longjmp**. **longjmp(env,val)** restores the environment saved by the last call of **setjmp**. It then returns in such a way that execution continues as if the call of **setjmp** had just returned the value **val** to the function that invoked **setjmp**.

The feature is really only intended as an escape from a low-level error or interrupt. This mechanism provides no opportunity for procedures which were active but have now been terminated *(passed-over* procedures) to clear up any resources they may have claimed. However, it is cheap and easy to implement.

The mechanism has been enhanced by Lee [Lee 83], who defines macros so that the C language appears to be extended with exception-handling operations, similar to those incorporated into the ADA language [US 80].

2.2.2. Thoth

2.2.2.1. Thoth Processes

Each process belongs to a *team* which is a set of processes sharing a common address space and a common free list of memory resources. Each process on a team also has its own stack and code segments, like coroutines. Processes which share no address space are said to be on different teams. Individual processes on a team are globally addressable, and may communicate via messages. Inter-process communication is achieved through fully-synchronized message passing; concurrency is achieved with multiple processes, one for each event, rather than with a non-blocking Send, which would use buffered messages.

2.2.2.2. Process Destruction as an Exception Mechanism

In Thoth, the idea is to have a separate process for each possible asynchronous event such as an i/o interrupt, and to ensure that it executes fast enough to be ready for subsequent interrupts. Each event is synchronous with respect to the process that responds to it. Global asynchrony of the total program is achieved by execution of its multi-process structure. The claim is that the synchrony of each individual process makes the program easier to understand.

All asynchronous communication is handled by process destruction, as follows. First, each process that encounters a fault or exception, or that is the subject of an attention, is destroyed. Second, if the program is to continue running after one of these conditions, the part to remain is designated as a separate process from the process to be destroyed. Processes which are blocked awaiting messages from, or sending messages to a process which has been destroyed are awakened immediately; the death of a process is detected synchronously when another process attempts to communicate with it.

2.2.2.3. Intra-process Exceptions

No special mechanism exists for handling within-process exceptions in Thoth, and the Zed language in which it was written [Cheriton 79c] provides no special features for exception-handling. Processes incurring exceptions are destroyed so their state does not need to be remembered.

2.2.3. PILOT

2.2.3.1. Processes and Monitors in PILOT

PILOT supports two types of processes -- tightly-coupled processes which interact through the shared memory of a monitor (these are similar to the processes on a team in Thoth), and loosely-coupled processes which may reside on different machines, communicating via messages passed over a network.

New processes are created by a special procedure activation which executes concurrently with its caller. Such a new process has its own local data, but may also share data with the parent process. Synchronisation and inter-process communication of these processes is achieved through monitors with condition variables [Hoare 74]. A monitor acts as a basic mechanism for providing mutual exclusion, as only one process may be executing code in a monitor *entry* procedure at a time. Thus access to a shared resource is usually managed through the *entry* procedures of a monitor. So a monitor is like a server process in a message-passing operating system. Synchronisation and multiplexing of client processes accessing a shared resource are provided by explicit use of condition variables rather than by queues of messages.

2.2.3.2. Intra-process Exceptions in PILOT

The Mesa language provide extensive exception-handling facilities in sequential code. Exceptions are propagated through the *calls hierarchy* until a handler is found. The root procedure of a Mesa process has no caller; it must be prepared to handle any exceptions which can be generated in the process. Uncaught exceptions cause control to be sent to the debugger; if the programmer is present he or she can determine the identity of the errant procedure. Unfortunately this takes considerable time and effort. The interaction between the Mesa exception-handling and the PILOT implementation of Mesa processes has been found to

be irksome [Lampson 80]. It can be expressed as a limitation of the signalling mechanism that an exception signal cannot be propagated from one process to the process which created it.

2.2.3.3. Inter-process Exceptions in PILOT

No special form of inter-process exception is provided in Mesa. It is difficult to communicate exceptional events using monitors. For example, if a pair of processes are communicating through a monitor and one dies, there is no means for the remaining process to be notified. Instead, a timeout interval may be associated with each condition variable, and a process which has been waiting for that duration will resume regardless of whether that condition has been notified. Interestingly, PILOT was originally designed to raise an exception if a timeout occurred. It was changed because programmers preferred the less complicated special timeout mechanism for simple retry situations, rather than employing the intra-process Mesa exception mechanism to handle such retries.

2.2.3.4. Exceptions within Exception Handlers

A further complication in the interaction between Mesa monitors and the exception handling mechanism arises when an exception is generated by an entry procedure of a monitor. There are two ways of dealing with this.

(1) A SIGNAL statement will call the appropriate exception handler from within the monitor, without releasing the monitor lock (as the monitor invariant might not be satisfied at the time of the SIGNAL statement). This means that the exception handler must avoid invoking that same monitor again, else deadlock will result. Also if the handler does not return control to the monitor, the monitor entry procedure must provide an UNWIND handler to restore the monitor invariant. Mesa automatically supplies the

code to release the monitor lock if the UNWIND handler is present, but if the entry procedure does not provide such a handler, the lock is not automatically released, leading to the potential for further deadlocks.

(2) Alternatively, the entry procedure can restore the invariant and then execute a RETURN WITH ERROR statement which returns from the entry procedure thus releasing the monitor lock, and then generates the exception.

However, neither mechanism is checked at compile time so their misuse is possible.

2.2.4. The Cambridge CAP System

2.2.4.1. CAP Processes and Protected Procedures

CAP supports a hierarchy of processes based on the master coordinator, which is a single protection domain roughly equivalent to the UNIX kernel. Any process can dynamically set up a sub-process by executing the enter-subprocess instruction. A process may call a procedure in another protection domain during execution, if it has a special ENTER capability for it. These procedures, accessible behind ENTER capabilities, are called *protected procedures* (PPs) and provide both protection and modularity. Each protection domain has its own stack and virtual address space. Changing protection domains is roughly equivalent to a Supervisor call in UNIX, or making an entry to a monitor in PILOT.

Protected objects, such as a file directory, are managed by PPs, which may be entered by any process possessing an appropriate capability. Although it is always entered at the same place, a PP usually examines its private data structures left by the the previous call, and makes further decisions on the basis of what it finds. In the CAP operating system, each PP is written as a complete ALGOL68C program, not as a procedure. A special run-time

library has been written to allow PPs to have a relationship very similar to a coroutine structure. The domains are fully protected from one another except for capabilities which are explicitly passed as arguments.

A CAP process is composed of the various PPs independently compiled, which can then be bound together to form a system image. A certain amount of dynamic linking of new PPs into a process is provided, but creation and deletion of new processes is a non-trivial task (cf. Thoth and PILOT).

During execution, the CAP process crosses protection boundaries, behaving somewhat like a UNIX process making calls to privileged system routines and changing its mode from user to kernel. However, the UNIX system is only 2-level, whereas there can be many levels of PP calls in CAP.

The CAP operating system PPs are required to provide several kinds of services, the majority being concerned with gatekeeping [Needham 77] (i.e. control of access to a service or resource). One example is in calls to the *master coordinator*. In the CAP operating system, critical Sections which may not be interrupted are executed only in the master coordinator process. Access to the ENTER-COORDINATOR instruction is pre-checked by a gatekeeper PP called ECPROC, running within the caller's process.

Alternatively, some services are provided in dedicated systems processes, rather than as PPs in the user's process - such processes correspond roughly to the classic server process model. Access to the service is controlled by a PP in the user's process.

Thirdly, some system-wide services such as message buffers are managed by a PP which has exclusive capabilities for the data structures. Processes wishing to access buffers must request the PP to do so on their behalf.

Inter-process communication is through messages held in buffers which are dynamically allocated from a fixed pool. They are accessed through objects called *message channels*. A PP exists for setting up message channels; it checks software capabilities and establishes a communication path between processes. A successful call to this procedure results in one or more capabilities for more primitive message passing procedures such as SENDMESSAGE, which is one of the entries to ECPROC, with arguments specifying the (software) capabilities for the message channels. ECPROC performs any transfer of data or capabilities which may be required, and then makes a call to the master coordinator if any scheduling action is required.

2.2.4.2. Intra-process Exceptions in CAP

CAP has a primitive fault-handling mechanism which causes a simulated ENTRY to a particular protected procedure called FAULTPROC in the user's process. FAULTPROC examines the state of the process and takes special action on virtual memory faults and linkage faults, eventually returning to retry the failing instruction. For other faults, it alters the stacked program counter of the faulting PP to a fixed value, and returns to it, having stored useful information about the fault in a fixed place. It also sets a flag to cause a further fault when the original PP itself does a RETURN, similar to the mechanism that is used to propagate asynchronous attentions. The code at the fixed address can examine its environment, including the information stored away by FAULTPROC, and decide what to do. Faults which have not been dealt with by a PP are propagated back to the calling domain, with an appropriate change in semantics on crossing the protection boundary. For example, the fault *limit violation* incurred by a procedure is distinct from *called domain suffered a limit violation and took no corrective action.*

2.2.5. RIG

2.2.5.1. Exceptions in RIG.

In RIG, internal exceptions (i.e. within-process exceptions), are handled by a procedure call oriented mechanism, based on two library procedures in the implementation language BCPL [Richards 79a]. **Errorset** and **Error** allow an error notification to propagate back up the calls hierarchy to a designated point. The program state at the point where the exception is raised, is lost. Thus RIG does not provide for program resumption. **Errorset** is a function that accepts a severity level and a procedure as arguments. **Error** accepts a severity and error code as arguments.

If **Error** is called, the call stack is unwound to the point of the most recent **Errorset** with a severity level equal to or greater than the severity of the error. The error code is then returned to the caller of **Errorset**, which can then attempt to recover from the error. Calls to **Errorset** may be nested. This mechanism is quite powerful, although as Lantz himself remarks that provisions should be made for associating handlers with particular exceptions, similar to Levin's or those of CLU.

The RIG inter-process exception mechanism is more unusual, as it employs a new message type -- an *emergency message*. An emergency message is delivered with highest priority, ahead of any other messages queued for the receiving process, and will cause a blocked process to be awakened. When an emergency message is received, the *emergency handler* associated with the process is invoked. RIG adopts a server-type process structure for its operating system utilities, in contrast to Medusa which uses a *shared object* concept. In RIG, a process has to explicitly register interest in another process before it will receive emergency messages from it, whereas in Medusa, users of shared objects are automatically notified of exceptions through

the backpointers stored with the shared objects.

Emergency messages are typically generated by *event-handlers*, although they may be generated by any process. An event handler is a process that is capable of detecting, or will always be informed about, the occurrence of a particular kind of event. In general, a process, PA, must register with an appropriate event handler that it wishes to be notified when a particular event, EPB, occurs in process PB. When EPB occurs, the event handler notifies PA via an emergency message.

One particular event with which all processes are concerned is the death of other processes and machines with which they are communicating. The RIG *Process Manager* acts as an event handler for suicide, crash or suspension. The Process Manager relies on the kernel to notify it whenever a process dies. The Process Manager then sends the appropriate emergency message to all interested parties.

The relatively simple approach of RIG to emergency handling has several advantages: it is cheap to implement, there is only one handler per process, so it is easier to read, and there is less non-deterministic program execution than in UNIX.

The disadvantages of this approach are that there is only one handler for all emergencies, and the internal exception handling mechanisms provide no means for implementing exception handling in a high level language such as ADA [US 80], so a separate mechanism must be used by each language compiler. Further, there is still some non-deterministic execution on receipt of an emergency message, as the process is awakened from either Send or Receive states.

2.2.6. Medusa

2.2.6.1. Processes in Medusa

In Medusa, cooperating processes, called *activities*, are grouped into *task forces* which form the fundamental unit of control. All the operating system utilities are task forces. Processes in both Medusa and RIG communicate by passing messages; in Medusa, processes on the same task force may also use shared memory. A task force is thus similar to a Thoth team; the difference is that Medusa activities are scheduled to run in parallel, whereas in Thoth, the principle that only one process on a team executes at a time is crucial in the design philosophy.

2.2.6.2. Exceptions in Medusa

In Medusa, whenever an exception is detected, it is sent to a single *exception-reporter* utility. This central utility acts as a clearing-house for the reporting of exceptions to handlers. Thus although the *detection* of exceptions can occur at any level, the *reporting* is encapsulated in a single utility. This provides a uniform reporting mechanism for all exceptions, both inter-process exceptions and within-process exceptions, regardless of their origin.

The predefined internal exceptions are divided into about a dozen *reporting classes*. One class is *floating point overflow*, another is *execution of unimplemented instruction*. There are eight other classes for external (inter-process) exceptions. Each activity may nominate a different handler to deal with each of these predefined reporting classes. The exception reporter utility also provides functions for adding user-defined exceptions to this list.

We include a discussion of Medusa's internal exceptions here; external exceptions are described later, in Section 4.2.3.

2.2.6.3. Internal Exceptions in Medusa

Four different types of handlers may be used for internal exceptions.

(1) By default, an internal exception is handled by the parent of the activity's task force -- the parent has limited access to the state of the activity and can resume it if desired. This method is commonly used as Medusa's recovery mechanism. For example, many small programs will be invoked from the user's command interpreter and will not deal with exceptions at all; the command interpreter will kill the exception-generating task force and will output a message on the user's terminal.

(2) The handler may be specified as an out-of-line handler, which is equivalent to the address of an interrupt routine. This type of handler is expected to be used as an entry into the reporting mechanisms of high-level languages, which may then propagate exceptions through abstractions in programs.

(3) An in-line handler is invoked only when the activity checks explicitly for an exception occurrence -- no special report of the exception is made.

(4) Medusa also provides the ability for an activity to name a *buddy* activity in the same task force to handle any specified exception class. When the exception occurs, the buddy receives access to all the private information of the exception-generating activity, which is suspended. Thus the buddy can be used to help in interactive debugging, and in many other situations where remote handling is essential for recovery.

One handler may also activate another handler if it is unable to deal with an exception (e.g. an out-of-line handler may activate the task force's parent, by defining extra exceptions to the exception reporter utility).

The notion of a buddy activity to handle exceptions is a departure from the procedure-oriented mechanisms previously discussed. Its chief advantage is that the handler can reside on a separate processor, thus protecting the handler code from the errant processor. There is also a saving of space occupied by the handler on the errant processor, and it is useful for recovery operations after certain computer limitation exceptions such as *stack full.*

2.2.7. Remote Procedure Call

RPC has been proposed as a high level language mechanism for the synchronous transfer of control between programs in disjoint spaces whose primary communication medium is a narrow channel. Nelson details two designs for RPC in his thesis [Nelson 81]. The first assumes full programming language support and involves changes to the language's compiler and binder, while the second involves no language changes, but uses a separate translator -- a source-to-source RPC compiler -- to implement the same functionality. The RPC paradigm is suitable for tasks with master/slave relationships.

However, RPC offers only that subset of general inter-process communication provided by message-based operating systems involving hierarchic control structures; the maintenance of a dialogue between peers having symmetrical control relationships is not easy. This is because in an RPC environment, control is passed from the requesting process when the server process is called, and is returned when the server has completed processing the request. The concept of asynchronously interrupting an executing server process is counter to the RPC paradigm.

RPC alone is not sufficient for conveniently programming in a distributed computing environment, because the server cannot make *out-of-order* replies to clients. (By out-of-order we mean that at any instant a server may have accepted more than one client request and the

server makes a reply to the least recent client request first). RPC is the only inter-process communication mechanism in the concurrent high level language ADA [US 80] (to which references are made throughout this thesis).

2.3. Mechanisms for Interactive Debugging

Interactive debugging operations such as step-by-step traces are often cited as a use for inter-process exception mechanisms. However, *programmed* exception-handling cannot usually be used for interactive debugging, because access to the local variables of a routine by another part of a program or by a another process is usually prohibited by scope rules. Thus special mechanisms must be employed for interactive debugging. Several of these mechanisms are now described.

Many commercial time-sharing systems provide a debugger, such as DEC's VAX-11 DEBUG utility [Beander 83], which uses special mechanisms. VAX DEBUG is strictly an object program debugger which has access to the symbol table of the user's compilation units. DEBUG uses VAX/VMS exception handling mechanisms to provide the needed control over user programs such as stopping execution at breakpoints. The main feature of this mechanism (described fully by Beander) is that breakpoints are converted to intra-process exceptions which are caught by DEBUG and handled by the debugger instead of being propagated up the calls hierarchy.

UNIX's software interrupt mechanism has been extended to provide a powerful but inefficient mechanism whereby a parent process can monitor the progress of one or more child processes. These tracing facilities can be used for interactive debugging and include the ability for the parent to inspect and even modify values within the data area of the child process. The child is traceable only if it gives its permission by explicitly executing a system call.

Then, every time the child process encounters a software interrupt the parent process is given the opportunity to intervene. The child may be blocked indefinitely if the parent ignores it. A better scheme would involve a more efficient transfer of control between parent and child, and would also only operate if both parent and child mutually agreed tracing should commence.

In Verex we have developed a unique process-oriented exception mechanism, which relies on the existence of an exception-server process. Whenever an active process incurs an execution error (i.e. an error detected by the hardware such as illegal instruction, illegal address), it is blocked and a kernel routine is invoked which forwards the message buffer to the exception-server as if it came directly from the offending process. The exception-server can inspect the status of processes that send messages to it, and can detect those that have incurred errors (the operating system sets a flag). The exception-server has the power to control the offending process -- either by destroying it, or forwarding it to any other interested parties, such as the debugger. The first handler for all errors is the exception-server; to avoid bottlenecks, the exception-server creates a separate team to handle each exception request. At present, the team created has full power to pry into user and system data areas.

The advantage of this approach in a distributed system is that it is often easier to handle an exception condition at a point that is logically distinct from the site where the exception was raised.

The Verex error handling mechanism could obviously be extended to handle exceptions as well as errors, by programming a RAISE statement to send an appropriate message to the exception-server. At present we use this mechanism only to terminate or debug the erroneous process.

In Medusa, the mechanism for debugging and tracing, called MACE, is separate from the exception reporting mechanisms already described in Section 2.2.6. MACE executes as a single task force with just one activity, and it is located on a dedicated LSI-11 processor. MACE has its own simple terminal i/o and exception handling facilities, so it need not rely on any of the other utilities; it is therefore almost crash-proof. The other utilities notify MACE of breakpoints using a pipe process; MACE can restart such broken activities when the operator desires.

The author implemented an interactive remote debugger [Atkins 83a] for programs written in BCPL running on machines connected over a Cambridge Ring [Wilkes 80]. Most of these machines have the Ring as their only peripheral, and use it to communicate with terminals and discs. Hence a resident debugger is of limited use, as many machine crashes will cause contact with the terminal to be lost, rendering the debugger inaccessible just when it is most needed. However, as a computer's Ring interface allows a remote machine to read and write words of its memory, and halt, start and reset it, it is possible to move the debug program to another machine. The debugger can inspect and modify data structures directly; simple communication can be achieved by polling fixed memory locations in the target machine. This relies on very little software being working in the target machine -- just enough to signal that an abort has happened and issue a message. The remote debugger acts interactively to read and update memory in the remote machine, which may be unaware that it is being examined. It also handles traps and aborts in the remote machine. It is thus a multi-event program, awaiting either a character from the user at the keyboard, or a signal at the remote machine. These events are asynchronous. This debugger is very successful, allowing new machines to be installed on the Ring in only a few days. Standard coroutines are used to implement the debugger because of their low set-up cost and low run-time overhaed

on coroutine switching. However, special mechanisms are used for signalling traps and breakpoints.

In Mesa, any exceptions which have been propagated up the calls hierarchy to the root procedure, are then passed to the debugger, as described in Section 2.2.3.2.

Thus the only operating system which uses the *same* mechanism for debugging and exceptions is Mesa; and it has been observed that in Mesa, the major reason for raising an exception is to invoke the debugger, and thus the exception *handling* facilities embedded in the language are secondary to the use of exception *detection.*

Thus in general, the mechanisms for debugging are not suitable for programmed exception handling, and so the thesis is concerned mainly with the general exception mechanisms described in the previous sub-Sections 2.2.

2.4. Exception Handling in Action

2.4.1. Inter-process Cooperation

An interesting approach to how servers may notify clients of exceptional events, is described by Reid *et al.* [Reid 83] for the Mesa file-system. In the Mesa file system, client processes are viewed as cooperative, and to achieve this cooperation they support file-sharing. If one process wishes to use a file in a way that conflicts with the way that a second process is using the file, the process that is using the file may be asked to relinquish it. For example, if a process wants to write a file being read by another process, the process that is reading the file is asked to stop. Also, a process may ask to be notified when a file becomes available for a particular use. However, the processes that share files need neither communicate explicitly, nor know one another's identities. We assume that such notification does not happen fre-

quently, and hence can be regarded as an exceptional event. Cooperating processes are used in the design of the Xerox Development Environment, an integrated multi-window programming environment to support sophisticated tools such as windows that load themselves with the new version of an error log each time it changes.

To achieve the notification, clients provide *call-back procedures,* by passing them as procedure parameters to the file-system monitor (which acts like a server). For example, a client can ask the file system monitor to notify it whenever a file becomes available for some particular access, as the client might want to be awakened when the file is available so it can try again. The procedure **AddNotifyProc** is called to register such a request with the file system, and the procedure **RemoveNotifyProc** is called to remove it. When the file system determines that the conditions have been satisfied, it calls the **NotifyProc** passed in as the parameter. The system has to guarantee that when a client is notified, it can indeed acquire the file for its desired access.

There is nothing new in using procedure parameters to provide call-back facilities from subprogram to caller [Black 82]. However, the novel way this has been used in providing inter-process communication for a monitor-based operating system is worthy of further examination. The main use of this mechanism is to allow cooperating clients to execute a **PleaseReleaseProc** whenever the file-server to which they have registered a **NotifyProc** needs to use the same file for another process. However, the authors state that there are difficulties with this mechanism.

First, because the client must be prepared to have its call-back procedures invoked at any time. This may cause subtle synchronisation issues in the inter-process communication between the client, the file system, and (indirectly) other clients.

Next, the difficulties are inherent in writing multi-process programs. As a means of communication, the call-back procedures expose these difficulties. Note that clients need not master the subtleties of call-back procedures to use the file system. They can choose instead not to cooperate in their use of files, using a system-provided **PleaseReleaseProc** that always returns **no.** Often, tools are first written with little or no cooperation and they gradually evolve to allow more cooperation.

Third, since many clients may be calling it simultaneously, the file system must lock some of its internal data structures while it calls the client-provided **PleaseReleaseProc** or **NotifyProc.** Although essential to preserving the consistency of data structures and to provide some guarantees on its behaviour, this means that there are file system operations that cannot be invoked from **PleaseReleaseProc** or **NotifyProc** without causing deadlock. Therefore, some of the file system procedures may not be called from within a **PleaseReleaseProc**; these include **Release.** If the **PleaseReleaseProc** calls one of these procedures, the process will deadlock on the file system's monitor for that file. If it must call one of these procedures, it must fork another process to perform the call and must not wait for that process to complete, since the process will not complete until the **PleaseReleaseProc** returns. The return value **later** from a **PleaseReleaseProc** may indicate that a process has been forked that will release the file.

We consider how to implement such a set of cooperating processes in the Thoth environment. As servers cannot **Send** to clients which have made **Send** requests to them (by convention for security and deadlock-avoidance [Gentleman 81]), some other means has to be provided for notifying clients of the exceptional event **PleaseRelease.**

The drastic approach would be for each client to nominate another process, the **victim**, which the server kills when it wishes to notify a **PleaseRelease** request. A fourth process, the client's **vulture**, awaits the death of the victim (usually by execution of a **Receive Specific**) from it, and is notified by the operating system when this occurs. The vulture can then notify the client by a **Send** request. This arrangement is shown pictorially in Figure 2.1.

The disadvantage of this mechanism is that the vulture and victim processes may be required in certain applications, merely to provide exception notification to the client.

An advantage of this mechanism is that the server does not block when it kills the victim, so no deadlocks occur during the message exchange. Another advantage is that the client receives the notification (via the vulture) when the client executes a **Receive** -- thus the notification is synchronous with respect to the client's execution. Furthermore, it always works, and its use is now well-understood in writing systems code. If the victim process can

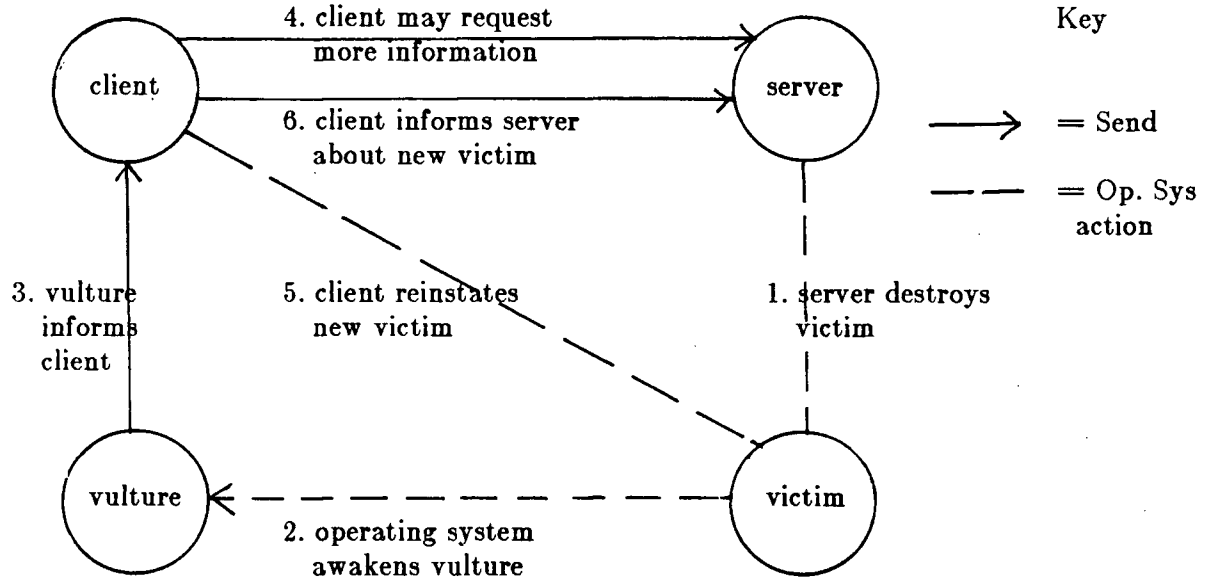


Figure 2.1. Thoth Server-client Notification using a Victim Process.

serve another role, such as a timer or worker whose state is not needed if the exception occurs, the overheads are not too great.

When the client receives a message from the vulture that the victim has died, the client must recreate the victim (for future notifications), **Send** to the server to establish the new victim, **Send** to the server for more details (as the victim's death serves only as a binary signal of the event's occurrence) before finally executing the **PleaseReleaseProc** code.

The Thoth solution described above requires the following process switches.

(1) For the server to destroy the victim, the equivalent of 2 process switches (e.g. in the Verex implementation, the server does a **Send** to the Team-invoker who takes appropriate action before making a **Reply** to the server although in the PORT operating system this is achieved by a kernel call taking the equivalent of <1 process switch).

(2) For the vulture to notify the client, 2 more process switches.

(3) For the client to recreate the victim (if necessary), the equivalent of 4 process switches (e.g. in the Verex implementation, the client does a **Send** to the Team-Invoker who makes a request of the Team-creator before replying).

(4) Then the client has to notify the server of the new victim, and, after receiving the reply, has to request more information from the server about the nature of the exception. This takes 4 more process switches. Therefore this mechanism requires the equivalent of 12 process switches, (including **Destroy** and **Create)** which could take 6 msecs[3].

In the UNIX environment, the client could be notified asynchronously that an exception had occurred, by a UNIX signal from the server. However, there are only 16 different signals, and their use is by convention pre-set to certain exceptions (as explained previously) so their

[3] assuming the Send-Receive-Reply sequence takes 1 msec.

use in this situation would not be encouraged. An alternative in UNIX 4.2BSD would be to use an asynchronous **Send** from the server to the client via a datagram socket. This in theory does not block the server, so deadlock should not occur. As for the previous example, the client process would receive the notification synchronously when it executed a **read** from that socket. The client process could then execute the **PleaseReleaseProc** code.

At first sight the UNIX implementation seems simplest, but there are major drawbacks. Firstly, the asynchronous **Send** *does* block when the i/o buffers are full, which can occur when communication is over a slow network, leading to potential deadlock.

Secondly, because of the nature of UNIX processes, and the asynchronous nature of the inter-process communication, the time for a Send-Receive-Reply is slow (10 msecs on a VAX 11/750), and the system call to a socket is slow (2.5 msecs). Thus the UNIX 4.2BSD notification (using a socket) takes 12.5 msecs[4].

An alternative in Thoth, is to keep the client process always listening to the server. The server then replies directly to the client. If the client has to be ready to act as a server itself to higher level clients (often the case in a hierarchy of servers) and accept requests, the client must spawn a worker on the same team to await notification messages from the server. This may mean that the server makes an out-of-order **Reply** to the worker. This technique is commonly used and reduces the number of process switches compared with the vulture-victim mechanism. The process switching is now simply from the server to the worker, and from the worker to the client. This takes only 1.5 msecs on a SUN workstation, much faster than the UNIX approach. If we provide the worker with the code of **PleaseReleaseProc,** we can call the worker an *exception process* and possibly even eliminate the need for the client to be told

[4]This is measured as the time for a datagram transfer of 32 bytes from server to client if client is ready to receive.

of the exception. In this case, the number of process switches would be still further reduced to two, whereby the server switches to the exception process and back again. This is the same as the UNIX situation using an asynchronous **Send** to a datagram socket. The separate exception handler process so described has the advantage that the exception handler code is separate from the normal-case code in the client process. Furthermore, the context of the client code does not have to be saved to execute the exception code, as the client process keeps its separate existence during the exception process's execution. Another apparent advantage is that increased concurrency is possible.

This solution is very attractive, but it is fraught with problems in real applications, particularly in the synchronisation of exception process and the client. This is discussed fully in Section 4.3 on **Exception handlers as processes.**

In RIG, an emergency message from server to client can provide the notification like a socket in UNIX4.2BSD. However, execution of the client is non-deterministic as receipt of the emergency message causes a change of flow of control if the client is either sending or receiving. If the client is executing a Receive at the time, the emergency message arrives synchronously. However, if the client is executing a Send at the time, the emergency message arrives asynchronously. Thus the degree of non-determinism falls between that of the Thoth and UNIX4.2BSD model, and the Mesa implementations. The same difficulties with potential deadlocks and difficulty of synchronisation will occur as with the Mesa and Thoth approaches.

2.4.2. Asynchronous Events -- Terminal Interrupt

One common problem is in defining what should happen to an executing program when an asynchronous event occurs such as an attention or an i/o interrupt. When a terminal user presses <BRK>, what happens depends on what the program specifies should happen, and

on what mechanisms are available. For example, an editor might specify that typing <BRK> will cause an exit to the program which called it only if all the user's files have been saved; otherwise it asks if that is really what is meant.

In UNIX one of the software signals (SIGINT) is used to interrupt all the client processes which are using the terminal when the user hits <BRK>. Many client programs do not catch interrupts, and are killed. However, a process can arrange to catch interrupts and take appropriate tidy-up actions before either continuing or not.

In the Thoth environment, process destruction is used to handle <BRK> by specifying each team to have an associated terminal server. Each such server provides a facility whereby a client process may nominate a victim process which is to be destroyed on input of <BRK>. The victim is destroyed immediately. Alternatively, the client can specify no victims; can make itself unbreakable, and can field the <BRK> as it wishes. Generally all processes on a team are breakable, so an interrupt on the team's terminal destroys the entire team.

The chief advantage of this mechanism is that it is simpler than UNIX-like signals as the user doesn't have to worry about a process's internal state after the event -- it doesn't exist! Furthermore, a list of processes blocked on others has to be maintained by the operating system for implementation of the message-passing primitives, so little extra effort is required to implement this destruction mechanism.

However, it is necessary to maintain the integrity of resources which are owned by processes which are destroyed, either by a list in kernel data, or by garbage collection, or by timeouts. The Thoth solution is to provide garbage collection of orphaned resources, or to make some processes (such as servers) unbreakable. Runaway unbreakable processes can

always be destroyed by an appropriately authorised **Destroy** command, like the UNIX KILL command. Unlike UNIX, any other processes blocked on a destroyed process are awakened immediately. Thus the Thoth process destruction provides a general mechanism suitable for communication of asynchronous exceptional events.

This mechanism has been used in the Thoth text editor to provide a very simple means for handling both <BRK> and other exceptions [Cheriton 79b]. In this scheme, the editor is split into two processes on the same team. The process *Main* holds all the data structures such as the text buffer which must survive a <BRK>. The other *Editor* process implements the functionality of the program, viz. reading and performing user commands. The *Main* process is unbreakable while the *Editor* process is breakable, so only the *Editor* is destroyed on a <BRK>. Because the *Editor* process executes the commands, <BRK> causes the execution of the current command line to stop. The text buffer remains consistent because it is maintained by the *Main* process. The *Main* process detects when the *Editor* process has been destroyed and creates another *Editor* process which proceeds to read a command line from the input. Thus, the handling of <BRK> appears to the user as that of the editor immediately returning to command level.

This structure is also exploited in handling exceptions. On detecting an exception, the *Editor* process destroys itself after leaving a pointer to an error message in a global variable indicating the exception. The user is notified of the exception by the next incarnation of the *Editor* process, which prints the error message. Full details are given in [Cheriton 79b].

In PILOT, communication with peripheral devices is handled by monitors and condition variables much like communication between processes. Each device has an interrupt handler process which is awakened by a signal on a condition variable. The target (user) process

receives notification of the event by the interrupt handler via another condition variable.

User interrupts from a terminal are treated as exceptions (usually the ABORT exception) which the user can chose to handle or ignore in the usual way. Thus there is no way to stop a runaway process except to reboot the system -- in a single-user environment this is an acceptable approach. A similar attitude is taken by the designers of TRIPOS, a single-user operating system designed at Cambridge University [Richards 79b].

In CAP, an asynchronous event such as an attention may be signalled via the master coordinator. The target process is notified of the event by suffering an enforced jump. There is just one attention handler active at any time in a CAP process, which is entered immediately an attention occurs. The operating system allows the user to trap all attentions by specifying an attention handler, which may either perform tidy-up actions on the current domain and then execute a standard RETURN to the calling domain, or may attempt to execute a CLEAR on the attention. The operating system sets a RETURN-TRACE flag so that whenever the currently executing domain executes a standard RETURN, the calling domain is also notified of the attention in exactly the same way. During the execution of the tidy-up operations, other protected procedures can be entered; the CAP operating system notes that these protected procedures are in a special state and disallows recursive calls to the attention routines through multiple attentions.

Attentions are of different degrees of severity, and different capabilities are required to clear them. The attention is propagated to the calling protection domain if the attention routine in the current protection domain cannot clear it. The calling protection domain then has a chance to handle the attention. The protection domain STARTOP, the initial protection domain in every user process, has a capability which allows it to clear every attention. By

default, an attention will terminate a user's job and return to initial state.

The advantages of this mechanism are that it allows the user to trap all faults and that it copes with multiple attentions. The disadvantages are that it allows non-deterministic execution, and that it cannot be used to stop runaway processes which loop during execution of attention handlers. And as the attentions are arranged in a simple hierarchy of levels, the mechanism has proven to be somewhat inflexible for handling different types of asynchronous events encountered in computer communications and in distributed environments.

2.5. Summary

The survey of the various operating systems shows the great diversity of exception handling mechanisms. Purely procedure-based mechanisms, used extensively before about 1977 have disadvantages when applied to distributed systems because of the assumptions of shared memory. For handling exceptions in distributed operating systems, in RIG a mixture of procedure-based and message-based techniques are used; RPC uses a procedure-based technique, and Medusa uses a process-based technique.

The examples show that the mechanism for *notification* of exceptional events (such as the asynchronous event of <BRK>) is system dependent, and varies according to the process model and the inter-process communication facilities provided by the operating system. For notification of <BRK>, UNIX uses its inter-process exception mechanism to provide a software signal which forces a control flow change in the target process. Thus the <BRK> exception is asynchronous in this environment (where a *synchronous* exception is defined as one where the process is in a state ready to receive it). Thoth uses immediate destruction of the process awaiting that event, and other processes receive notification when they attempt to communicate with the destroyed process. This is therefore treated as a synchronous

exception. The PILOT handler for each device uses monitors with condition variables to notify user processes of device interrupts, again this is a synchronous exception. In CAP, i/o device interrupts force an asynchronous transfer of control to an attention routine in the currently executing process, similar to a hardware interrupt. Of the mechanisms aforementioned, only Thoth's can be conveniently extended to a distributed environment, as the others rely on shared memory for their execution. In RIG, an emergency message is used to notify the target user process: this message unblocks the target either synchronously or asynchronously. In all cases, the default action of the operating system is to destroy the target user process (or processes) to which the terminal was connected.

Thus it cannot be stated in isolation that X is a good technique for exception handling, or that Y is a good technique for exception notification; exception mechanisms must be considered in the context of the process model and the inter-process communication facilities available.

CHAPTER 3

Principles for Improving Programs through Exception Mechanisms

3.1. Classification of Exceptions

The discussions and examples in the previous chapter illustrate several features of exceptions encountered by event managers, and these features are used to derive an informal classification of exceptions, as shown in Figure 3.1 below.

Exceptions have been divided into three main groups, according to where the exception is handled.

(1) Server-client exceptions. These need to be communicated from the server to the client (i.e. contrary to the usual inter-process communication from client to server), in order to be handled by the client. The exception *must* be communicated to the client at the layer

Level 1

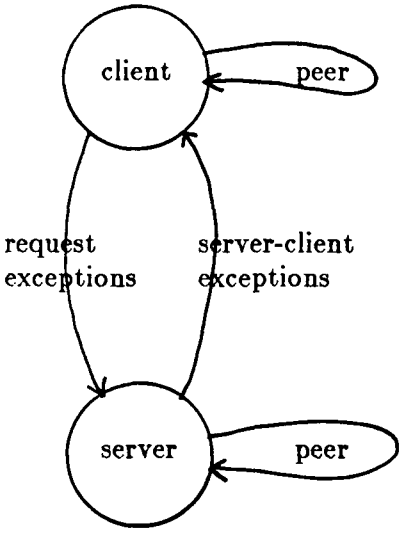


Figure 3.1 Level 1 Classification of Exceptions

above, possibly after the server has handled it and transformed it. When the client and server form a uni-process system, such exception notification is called *call-back* notification. When the client exists as a separate process, we call server-client exceptions *inter-process exceptions.*

(2) Peer exceptions. These exceptions are detected and handled at the same level, and are typically represented by the less frequent arm of an if-then-else branch. So a peer exception detected by a server during processing of a normal request event is handled entirely at the server's level and is therefore transparent to the client at the layer above. If the server consists of a single process, peer exceptions may be conveniently caught and handled using existing mechanisms for intra-process exceptions. In a multi-process system where the server may use related worker processes at the same level to handle specific events, the server may use a cooperating *exception process* to handle the exception. An example is described in Section 5.1.5, where the ITI protocol's exception process handles reconnection after a network failure, transparently to its client process at the level above. Still further distribution of peer exception handling can be achieved, by arranging that a peer exception is handled by a process on a different host implementing the same level of abstraction as the server. For example in a single level of a communications protocol a checksum error on data received from host B may be detected by the link level protocol on host A. The link level protocol on host A handles this exception by requesting a retransmission from the host B link level protocol; host A's client at the network level above is unaware of this retransmission.

Therefore there are two different types of exceptions; those which occur at a level *below* the code which is to handle it (i.e. server-client, or inter-process exceptions), and those which

occur at the *same* level as the handler code (i.e. between cooperating peers). An analogy with the structured layered approach to communications software, as displayed by the ISO Reference Model of Open Systems Interconnection (OSI) [Zimmerman 80] shows that server-client exceptions are part of the *layer interface* and that peer exceptions are part of the *peer protocol.* The communication peer protocols are tightly specified, whereas the interfaces are not, as they are system-dependent.

Thus a starting point for the classification of exceptions has been taken along the dimension of where they are handled -- whether an exception is an inter-process exception from server to client where the exception is specified as part of the interface (the exact nature of which is system dependent), or whether it is from peer to peer, in which case the exception and its handling can be specified as part of the protocol.

(3) Request exceptions. For completeness one more type of exception is distinguished; that of an *unusual request* from a client to a server. This is a request event which occurs with a low frequency, say $<10\%$ on average; such events represent an infrequently used operation in an interface. For example, the operating system i/o servers (or monitors) are driven by events such as requests for i/o from the clients, and by interrupts or messages from the i/o devices signalling i/o completion (a call-back notification). Some of these events will occur much more frequently than others -- e.g. the user request to open a file for i/o will occur only once, whereas subsequent read/write requests to the same i/o server will usually occur many times. It is useful to consider the user's *OPEN-FILE* request to be an exceptional event from the server's point of view, so that the server can use appropriate code to detect it. Such a rare request is a *request exception* and the server's reply is communicated synchronously to the client, which may or may not per-

form further exception handling.

3.1.1. Inter-process Exceptions

This thesis concentrates on inter-process exceptions, as the linguistic and operating system mechanisms for managing inter-process exceptions are very diverse, (as described in Chapter 2). We now show that inter-process exceptions can be further classified along three dimensions, illustrated in Figure 3.2.

Server-client communication is in direct analogy with the intra-process exception mechanisms embedded in high level languages, where a handler may be attached to a procedure call at a higher level in the program unit. An exception occurring in a procedure is usually propagated up the procedure calls hierarchy until a handler is found for that exception. Similarly, an inter-process exception is detected by the server and transferred up the process call hierarchy (i.e. from a server to its client, or from a remotely called procedure to its caller, or from an inner nested monitor to its enclosing monitor). For example, a client may request to open a file of unknown name. The file-server or monitor detects the exception *FILE-NOT-FOUND* while trying to open the file -- this is then communicated to the client

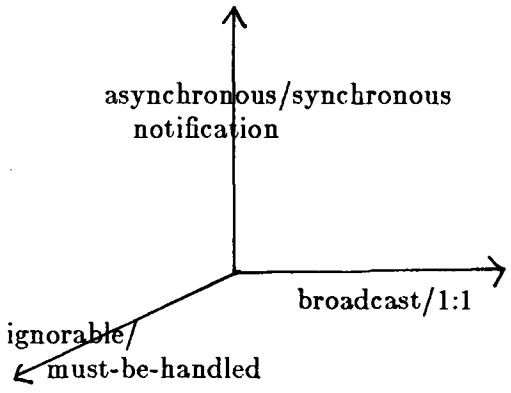


Figure 3.2 Level 2 Classification of Inter-process Exceptions

which handles it by taking appropriate action such as prompting its user for another filename.

Exploiting the analogy with high level languages further, we assume that an inter-process exception mechanism would allow exceptions to be propagated back up the abstraction hierarchy just as the linguistic exception mechanisms allow exceptions to be propagated up the procedure calls hierarchy. Such inter-process call-back mechanisms are available in most operating systems, as described previously in chapter 2. Now in [Liskov 79] the view is held that uncaught exceptions within a process should cause the signaller to terminate. This approach also coincides with that of Bron [Bron 76] and the ADA designers [US 80] who state that the program unit raising an exception signal must be terminated. Therefore we would expect that a structured inter-process exception mechanism would terminate the process which detected and raised the exception. But the analogy with the intra-process linguistic exception mechanisms breaks down here, because one of the purposes of a server or monitor is to handle multiple concurrent requests, so the server cannot be arbitrarily terminated[1]. Instead, the server process detecting the exception should return to its usual blocked state awaiting another event. Similarly, if a monitor detects an exception, the monitor should ensure its locks are released and the monitor return executed, before the exception is raised at the higher level. Thus it is noted that the signaller of an uncaught inter-process exception needs to be resumed, not terminated, after raising the signal. These points of view can be merged by stating that for single process systems, the only handlers which resume operations at the point where the exception was detected at the time they were raised (excluding flag-setting handlers), should be invoked by exception signals which would evaporate if the exception signal was not caught. In other words, allowing resumption after an exception signal is

[1]Unless a separate server instance is used to service each request

using it like an ignorable information signal. In contrast, for multi-process systems where the server continues after signalling an inter-process exception, the exception can be either ignored or not.

This leads to the definition of three major dimensions on which these inter-process exceptions can be placed, already illustrated in Figure 3.2.

(1) One dimension is the degree of necessity of handling; whether the exception is *ignorable* (such as the **PleaseRelease** exception), or whether the exception *must* be handled for correct operation of the system, (such as the <BRK> exception). An *ignorable* exception is defined in this thesis as one which, if uncaught, would just evaporate[2] (i.e. the signaller would resume without delay). Now of the high level language exception mechanisms embedded in the languages ADA, CLU and Mesa, only Mesa allows resumption of the signaller, yet in Mesa, uncaught exceptions are transformed into an invocation of the remote debugger. Thus in uni-process systems, ignorable exceptions are not conveniently handled by any existing high level language mechanisms. However, in a multi-process system as noted previously, the signaller needs to be resumed after communicating an inter-process exception. Thus *ignorable* exceptions take on a new significance in a multi-process environment, because the desired semantic of the signaller being allowed to resume after signalling an interprocess exception is already supported by systems servers. Furthermore, uncaught inter-process exceptions are already ignored in many systems. For example, in UNIX, a SIGNAL to a non-existent process has no effect on the signaller, and has no effect on the system -- it is safely ignored.

---

[2]Note that this definition means that *errors* cannot be classed as ignorable exceptions, unless, during program development, we temporarily wish to ignore uncaught errors.

(2) Another dimension is the number of interested parties to be notified; an exception may need to be communicated by a server to several clients (such as **PleaseRelease**), or to one client (such as <BRK>). For convenience, we distinguish just two cases; inter-process exceptions which may be broadcast to potentially many clients, or sent to one specific client. In UNIX, a SIGINT signal can be sent to ALL the processes in a group, or to one specific process. The processes wishing to receive a signal join a group in order to receive a broadcast inter-process exception; these processes are *cooperating* together to perform some task. Thus exceptions which are broadcast by a server to several clients are intended to develop further cooperation between the clients; as such, the broadcast exceptions may be handled differently from 1:1 inter-process exceptions, adding another dimension to the classification of exceptions.

(3) Finally we consider the notification of an inter-process exception to a client -- if the notification is *asynchronous,* (i.e. if the client is not in a state to receive the exception by having an outstanding request to the server) then it is handled differently than a *synchronous* inter-process exception (in direct response to a request). This feature is part of the server/client interface and is system-dependent, as asynchronous communication usually requires special communication mechanisms. Thus another dimension of inter-process exceptions is the method of notification.

3.2. Model for Systems Design

With the above classification of exceptions in mind, we are now ready to develop a model for systems design which exploits the dichotomy between normal and exceptional events. The model is derived from the author's experiences in designing and implementing communications software and software utilities. The first observation made by the author is

that many systems and communications programs which retain little state between events, are suitable for an event-driven approach, where the programs take the form of an outer loop with an event wait at the top followed by a case analysis. Advantages of this approach for systems software design over the alternative mechanism of encoding the state in the program counter, are given in [Macintosh 84]. The Macintosh approach to designing applications software is to avoid the use of modes[3] which are often confusing to the users. The programming technique proposed to achieve this is an event-driven approach, where the heart of every application program is its *main event loop,* which repeatedly calls the Macintosh Toolbox Event Manager to get events (such as a press of the mouse button) and then responds to them as appropriate.

The event-driven approach may be conveniently achieved by arranging that the client maintains as much state as possible so that only the minimum state is maintained by the event manager between events. The state information is then encoded in the request event to the server. This approach is used in the Apollo DOMAIN system [Leach 83]. For example, a client request to read page N of a file is made idem-potent by specifying the page number, N, in the request event to the file-server. Such a file-server need not then maintain state between events, and can be designed as an event-driven server. The client must maintain the state information of the current page number.

Such multi-process event-driven systems are conveniently modelled by datagram-based message systems such as Thoth, RIG and UNIX 4.2BSD. However, they are not so conveniently modelled in the monitor-based systems such as PILOT, where the inter-process communication is through a monitor and not direct. The dual nature of message-based and

[3]A mode is part of an application that the user has to formally enter and leave, and that restricts the operations that can be performed while it's in effect.

monitor-based operating systems has been discussed by Lauer and Needham in [Lauer 79], so that in general an event-oriented approach can be simulated in all systems. The differences between them are illustrated in the examples of Chapter 4. Programs which are *not* amenable to the event-driven approach include numeric systems such as the evaluation of a function in a deterministic purely functional programming language such as pure LISP; these systems are outside the scope of this thesis. Programs particularly suitable for the event-driven approach include simulation programs, operating system software and non-deterministic programs.

Given a system which is tractable to the event-oriented approach, the author considered what top-down design principles are to be employed. Clearly efficiency is an important criterion, as inefficient programs are very susceptible to subsequent kludges to improve performance. The author was also concerned with mechanisms to achieve modular, incremental changes to programs, based on experiences with dynamic program specifications to add bells-and-whistles features -- difficult to achieve without adding uneccessary complexity to the system.

From a bottom-up approach, there are many possible configurations of processes and inter-process communication paths in multi-process systems. Different choices may be made according to the major objectives of the system. Merging the top-down and bottom-up design ideas result in the model outlined below in Figure 3.3.

The model treats program systems as event driven, and proposes the isolation of the events of the system which may be handled with equal cost. These equal cost events are then identified as normal or exceptional, according to their probability of occurrence and the nature of the event. The program system is then designed according to an appropriate objective, chosen from three common cases -- minimising the average run-time, minimising the

exception processing time (useful for real-time processing), or increasing program functionality. Other objectives such as maximising throughput or maximising resource utilisation may be modelled similarly; for conciseness we limit our discussion to these three objectives. In the rest of this Chapter, the features of the model are elaborated in detail, with many examples to illustrate their use.

(1) Treat the program system as event driven. Most systems programs respond to events; even some data-driven utility programs may be considered as event driven, where the value of the data represents the event to which the program responds. An example of such a program is a **spell** utility program which compares words in a source text file with those in a standard dictionary. Such a program reads characters one by one from the input file till it finds a word delimiter. A word delimiter may be any special symbol such as , (; ! . : etc. The completed word is then compared with the dictionary. An analysis of the **spell** utility shows that it may be considered as *event-driven*, where the

---

Model for Systems Design

(1) Treat the program system as event driven.

(2) Isolate the events of the system which may be handled with equal cost, and group together any such events.

(3) Identify events as Normal or Exceptional (based on their probability of occurrence and the nature of the event).

(4) Design the program according to the objective:

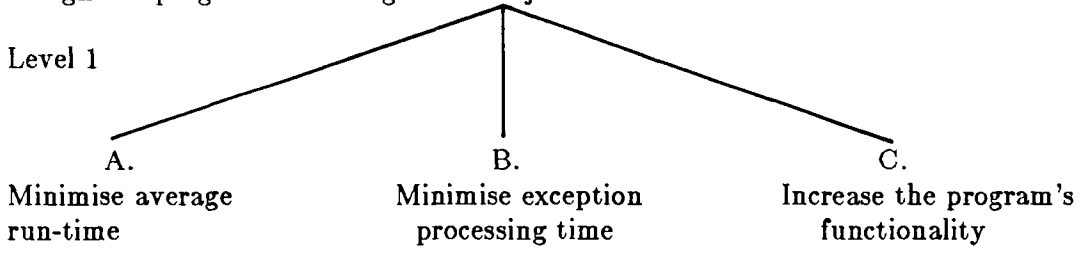


Figure 3.3 Level 1 Model for Systems Design

value of the next character from the input file corresponds to a particular event. Thus for 8-bit characters input, there are 2^8=256 events. The program has just one state, simply *RUNNING*. Like all event-driven programs, **spell** executes a perpetual loop, getting the next event (character from text file) and branching on that state-event pair to an appropriate event handler.

(2) If two or more events are detected and handled in the same way, they can be grouped together to form a constant-cost state-event group. The probability of this new composite event occurring is the sum of the probabilities of the individual events occurring. For example, in **spell,** each alphabetic character can be treated in the same way; therefore events corresponding to alphabetic characters can be grouped together and considered *normal.* The probability of a normal event occurring is the sum of the probabilities of the individual alphabetic characters occurring.

(3) In general, the division of events into Normal or Exceptional should be made such that all normal events occur with a probability greater than all exceptional events. However, in some systems it may not be easy or obvious how to separate events in this way. A guideline for whether to describe an event as normal or exceptional is to decide whether an exception-handling mechanism, requiring a different programming style and incurring different overheads (usually costly) at compile and at run-time, is appropriate for the event. Recall that an exception mechanism should be used when a programmer wishes to improve clarity of the normal-case computation, provided it does not incur unreasonable costs or impair reliability. In particular, *errors* may always be treated as exception events, even if an error event occurs with a probability greater than some other so-called normal event. However, for some systems, the probability of an event varies widely

from time to time, and the occurrence of events may be highly correlated. For example, in a database transaction system, some queries may be rare on average but when one such query is made, other related rare queries may occur soon afterwards. In such situations, the best rule is to take the approach that all events should be treated in the same way. (An example of this is the X.25 protocol implementation described in Chapter 5). For yet other systems, the probabilities of the events are not known *a priori*. Again, it is best to treat all such events in the same way, and as stated in the introduction, the model cannot be used in such situations. Thus we restrict our discussion to event driven systems where the probabilities of the events are known *a priori*.

(4) We have chosen to model systems meeting one of three common objectives.

A. Minimise the average run-time. A model for this important objective is described in the Section following, and many examples are described in detail in Section 3.3.

B. This objective is considered, because many real-time process control systems require prompt, urgent action in response to an exception event such as the temperature exceeding a permitted threshold. Speedy exception processing can usually only be achieved by adding overhead of data-collection to the normal events, so this objective conflicts with objective A above.

C. A new approach to program design is to achieve incremental increase of the functionality of a server/client system by first designing a minimal system, and then providing for incremental addition of new features. The model for this objective, described in Section 3.2.3 below, proposes exploiting inter-process exceptions from a server to its clients to achieve the incremental functionality.

Another objective, that of ensuring that the exception handling code is correct could be considered separately. This objective may be appropriate in systems where the efficiency of the mechanisms are not important, compared with the necessity of correct exception handling, and where implementation of correct code may be to the detriment of performance. This situation may occur in a real-time process control system where failures are infrequent and may never be fully testable. Our model does not address this important issue in detail, because proofs of program correctness are outside the scope of this thesis.

The model is now extended to cover these different objectives.

3.2.1. Objective A: Minimise the Average Run-time

The model is shown in Figure 3.4.

(1) At the first level, the major components of executing the normal events are determined, so the program design can be concentrated upon reducing the greatest cost component from the following[4]:

a) cost of detecting that the event has occurred.

b) approximate cost of handling the normal event

c) cost of maintaining minimum information needed just for correct handling of all exception events -- called the *context-saved* cost.

(2) To structure the program, the most significant cost component is considered from the above costs.

a) If there are many events, and/or the handling costs are small, the event detection costs may be significant. The model proposes mapping one exception onto another to

[4]The case where the exception handling time has a significant impact on the average run-time, such as when a very expensive inter-process exception notification mechanism is used, is considered under the next objective: that of reducing the exception-handling time.

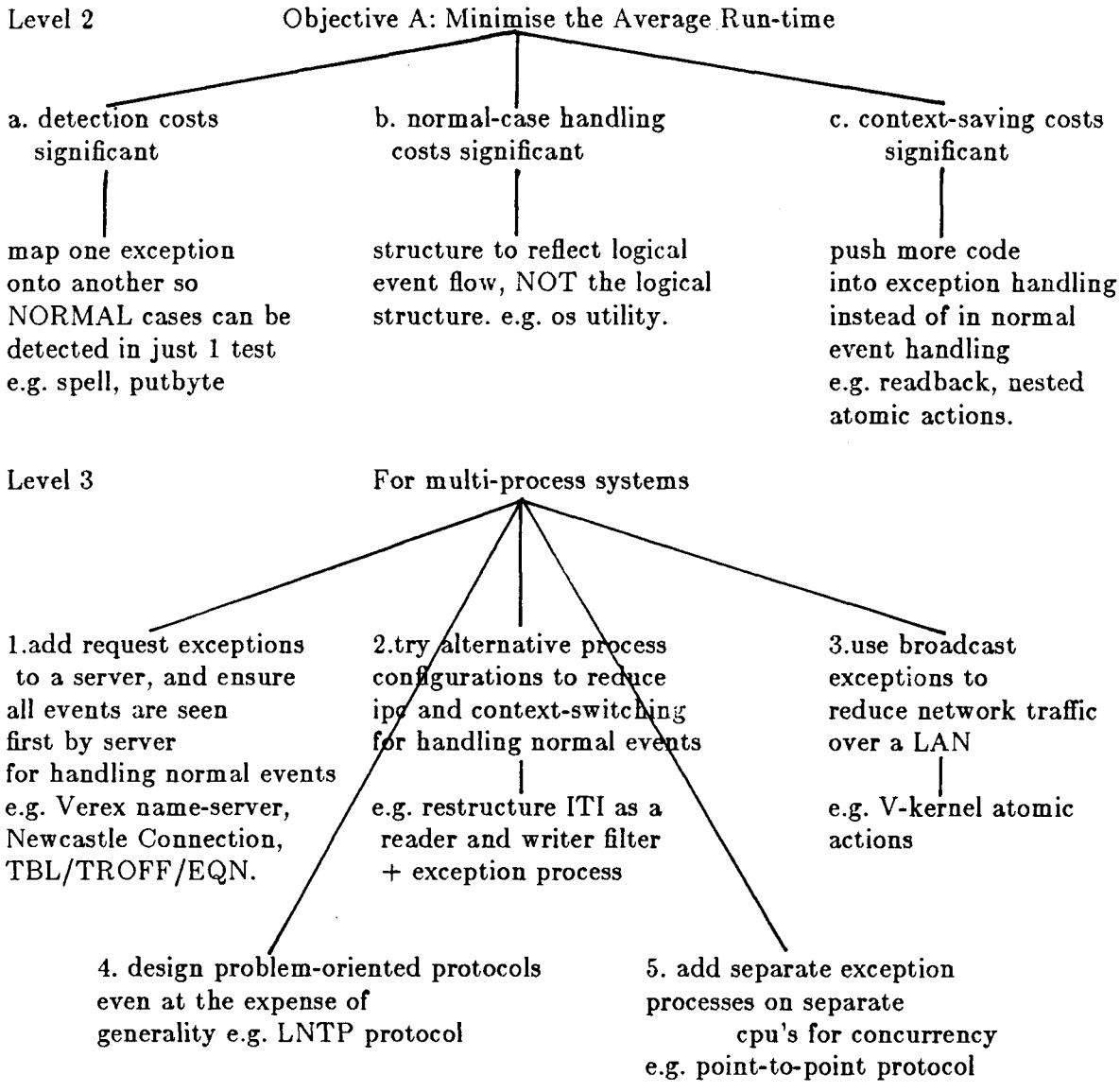


Figure 3.4 Level 2 and 3 Model to Minimise the Average Run-time

enable normal events to be detected in just one test. For example, the **spell** and **putbyte** utilities described in Sections 3.3.1.1 and 3.3.1.2 respectively.

b) If handling costs are significant, the program system should be structured according to logical *event flow*, not the logical structure, e.g. the uni-process **os** utility described in

Section 3.3.2.1. In multi-process systems there are several ways to achieve this. They are elaborated in the third level below.

c) If information is saved during normal case processing purely for correct handling of exceptional events, then an attempt must be made to minimise this *context-saving* code. The model proposes pushing as much code as possible into infrequently executed exception handling code[5]. For example, the provision of a read-back facility described in Section 3.3.3.1, and the design of the protocol LNTP for local area networks, described in Section 3.3.3.2. The author's implementation of nested atomic transactions, described in Chapter 5 also reflects this design principle of minimising context-saving code.

(3) In this level, various techniques are proposed for structuring multi-process programs to minimise the average run-time.

1. Add request exceptions to servers, and allow all events to be seen first by the server which handles normal events. For example, the Verex name-server described in Section 3.3.2.2, and the Newcastle Connection system, discussed in Section 3.3.2.3. The TBL/TROFF/EQN system also can be redesigned to exploit this technique, and an extended example to achieve this is described in Section 3.3.2.4.

2. Try alternative process configurations to reduce the inter-process communication (and therefore context switches) in handling normal events. An example which follows this design principle is the author's implementation of the ITI server described in Chapter 5. It was redesigned from a server process with a reader and a writer worker, to a system consisting of a reader filter, a writer filter, and an exception handler process.

---

[5]bearing in mind that it might be worthwhile to increase the expected use of the exception handling code to ensure it works correctly by mapping one exception onto another.

3. Use broadcast exception messages to reduce message traffic over a local area network (compared with many 1:1 messages), e.g. the author's implementation of atomic actions in the V-kernel, described in Chapter 5.

4. Design problem oriented protocols to handle normal events most efficiently, at the possible cost of reducing the protocol's generality and functionality. This approach has been successfully used in the Apollo DOMAIN system [Leach 83]. Another example is the LNTP protocol, described in Section 3.3.3.2, in which a tradeoff has been made between functionality and performance in that the protocol's flow control has been tuned at the transport layer to the specific environment of a Local Area Network.

5. Distribute exception handling code so it is situated on physically distinct processors thus increasing the concurrency so normal case processing can continue without interruption. An example of this is a point-to-point network protocol described in Section 3.3.2.5.

3.2.2. Objective B: Minimise the Exception Processing Time

The model is shown in Figure 3.5.

As for Objective A, the major components of responding to an exception are determined, so the program design can be concentrated upon reducing the greatest cost component from the following:

a) cost of detecting that the event has occurred.

b) cost of the notification of the exception to interested parties.

c) approximate cost of handling the exception.

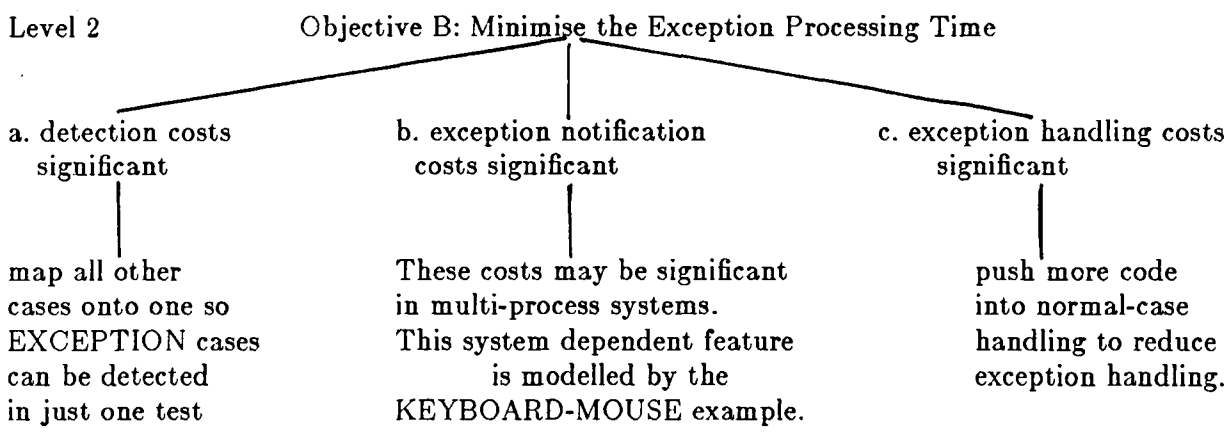


Figure 3.5 Model to Minimise the Exception Processing Time

To structure the program, the most significant cost component is considered from the above costs.

a) Techniques for reducing the detection costs for exception events are similar to those for reducing the detection costs for normal events, considered in Section 3.2.1.

b) In some situations, the inter-process exception notification costs are significant. This is part of the system dependent interface layer between client and server. Therefore, instead of a general model for minimising this cost, program paradigms for different systems and languages are given. An example problem, that of multiplexing input from a keyboard and a mouse, is described in Section 4.1, which illustrates various methodologies for reducing the 1:1 inter-process exception costs, and an example for using 1:many inter-process exceptions is the storage allocator described in Section 4.2. Further, if even the small extra cost of process switching is intolerable, the time for server-client exception notification may be minimised by using a uni-process client/server system with asynchronous exception notification from the server to an in-line procedure in the client via hardware or software interrupt signals.

c) For many systems, the exception handling time can be reduced only by increasing the contextual information saved while processing normal events. Techniques for reducing the exception handling time are the same as those used for reducing the normal case handling costs considered in Section 3.2.1.

3.2.3. Objective C: Increase the Functionality of the System.

The model is shown in Figure 3.6 below.

The minimal system for handling normal events is first implemented, bearing in mind that extra features, which may be termed exceptional, may be added later. The idea is to allow modular increase to the program's (or system's) functionality by treating new features as inter-process exceptions from a server to its client or clients. These extra features must be largely independent of the rest of the system.

One convenient way to achieve this incremental increase of functionality is by adding a separate exception process at the client level to handle each new feature provided by the server. This can be achieved if there is only weak cohesion between the processes, and has the desirable effect of increasing the concurrency of the system. Examples are the optional reconnection of a broken network connection in the author's ITI protocol implementation, described in Section 3.4.2.1, the ITI <BRK> exception handler, and the storage allocator in Mesa, described in Section 4.2.3.

By using *ignorable* inter-process exceptions, extra features provided by the server can be ignored by a client until a handler is provided. The feature is implemented when the client provides a handler for the inter-process exception. Ideally, the handler should be almost independent of the client code, so that the client and handler are only loosely-cohesive modules; this is good software engineering practice and it facilitates proofs of program

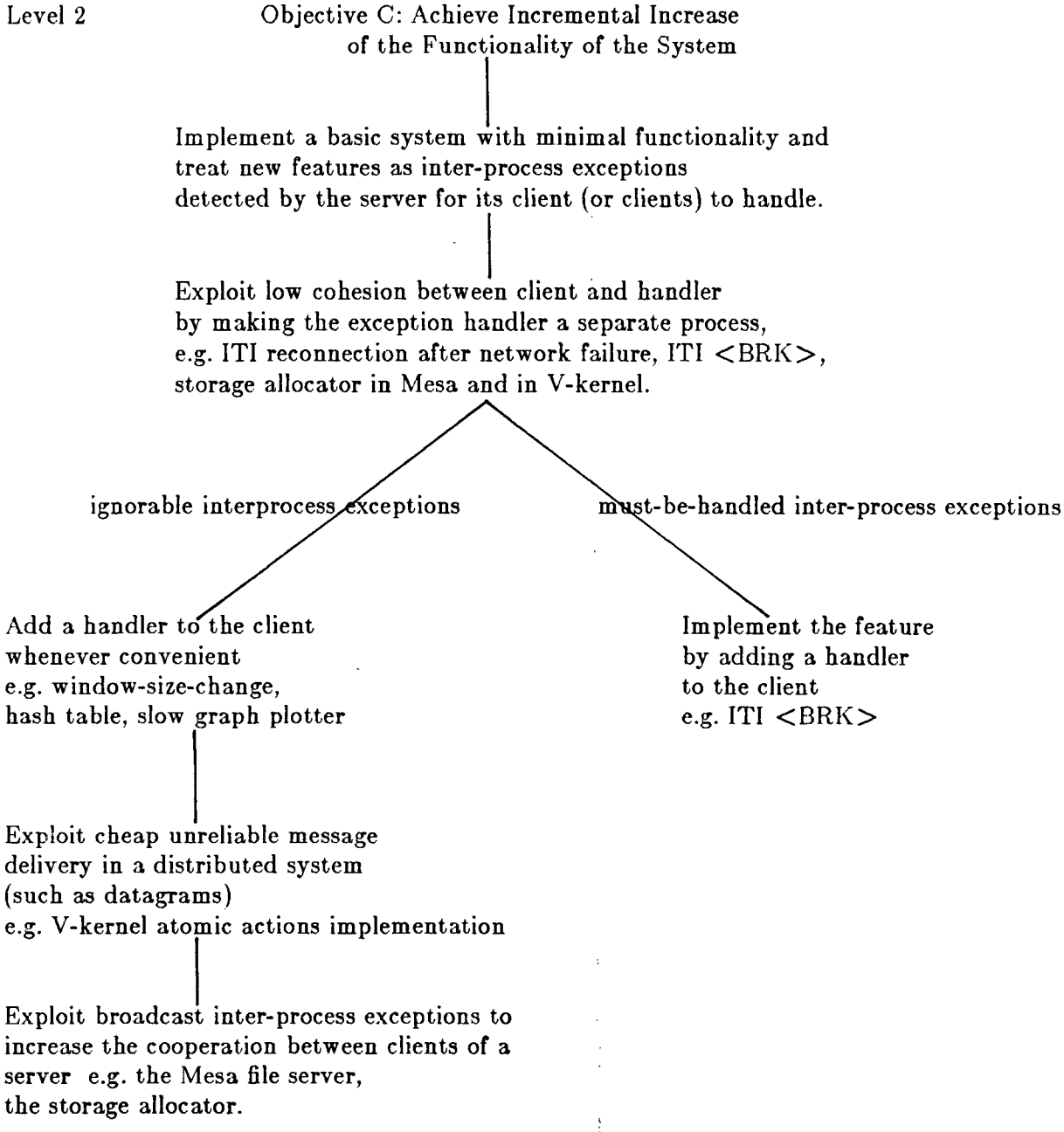


Figure 3.6 Model to Increase the Program's Functionality

correctness. For example, the author implemented and tested the ITI protocol without a handler for the <BRK> exception -- this was successfully added later, as described in Sec-

tion 3.4.1.1 below. Other examples of the use of ignorable inter-process exceptions are given in Section 3.4.2.2-Section 3.4.2.4.

In distributed systems where the cost of reliable message delivery is high, it can be advantageous to use a cheaper, but unreliable, message delivery system. An example is the author's implementation of atomic actions, described in Section 5.2.

One aim may be to provide increased cooperation between the clients of a server, for which broadcast inter-process exceptions are particularly useful. An example which achieves this objective through the techniques described is the Mesa file-server, described in Section 3.4.2.5. As the inter-process exception mechanism is system dependent, program paradigms are given in Section 4.2 for a storage allocator problem which uses broadcast inter-process exceptions to increase cooperation between clients, in different languages and operating systems.

3.3. Examples Illustrating Minimising the Average Run-time

First the costs of the program events must be analysed to determine which component costs are the most significant, so the appropriate branch of the model can be followed. A simple technique for uni-process programs has been developed by the author, and a typical analysis of a general program is detailed. This technique has proved very helpful in analysing uni-process programs and systems, and extensions of the techniques to multi-process systems have also been made by the author.

In a general event-oriented program, there is a set U of possible events that can occur during program execution, and a probability function p such that p(u) indicates the probability of event u, where $u \in U$. Efficient performance is important in software programs, and so a cost function, C, is also defined, such that the cost C(u) is the cost of executing the

program on receipt of event u.[6]

Then the expected cost of executing the program, C_T is given by:

$$C_T = \sum_{u \in U} C(u) * p(u)$$

It is assumed that the total cost of executing an event, C(u), is composed of several parts, including the cost of detecting that the event has occurred. By restructuring the program to reduce a component of the execution cost, such as the detection cost, the expected run-time may be reduced.

3.3.1. Partition the Event Set into 2 Groups to Reduce Detection Costs

3.3.1.1. Spell Example

Suppose there is an event-oriented program with events $u_1, u_2, \ldots u_{(n-1)}, u_n$. There is a simple test to detect the occurrence of each of the first (n-1) events, and the n^{th} event is assumed to be none of the others. Further, the tests for events 1 through (n-1) must be made, in turn, before the n^{th} event can be detected, and the n^{th} event is the most probable.

An obvious algorithm to execute this is given by:

```
loop
  getevent( event );
  if event = u1 then handle u1;
  else if event = u2 then handle u2;
    .
    .
    .
  else if event = u(n-1) then handle u(n-1);
  else handle un ;
endloop;
```

[6]We assume that this cost of executing an event is independent of program state. If this is not the case, then the event set may be increased to include compositions of program state with event, so that the constant cost assumption holds, as shown in the **putbyte** example in the next sub-Section.

An example of such a program is a spell utility program mentioned previously in Section 3.2, where each event is the next character from a text file, and all alphabetic characters are considered to be normal events[7]. Now if there are (n-1) word delimiters, events $u_1, u_2, \ldots u_{(n-1)}$ correspond to the character event being one of the word delimiters and event u_n corresponds to the character being an ordinary alphabetic[8]. Note that the word delimiters are *peer exceptions* in that they are transparent to the caller of spell, and they are handled within the utility program.

Assume the probabilities of the exceptional events are all equal[9] to p, as shown in Table 3.1, and the handling cost H is the same for each event. The cost of detection of the events is given in the second column of Table 3.1, assuming that each test takes 1 instruction. The total execution cost, (excluding the constant extra cost of handling each event) is shown as

Table 3.1 Execution Costs for a General Event Manager

event	probability of event	cost of detection	expected execution cost	new detection cost	new expected execution cost
u_1	p	1	1p	2	2p
u_2	p	2	2p	3	3p
.	.	.	.	.	
.	.	.	.	.	
.	.	.	.	.	
u_{n-2}	p	n-2	(n-2)p	n-1	(n-1)p
u_{n-1}	p	n-1	(n-1)p	n-1	(n-1)p
u_n	1-(n-1)p	n-1	(n-1)(1-(n-1)p)	1	1-(n-1)p
Total expected costs/event (excluding handling costs)			(n-1)-p(n-1)(n-2)/2		1-p+np(n-1)/2

[7]We assume that a simple test to see if a character is alphabetic cannot be made

[8]In our analyses we assume events occur independently of each other, so there is a constant probability p(u) of an event occurring. Violation of this assumption merely reduces the strength of the performance gains to be made from following our model, and does not affect the nature of the gain.

[9] In the spell utility, the character BLANK will appear as a word delimiter much more frequently than other special characters. For simplification in the general example, we assume that all exceptional events are equally probable.

column 3 of the Table.

To improve this program, the events are divided into 2 sets -- normal $\{u_n\}$ and exceptional $\{u_1, u_2, \ldots, u_{(n-1)}\}$. Then a dummy variable is introduced so that all the exceptional events can be mapped onto the one exception, to reduce the detection costs of the normal event[10]. Then the program could be written as follows: The cost of detection of the events is now given as column 4 of Table 3.1. If we assume as before, that the handling cost is H for each event[11] and the probabilities of the exceptional events are all equal to p then new costs are shown in the last column of the Table.

```
loop
  getevent ( event );
  if f(event) then HandleExceptions(events);
  else handle u_n ;
endloop;

HandleExceptions(events)
begin
 if event = u_1 then handle u_1;
 else if event = u_2 then handle u_2;
   .
   .
   .
 else if event = u_(n-2) then handle u_(n-2);
 else handle u_(n-1);
end;
```

[10]The choice of a dummy variable depends on the application. Suppose that all exceptional events can be mapped to return a value TRUE from some function, f (*event*), and all normal events to return FALSE. In particular, for the **spell** program, such a function could be a simple look-up table, ODDCHAR, with 256 entries, one for each possible 8-bit character, indexed by the character's bit-pattern. The entry in ODDCHAR for each delimiter is the value TRUE, and for each alphabetic character, the value is FALSE. Thus ODDCHAR[event] corresponds to f (*event*).

[11]without making further aggregations of constant-cost events

The last row of the table shows C_1 = total expected cost per event in the initial program and C_2 = total expected cost per event in the new program.

Then $C_1 = H + (n-1) - p(n-1)(n-2)/2$ and $C_2 = H + (1-p) + np(n-1)/2$, from columns 3 and 5 of the Table. The expected change in run-time, C_1/C_2 can be calculated for different values of the combined probability, np, of the exception events. Let $C_1/C_2 = \alpha$, the change in run-time. Then there is a reduction in run-time when $\alpha > 1$ and when $C_1/C_2 \geq \alpha$.

Now $C_1/C_2 \geq \alpha$ when $H + (n-1) - p(n-1)(n-2)/2 \geq \alpha(H + (1-p) + np(n-1)/2)$.

Let $np = y$. Then $C_1/C_2 \geq \alpha$ when

$$H + (n-1) - y(n-1)(n-2)/2n \geq \alpha(H + (1-y/n) + y(n-1)/2)$$
$$\therefore y \leq 2n(n-1-\alpha+H-H\alpha)/((n-1)(n\alpha+n-2)-2\alpha)$$

For the best possible speedup, when H=0,

$$y \leq 2n(n-1-\alpha)/((n-1)(n\alpha+n-2)-2\alpha)$$

$$\text{As } n \rightarrow \infty, np \rightarrow 2/(\alpha+1).$$

Values of $y = np$ against n are plotted in Figure 3.7 for α = 1.25, 1.5 and 2.

From Figure 3.7, it is seen there is a reduction in run time for all cases where $n > 2$. Significant reductions of more than half, when $\alpha = 2$ (and H=0), occur when the combined probability np of the exception events is less than 0.5.

To achieve a speedup of two times when H>0, the numerator in the above expression must be positive i.e. $n - 3 - H > 0$, so $H < n - 3$.

Thus the partition of the event-set U into two disjoint sets N and E, and the mapping of several exception events onto one to reduce detection costs in N roughly halves the run time

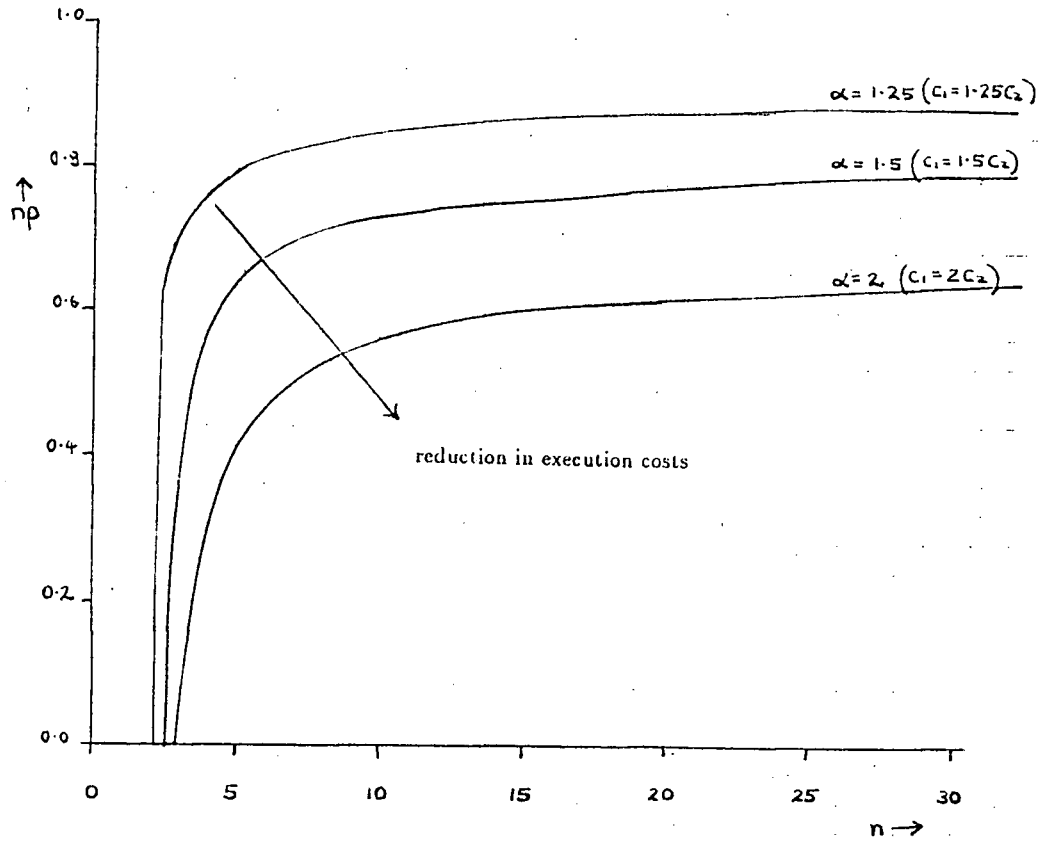


Figure 3.7 Expected Execution Costs for Different Values of np and n

when the handling cost for each event is small relative to the number of events[12].

3.3.1.2. Putbyte Example

The author has analysed a more complex example illustrating how exception detection costs have been reduced in a real program, **putbyte.** In the Verex operating system [Cheriton 80] the routine **putbyte** outputs a character to a selected output device. Cheriton reduced the average run-time of this routine by mapping several exceptions onto one using a

[12]Performance gains in testing for the common cases first have long been recognised, as reflected in the *FREQUENCY* statement in FORTRAN 1 and 11, which enabled the programmer to inform the compiler of the probability of various cases by facilitating flow analysis. However it was dropped from subsequent versions of FORTRAN as the statement required a tremendous amount of analysis at compile time and did not yield enough benefits at run-time. But as Sammet remarks in [Sammet 69], the fact that the statement became unimportant is **not** a reflection on the value of the initial technological contribution -- that of giving considerable thought to the possibilities of compiler optimisation of object code.

fix-up function, and hence reducing the detection costs. The author analysed the programs in detail, which are described in Appendix A.

The program was redesigned by dividing the events (each byte to be output) into 2 sets -- normal { **putbyte** called to a valid output device with the in-core buffer not full} and exceptional { **putbyte** called to an invalid output device, **putbyte** called to a valid output device with the in-core buffer full, and **putbyte** called with the END-OF-FILE character}. In the modified program the normal event is detected in just one test, instead of in two tests as in the original version. Because the handling costs are very small, a reduction in expected run-time from 5.75 to 4.75 instructions/event is achieved, which represents a significant saving for such a commonly used utility.

The **putbyte** function illustrates the principle of exploiting an existing exception-detection mechanism which catches peer exceptions -- namely the check made within **putbyte** for more space in the output buffer -- to catch a server-client exception occurring at a higher level, namely, there being no selected output. This saves run-time overhead on exception detection in the normal case -- alternative implementations incur the overhead of checking for selected output on every call of **putbyte**. Furthermore, it saves code.

The author has observed that this practice of exception detection at low levels is used extensively in high-level languages. For example, consider the DIVIDE operator. In most implementations the user does not bother checking if the dividend is zero before trying to divide - but the underlying hardware *does* trap the error, and, ideally, some mechanism is provided in the high-level language for the user to take suitable recovery action. This raises the issue of what level is most appropriate to catch and handle an exception; this example shows that it is sometimes practical to catch and handle it at the same level, whereas in later

examples we show it is sometimes appropriate to catch it at a lower level and handle it at the level above (as in the Newcastle Connection example of Section 3.3.2.3).

3.3.2. Reduce Handling Costs by Restructuring Programs

The division of the events into two groups for increase in efficiency of detection can be exploited further to make programs more efficient.

Suppose the total cost in running a program is given by:

$$C_T = \sum_{e \in E} C(e) * P(e) + \sum_{n \in N} C(n) * P(n)$$

where C (e) = the total cost of executing an exception event, and C (n) = cost of executing a normal event.

This must be minimised to reduce the average run-time cost. Now as many event-driven programs are driven by a small number of events, we assume that p(n) >> p(e). Thus reducing the cost of handling a normal event, C(n) should reduce the total cost, even if the cost of handling another, exceptional, event is correspondingly increased.

3.3.2.1. Os Example

An example of restructuring for the statistically dominant case to reduce average run-time (due to Cheriton), is a utility program **os** (for over-strike). **Os** converts a formatted text file containing backspace characters for underlining and boldfacing, to a file achieving the same printed effect using overprinting of whole lines, and containing no backspaces. Like the **spell** program described earlier, this data-driven utility can be treated as an event-driven program, where the input value of each character read represents an event. In its original structure, the program used multiple line buffers, one for each level of overprinting on the line. Each character except backspace and line terminators was placed in the highest level line

buffer that was currently unoccupied in the current character position. Thus, its structure reflected the logical operation it was performing, translating a character stream into over-printing line buffers. However, its structure also meant that the character it processed with the least overhead was the backspace character. Given that backspaces constitute about 0.5% of the characters in most text files, this program was structured inefficiently. Details of the program, and the author's analysis are given in Appendix B.

From the analysis of the event costs, the expected cost of execution of the program is $R + 1.25\,W + 13.71$ instructions/event, where R = average number of instructions needed to read a byte and W = average number of instructions to write a byte.

This program does NOT reflect any structuring for the statistically dominant case -- indeed it processes one of the least likely bytes (backspace) most efficiently. Cheriton undertook to rewrite the program. First, statistically speaking, the program is simply a null filter copying its input to its output unchanged. Starting with this view, the exceptions to recognize are *END-OF-FILE*, a *server-client* exception, and a *BACKSPACE*, a *peer exception* transparent to the user. This leads to a different program structure, also detailed in Appendix B, with an expected cost of execution of (R + 1.24 W + 3.8) instructions/event.

If W=R=4, a reasonable assumption for UNIX, this new program shows an analytical speed-up of 22.7/12.8, comparable to that of two times observed.

In the analysis, we find that the processing of normal-case characters changes from 9.85 to W+3 instructions/event in the new program, and as it is assumeed that W = 4, this provides an improvement. Therefore, some of the speed-up is achieved through the more efficient processing of normal characters. The rest is achieved by the much-reduced cost of processing the *NEWLINE* character - from (125W + 382) to (W + 3) instruction/event. Here analysis

shows that not only is the normal case processing in the restructured program more efficient, but also one of the so-called *exception* cases, *NEWLINE,* has been made much more efficient[13].

This example shows that by restructuring the program to reflect the statistically dominant case rather than the logical flow, it is possible to make major savings in run-time costs[14].

This particular example corresponds to one of Bentley's rules in his book *Writing Efficient Code*, [Bentley 82], called **Lazy Evaluation**: the strategy of never evaluating an item until it is needed, to avoid evaluation of unnecessary items, where, in the new version of the **os** utility, the multiple line buffers for overprinting are not established until they are needed.

A similar technique could be used for a high level language parser. In most high level languages, the most likely statement after an assignment statement is another assignment statement. Therefore the parser should hand code to the assignment statement handler first once an assignment statement has been found; if the parse then fails, the program should be able to back-up and recover.

3.3.2.2. Verex Name-server Example

To improve run-time efficiency of event-driven systems involving multiple processes, the system should be structured to minimise the inter-process communication or message-flow for the statistically dominant case, not necessarily for the logical or functional flow. By doing so,

---

[13]but at the increased expense of handling the BACKSPACE exception

[14]but note that if W is greater than 6, it would actually take *longer* to process normal case characters. Therefore the cost of the system-provided i/o instructions must always be carefully checked to see whether structural changes would be in fact beneficial.

the handling cost of the normal events is reduced.

As an example, consider a system design for a name-server. Any system resource accessed by name is checked by the name-server and passed to the controlling process. Ideally, the name-server would handle access requests to disc files by handing them to the file-server, input-output requests to the appropriate i/o device handler, and so on. However, this is too inefficient for file access (which is the most common resource request by name), so many systems have adopted the solution of two types of naming.

An alternative approach, implemented (by Demco) in the Verex operating system is to treat named requests for files as *normal events* and named requests for other devices as *exceptional events*. It is illustrated in Figure 3.8 below. All requests for named services made by a

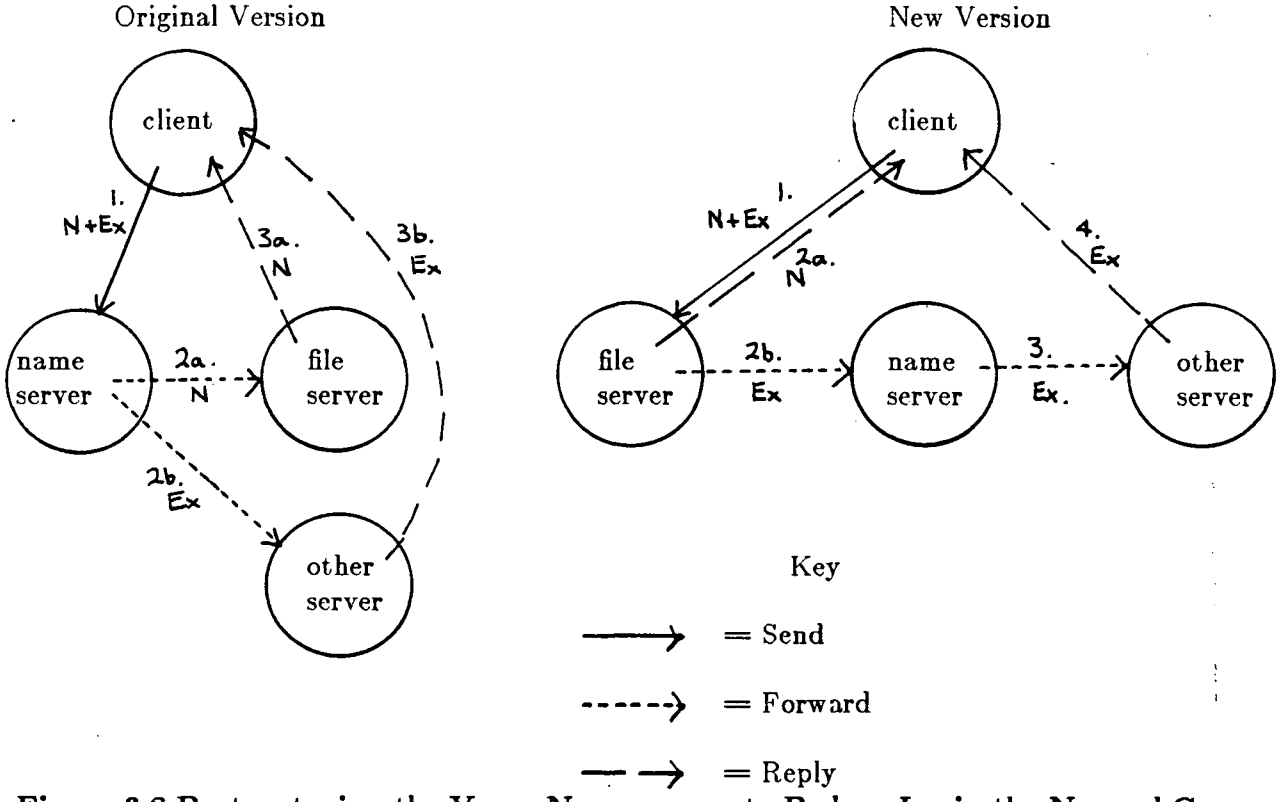


Figure 3.8 Restructuring the Verex Name-server to Reduce Ipc in the Normal Case

client go first to the file-server which handles file-name requests as before. The file-server detects exceptions *NOT-FILENAME* and handles them by executing a kernel request **Forward,** which forwards the message buffer to the name-server as if it came directly from the client. The name-server treats the request as before. In this way only 2 ipc messages are needed to access each file, involving 2 process switches. If 90% of named requests are for files, for every 10 requests there are 22 process switches -- 9 requests are handled by the file server in 2 process switches each, and one request is forwarded from the file server to the name server and then to the required i/o server, which makes the **Reply** to the client. With the original implementation 30 process switches would be used -- each request passes from the name server to the required i/o server which makes a **Reply** to the client.

The client is unaware of this implementation; hence client requests for named resources other than file names are treated as *request exceptions* by the file server (which previously only received file names from the name server). Thus restructuring has been achieved by adding *request exceptions* to the file server and removing the common *file-name requests* from the name server.

3.3.2.3. Newcastle Connection Example

The Newcastle Connection Distributed UNIX Operating System [Brownbridge 82] is another situation where the system could be restructured to improve its efficiency by reflecting the statistically dominant case rather than the logical flow. In this system, before each system call that uses a file is performed, a check is made, to see whether the file is local or remote. This check consists of a system call, **stat,** which returns **local** or **remote.** Local calls are placed unaltered to the underlying kernel for service; remote calls are packaged with some extra information, such as the current user-identifier, and passed to a remote machine

for service. Assuming at least, say, 90% of requests are for local files, this implementation uses the server-client exception *REMOTE-FILE* detected by the **stat** server (in UNIX the system calls **stat** and **open** act like servers) to control the client's subsequent actions. It introduces the overhead of a **stat** call whenever a file is accessed by name; this system call takes 1.6 msecs on a VAX 11/750.

An alternative approach, designed by the author, is illustrated in Figure 3.9 below. It eliminates the overhead in the normal case where local file access is required, by assuming all file accesses are local, and by making the client try a local file access first. In this case, the exception event *REMOTE-FILE* is represented by an additional *request exception* to **open,** the file-access server. The file-access server **open** detects when the request is of the exceptional form *NOT-LOCAL* (i.e. possibly remote, or erroneous) and handles it by making a

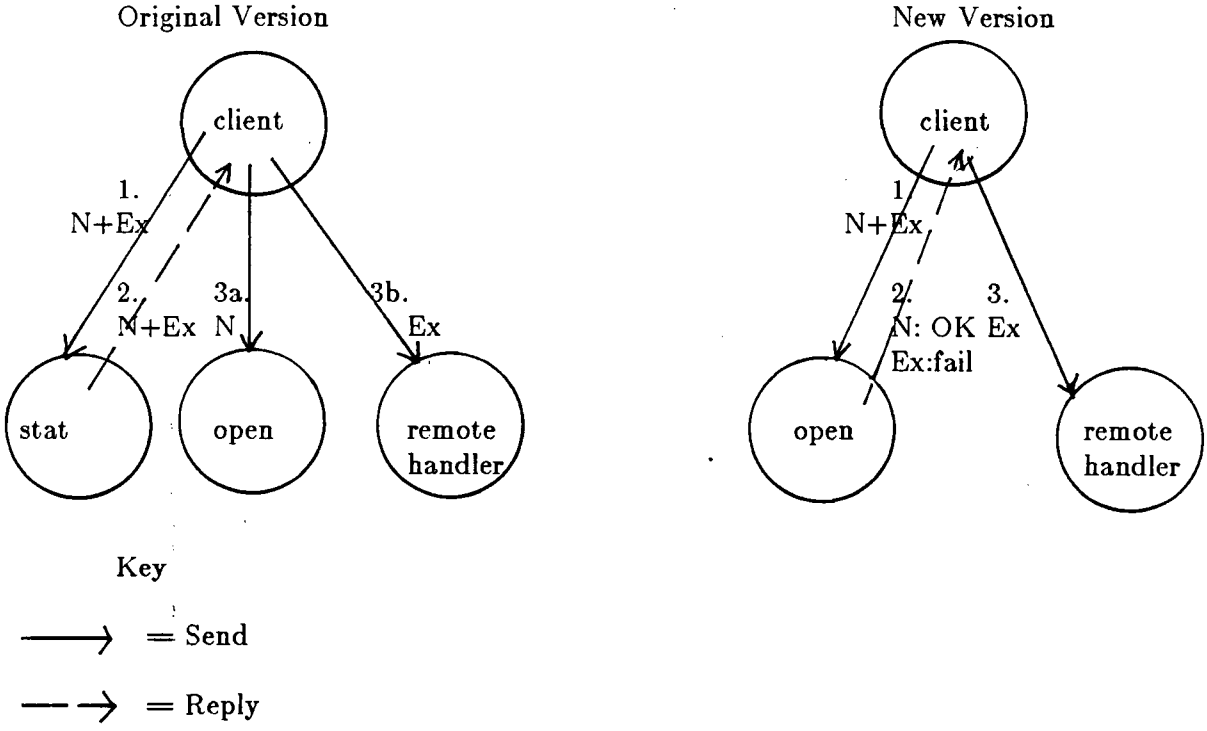


Figure 3.9 Restructuring Newcastle Connection to Reduce Ipc in the Normal Case

failure return from the client call. The client checks the reason for failure, and then re-tries as a remote file access. This saves the overhead of a system call on every local access at the cost of extra overhead for every remote access. This example also illustrates restructuring by adding a *request exception, REMOTE-FILE* to a server **open** which already has to check for the exception of the file not being found locally, and by removing all the common requests for local file detection from the **stat** server.

Both these examples illustrate how to reduce the context switching for the normal case by adding a *request exception* for exceptional cases to the server which is used for handling *normal* cases, and then directing all requests first to that server. It is a particularly good technique in these examples because the normal-case server already had to check for the exceptional cases.

3.3.2.4. The TROFF/TBL/EQN Example

This technique can also be used where a program suite consists of a system of multiple processes, where output from one process is used as input to another via a pipe, such as the UNIX word-processing system which consists of a suite of programs, TROFF, TBL and EQN, for various specialised tasks.

TROFF is a phototypesetting language which formats text documents for a photo-typesetter. It accepts lines of text interspersed with lines of format control information, and formats the text into a printable, paginated document having a user-designed style.

TBL is a document formatting preprocessor for TROFF. TBL turns a description of a table into a TROFF program that prints the table. It isolates a portion of a job that it can handle (viz. a table) and leaves the remainder for other programs, by isolating a table with delimiter pairs such as TS and TE.

EQN is a system for typesetting mathematics, and it interfaces directly with TROFF, so mathematical equations can be embedded in the text of a manuscript. This is done by surrounding an equation with delimiters such as EQ and EN.

If a text document contains tables and equations, some of which are embedded in the tables, it is processed in UNIX thus:

TBL <document> | EQN | TROFF

where | is the symbol for a UNIX **pipe,** meaning that the output from the program on the left of the pipe, is sent as input to the program on the right of the pipe. A pipe is implemented as a large buffer in kernel memory, which saves the overhead of creating a temporary file for the program's output. This provides a structured, modular system which reflects its logical action; the process TBL is a module that passes the next input character to the process EQN, which in turn passes the next input character to the process TROFF.

This way of structuring modules has been found to be successful in many applications. For example, a compiler may produce unoptimised code quickly for test purposes, or the output from the compiler may be passed as input to an expensive optimiser before final code generation for production programs. That is, the test compiler output may be passed as input to a code generator (unoptimised), or the test compiler output may be passed as input to a final code generator (optimised).

However, in many documents, most characters require no processing by TBL or EQN, so there is a performance flaw with this design. We now consider how such systems can be made efficient as well as nicely-structured.

The model proposes that the system can be restructured so the inter-process communication is minimised for the statistically dominant case. This can be achieved by treating lines

with no equations or tables as normal lines, and lines with tables or equations as exceptional. All lines are sent to TROFF first. TROFF detects when the line is of the exceptional form EQN or TBL and handles it by forwarding the line to the appropriate server, where it is treated as before -- which means that after processing by, say, EQN, it is passed back to TROFF for further processing. Thus the EQN-TROFF relationship is complex: TROFF effectively calls EQN which in turn calls TROFF again to process its output. To solve this problem of mutual recursion in a **feed-back** loop, the UNIX designers invented **pipes** and **filters** in the early 1970's[15].

The thesis model proposes that TROFF, EQN and TBL are each structured as simple unbuffered server processes, executing the following loop:

```
loop
  Receive(msg, id); **from the level above or from EQN or TBL
  process-as-necessary;
  Send(msg, to-server-below);
  Reply(msg,id );    **to the sender
endloop;
```

Normal text is processed only by TROFF. Suppose TROFF encounters an equation. Then TROFF passes it to EQN (in Thoth-like operating systems this may be done by executing a kernel routine **Forward,** which forwards the message buffer to EQN as if it came directly from the client), and TROFF completes its loop immediately instead of making a **Reply** to the client.

---

[15]Note that in the **os** utility the problem was solved procedurally by constructing line-buffers to hold the intermediate output processed in **HandleException.** The data was processed by writing in-line code, instead of making a call-back procedure. The problem with the text formatting example is that the size of the intermediate buffer for EQN and TBL is not known, and the code of TROFF is too large to write again in-line.

After EQN has processed the data it executes a **Send** to TROFF for the data to be processed further. TROFF recognizes that this data must not be forwarded; instead it processes it as usual, finally unblocking EQN by making a **Reply** to it, so EQN can then unblock the client process.

The author's design and analysis of both the pipe system and the restructured system design, for a Thoth-like operating system, are given in Appendix C. Analysis shows that for each 1000-line document there are 12000 context switches in the initial design. In the new structure, assuming 90% of the text is normal, and 5% is equations and 5% is tables, there are 4300 context switches, a reduction in process switches of nearly 3:1.

If a context switch is equivalent to n instructions, 7700n instructions are saved on processing a 1000-line document, a significant amount in most systems where context switching may take half a millisecond.

One could argue that the EQN|TBL|TROFF system could be improved by combining all the programs into one module, as the TEX word processor does [TEX 84]. But multi-process structuring is good design; our approach is to develop techniques to optimise it.

3.3.2.5. Point-to-point Network Protocol Example

In multiple-processor systems, further improvements can be made in run-time by distributing the exception handling code so it is situated on physically distinct processors. An appropriate technique is to implement each exception handler as a separate process. This allows the normal case code to continue without interruption, with a maximum saving in run-time equal to the cost of the exception handling. This technique can be used to advantage with handlers for peer exceptions and for server-client exceptions.

An application of this principle is found in the design of a point to point network protocol. In some protocol implementations, if the network becomes busy, a busy receiver simply discards incoming data packets, instead of trying to handle them by sending a message such as HOLD, which would take up more processor and network time. As the sender already has exception code to handle lost packets, the sender will use this mechanism to retransmit at some later time. This may help the receiver to reduce the congestion; further, in some dynamic routing algorithms, this induced delay on the channel to the congested node will reduce its preference as a route, further reducing the congestion.

From the view of the model, this example illustrates the principle of minimising the cost of handling a critical real-time exception by mapping one exception (RECEIVER-BUSY) at the receiver onto another (LOST-PACKET), handled at the sender. This shows how peer protocols can map peer exceptions between peer processes.

This example also illustrates how an implementor can choose the event (in this case, the exceptional *RECEIVER-BUSY* event), to be handled most efficiently, so that recovery is possible. It also leads to a general principle of how exceptions may be handled simply, by translating them into exceptions that are already handled by another mechanism. The protocol runs on two distinct processes, and when one, the receiver, becomes busy, the exception handling is effectively forced onto the other viz. the sender, thus saving run-time at the busy node. However, most programs and algorithms are not structured to exploit such concurrency, and the problems of coordination of exception handling and normal case processing are not yet well understood.

On a single-processor system there are obviously no gains in run-time to be made by executing the exception handler as a separate process, but there are clear logical advantages, viz:

(1) good separation of normal case code from exception code

(2) context saving of the normal case is locked in the well-defined state of a process

(3) experience gained in structuring programs this way will be useful in designing distributed systems which exploit true concurrency.

In the operating system Medusa, the exception handling code can be specified as another process; the **buddy** of the **victim** which incurs the exception. This feature was described in Section 2.2.6.3, and it was shown to be useful in certain situations, such as in remote debugging.

3.3.3. Reduce Context-saving Costs in Normal Events

The third approach to improving efficiency through dividing event-oriented programs into normal and exception events, is to reduce the context saved in the normal case appreciating that in some critical real-time programs, rapid exception handling may be needed which may conflict with this objective.

3.3.3.1. Read-back Example

For a simple application of the principle of reducing context-saving in the normal case, consider the **os** program of Subsection 3.3.2.1. In the revised program, the column count on the current line is maintained with every byte read, in order to handle the backspace exception, thus adding the overhead of context saving to the normal-case processing. This amounts to 1 instruction per byte which equals 6% of the total handling cost of 13 instructions/event. By maintaining this context, the exception handler is quite straightforward to write.

As an alternative, the normal-case context saving could be eliminated by pushing more work onto the **HandleException** routine, for example, making it *read backwards* to check

the last occurrence of the newline character. This illustrates a tradeoff between maintaining the context up front so that exception handling is relatively simple, or by maintaining little or no context, and making the exception handling more complex. This could be automatically and efficiently maintained by the system, if there was a way of specifying it in a program. Such a facility could be used as illustrated in the program fragment given at the end of Appendix B.

3.3.3.2. LNTP Example

A more complex example is now described, showing how improved efficiency may be obtained through reduction of context-saving for exception handling in communication protocols. Communication protocols can be treated as event-oriented programs, each layer of which is typically implemented as a finite state machine making state transitions in response to the messages received from the layers above and below. In communication protocols, much of the code deals with error detection and recovery. In long haul networks, (LHNs), characterised by a long network delay (low bandwidth) and high error rate, there is a high overhead on normal case processing[16] of virtual circuit point-to-point connections, to maintain context so that errors can be handled without a breakdown of the virtual circuit. For example, the ARPANET protocol TCP/IP [DARPA 81] was designed to provide internet packet transport over error-prone subnets. The services provided include internet address handling, routing, internet congestion control and error reporting, multiple checksums (one at each the TCP level and the other at the IP level), segment fragmentation and reassembly, datagram self-destruction mechanisms and service level options to clients. The large, byte level sequence numbers used in TCP (32 bits) incurs considerable processing overhead [Chanson 85a] but is

[16]We assume normal case is when there is no lost or erroneous data

justified in the context of a LHN on the grounds that a large recycling interval allows easy identification of out-of-order and duplicate segments, and that byte level sequence numbers facilitate the fragmentation of segments at the local IP layer and the intervening gateways, and the reassembly at the remote TCP entity.

In contrast, a single local area network (LAN) is characterised by a low transmission error rate, by single-hop routing and high channel speed such as 10 Mb/sec, compared with a limiting speed of about 9.6Kb/sec for a long haul network. In a local area network, the major portion of the packet delay time (time to process and successfully deliver a packet to its destination) shifts from the transmission delay time of long haul networks to the protocol processing time. Thus the efficiency of the protocols is an important design issue for local area networks. The overheads in maintaining context for correct error handling of a connection oriented protocol such as TCP/IP are unjustifiably high for a local area network as the error rates for local area networks are so small (less than 10^{-9}). The overhead of layered protocol implementation has been described in [Bunch 80] -- adding protocol layers adds overhead.

One solution, adopted in [Chanson 85b], in which a virtual circuit is implemented by a very simple protocol for point-to-point data transfer, applies this principle to reduce normal-case run-time. This protocol, called LNTP (Local Network Transport Protocol) takes into consideration the characteristics of LANs. LNTP runs under 4.2 BSD UNIX replacing TCP/IP for local communication[17]. Since the majority of the packets in a LAN are for local consumption, this scheme greatly improves the network throughput rate as well as the mean packet delay time.

---

[17]When packets are destined for other networks supporting TCP or some other protocol, the protocol can be implemented at the gateway.

The fundamental philosophy in the design of LNTP is simplicity. The objective is to reduce the protocol processing time, in addition to improving understandability and ease of maintenance. Thus we set out to design a new protocol that includes only the features strictly required in a single LAN operating environment. Any functions that are needed only in rare occasions, particularly if they can be easily achieved by the application programs, are not included. Therefore, internet congestion and error control, routing, service options provided to high level protocols and certain functions that handle damaged packets found in a typical LHN protocol are not included as they are irrelevant in a LAN environment. Even checksumming is specified as an option since the error rates of the communication medium are negligible. The only mandatory error control feature of LNTP is the selective retransmission scheme to handle packet loss due to buffer overflows. Consistent with the characteristics of a single LAN environment, the peer address (16-bit address space) does not include internet component. Furthermore, since the probability of out-of-order delivery and duplicates is negligible, the sequence number space is made small (4-bits) -- thus detection of lost packet exceptions is achieved using very simple sequence numbers.

To reduce the state information required to maintain a connection (which simplifies the control structure leading to reduced processing overhead), the sender and receiver are logically separated and the send and receive channels are completely decoupled. A consequence of this decision is that a receiver is unable to piggyback control information on reverse data packets to the sender, a feature commonly supported by LHN protocols to conserve communication bandwidth. However, network bandwidth is not a scarce resource in a LAN. Moreover, our measurement results [Chanson 85a] show piggybacking is rare due to the unavailability of reverse data packet at the right time. Thus in view of the simplicity and reduced processing overhead, LNTP is asymmetric in send and receive.

The flow control mechanism in LNTP maximises the parallel operations of the sender and receiver, and minimises the number of control packets that has to be exchanged. The concept of threshold in the window space reduces the control traffic (no control before the threshold is exceeded). Because of the deterministic nature of packet delays, a mathematical model can be formulated for a proper threshold to be set as a function of the system parameters to maximise the network throughput rate.

LNTP implements a single logical timer; in contrast, almost all other protocols specify a separate timer for each outstanding packet (though some implementations cut corners in this).

A preliminary implementation of LNTP in the 4.2 BSD UNIX kernel running on a SUN workstation has been made. The protocol was tested in a software loopback mode, and its performance compared to TCP/IP. In UNIX 4.2BSD, the file transfer rate is increased to 450 kbits/sec compared with 360 kbits/sec for TCP/IP running under identical conditions. This improvement is basically due to the simplicity in the control structure resulting in lower protocol overhead for LNTP.

This design illustrates the reduction of average run-time by reduction of context maintenance specifically for error-handling, and by tuning a protocol's flow control to be most efficient for normal error-free data transfer.

3.4. Examples Illustrating Increasing the Functionality of the System

3.4.1. Using Inter-process Exceptions from Server-client

3.4.1.1. ITI <BRK> Example

An example of the technique for providing incremental increase to a system's functionality was made by the author in implementing the transport layer protocol X.29 [CCG 78], here referred to as Datapac's Interactive Terminal Interface (ITI) protocol. ITI uses the services of the network layer protocol X.25 [CCITT 76] in the Verex operating system. The author designed the ITI protocol as an input/output filter consisting of a reader process and a writer process [Atkins 83b]. This minimal configuration supported normal read/write requests, but did not support the exception event of a <BRK>. The author was able to implement and test this basic system without any exception handling facilities. An exception process for handling <BRK> was then added to the ITI layer and the system retested with the <BRK> exception. This example illustrates implementing a new feature by making the server export an inter-process exception to a loosely-cohesive handler at the client layer. It is described fully in Chapter 5.1. It also shows how concurrency is increased through multi-process structuring.

3.4.2. Ignorable Exceptions

So far, only those exceptions which **must** be handled for correct operation of the system have been considered. However, certain exceptions, which we call *ignorable* exceptions, may be used for notification only; if there is no handler to receive the notification, the system still runs correctly, though maybe with reduced functionality or less efficiently.

3.4.2.1. ITI Reconnection Example

As already mentioned in the previous Section, the author implemented a transport layer protocol ITI which uses the services of the network layer X.25 protocol.

It was observed that on several occasions a remote user would lose his connection, and would immediately log on again, only to find that the program he was working on had been destroyed because of the broken virtual circuit. It therefore seemed desirable to provide an ITI implementation and session-layer service which would hide a failure in the underlying network from the application layer until either a timeout fired, or the same user logged in again. In the latter case, the application program will be available to the user just as if no disconnection had occurred (except for a brief RECONNECTED message). The author decided to exploit an ignorable inter-process exception from the X.25 server to the client ITI to provide this incremental increase to the system's functionality.

As already discussed, the ITI layer already had a separate exception process to handle the <BRK> exception. The author decided to use the same exception process to handle the new feature which would achieve reconnection as shown above, by exploiting an ignorable inter-process exception from the X.25 layer below. The author implemented the ignorable inter-process exception as a non-blocking **Reply** from the X.25 server to the ITI exception process whenever the network connection was lost. The exception process was modified to handle the reconnection. The technique used is detailed in Section 5.1.5. This is a powerful example of increasing the functionality of a client/server system incrementally by using ignorable 1:1 inter-process exceptions.

3.4.2.2. Window-size-change Example

Consider how a window management program could use an exception mechanism to handle a WINDOW-SIZE-CHANGE notification. If no handler exists for this exception, the window manager takes no action for line size truncation or expansion. If an exception handler exists to handle this notification, a more intelligent action such as line wrap-around may be

taken. This illustrates increasing a program's functionality through the use of 1:1 ignorable inter-process exceptions.

3.4.2.3. Hash Table Example

In many software systems, a program may be made more efficient by being able to invoke an exception mechanism which runs an alternative algorithm under certain exceptional situations. If no exception mechanism is present, the program still runs correctly, but more slowly. As an example, consider a program which implements a hash table. If the hash table entries get too long, say greater than 100 entries, an information signal is sent to the caller of the routine. If the caller wishes to handle such a signal, it can call an alternative algorithm (i.e. an exception handler) for hashing the table entries, which will reduce the table entries to a reasonable number, allowing subsequent calls to proceed more efficiently. If no alternative algorithm is available, the notification signal is simply ignored by the caller. This approach could be used in a compiler implementation.

3.4.2.4. Slow Graph Plotter Example

Another example where an alternative algorithm could be useful, would be for a slow graph plotter. If a large plot were requested, and there was already a long backlog of work, an exception notification could be made to the caller. The caller could then invoke a time-consuming algorithm to optimise the plotter steps for that job, thus reducing the plotting time estimate. If the caller provided no alternative algorithm, the plot would still be correct, but would take a long time.

This approach could be extended in a distributed system so that alternative algorithms for calculating the plotter steps would all execute concurrently. When the plotter became

available to plot this job, the algorithm which had completed with the fewest steps would be chosen, and the other algorithms would be aborted.

3.4.2.5. Mesa File-server Example

The Mesa file server described previously in Chapter 2 illustrates how a program that provides certain basic default facilities, may be subsequently enhanced by using notification to an exception handler. The program runs to a minimal specification without the exception mechanism, and may provide further functionality using the exception mechanism. If the client does not provide a handler for the **PleaseRelease** notification, the notification is simply ignored. Otherwise, the client provides increased functionality by attempting to release its files. This is an example of a increasing the cooperation between clients by using ignorable broadcast inter-process exceptions.

CHAPTER 4

Program Paradigms for Implementing System Dependent Features

This Chapter presents program paradigms employing the design principles described in the general model for two example problems. The program models are given for different operating systems, and in different high level languages. High level languages are discussed because concurrent programming is the basis of both applications and systems, and one wishes to see a uniformity through all levels of the system. Many designers favour a language-based approach to distributed operating systems, with the aim of providing uniform access to user and kernel supplied resources, both locally and remotely. For example, programmers at the University of York in England have decided to use ADA [USA 80] to implement a distributed UNIX, and they will only allow users ADA-like tasks, not UNIX-like processes.

The model, shown in Figure 3.3, proposes designs for achieving 3 objectives: viz. A, for reducing the average run-time costs; B, for reducing exception handling costs; and C, for increasing program functionality. For all these objectives, a general-purpose and efficient inter-process exception mechanism is necessary. For objective A, in a multi-process environment, it is important to ensure that the inter-process exception notification costs do not overwhelm the normal case computation costs. Minimising the inter-process exception notification costs is a declared part of objective B (see Figure 3.5). And for achieving incremental increase of a program's functionality, a general purpose inter-process exception mechanism is essential (see Figure 3.6). Mechanisms to achieve this are system dependent, so a general model cannot easily be described. Instead, a set of program paradigms is presented

for achieving inter-process exception notification. The 1:1 inter-process exception notification is illustrated by a problem which often arises in concurrent programming involving exception notification of asynchronous events -- that of a window manager process which multiplexes i/o from 2 or more devices simultaneously -- the so-called KEYBOARD-MOUSE problem.

The model also proposes the use of ignorable exceptions for increasing a system's functionality; again, the implementation of ignorable 1:many inter-process exception notifications is system dependent, so a set of program models for exploiting such exceptions is described. The problem chosen is that of a storage allocator client/server system, where the cooperation between the clients is enhanced by the server's ignorable, broadcast inter-process exceptions.

Finally, this Chapter discusses program models for achieving mutual exclusion between processes, as it is often logically advantageous to use a separate exception handler process to receive ignorable signals synchronously, rather than use a procedure in the client process, to increase concurrency and to emphasise the independence of the exception handling code from the normal case code. The implementation of mutual exclusion is system dependent, and several program paradigms are given for implementing the mutual exclusion necessary when exception handlers are implemented as separate processes. We show that atomic transactions are helpful for implementing exception handlers as separate processes.

4.1. Program Paradigms for Inter-process Exception Notification

4.1.1. The KEYBOARD/MOUSE Example

Efficient and general inter-process exception notification mechanisms are needed to reduce exception handling costs in multi-process systems, and also to distribute exception handling by using multiple processes, where the invoked process (the exception handler), is

not necessarily related to its invoker. The example problem is concerned with synchronising simultaneous input from two or more input devices, such as a window manager process which multiplexes input/output by reading input from a keyboard and a mouse simultaneously. The problem arises in ensuring that an exception such as <BRK> is processed within a critical real time, whilst other i/o events are processed as expediently and fairly as possible. Furthermore, it should be possible for the user to control the arrival of input from the various i/o devices. For example, when no mouse input is required or expected, the window manager does not wish to receive input from the mouse whenever it is accidentally moved, as processing such messages could slow down the system considerably. Thus the keyboard input may be considered as normal-case, and mouse input as exceptional.

A new solution for this problem derived from the model is described for Thoth-like operating systems. We then discuss how modern concurrent programming languages allow one to model these designs, within the framework of the three types of concurrent programming languages described by Andrews and Schneider in their survey *Concepts and Notations for Concurrent Programming* [Andrews 83]; viz. monitor-based languages, message-oriented languages and operation-oriented languages.

4.1.2. Exception Notification in a Synchronous Message-passing System

In the synchronous message-passing environment of Thoth-like operating systems, asynchronous input-output can be readily modeled by a window manager with 2 workers, KEYBOARD (KB) and MOUSE (i.e. one worker for each kind of asynchronous event). This is shown in Figure 4.1 below.

The device server X at the level below demultiplexes the input and makes a **Reply** to the appropriate worker. The window manager receives notification fairly when input is ready

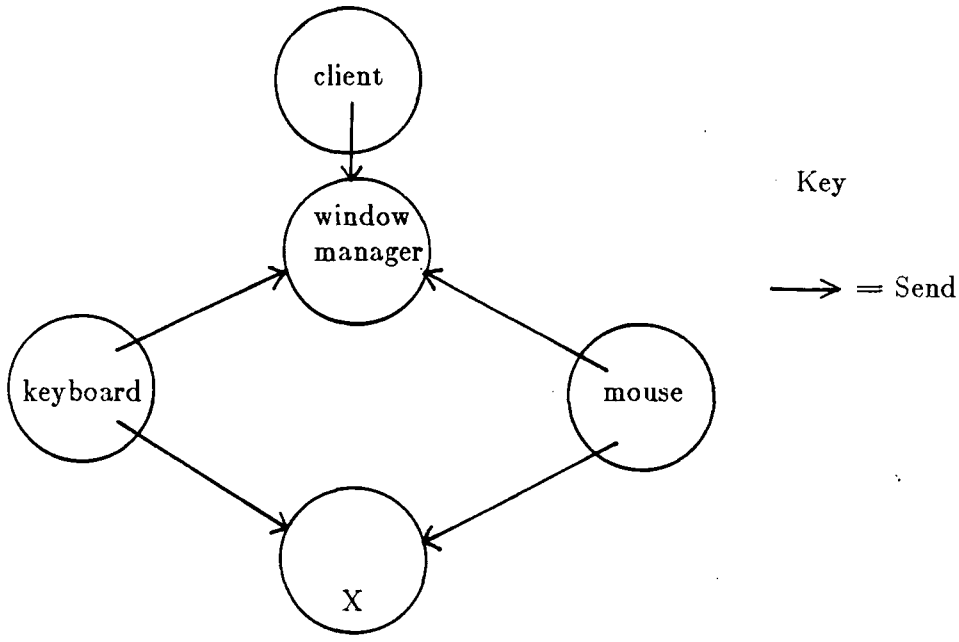


Figure 4.1. Thoth Server Notification Using Two Worker Processes

on each device *provided the operating system is fair in its scheduling.* Thus the problem of scheduling has been pushed down to the level below the server.

For a message-based system, it is not easy to provide an efficient way to control the possible input devices selected. For example, if the mouse reader reported each move to the window manager, including accidental moves from the user, there would be 2 process switches for each move, which would slow the system down considerably.

We propose a solution to this problem, in which the window manager provides a switch for its clients to toggle *on* or *off* via a simple request. The window manager transmits this request to the level below, say (server) process X. This is illustrated in Figure 4.2.

Let us call the client requests to toggle the switch **PleaseNotifyMouse** and **NoNotifyMouse.** The server X must maintain a toggle for each input device. Whenever X has input, X checks its toggle for that input device (initially all are *on).* If the toggle is *on,* X

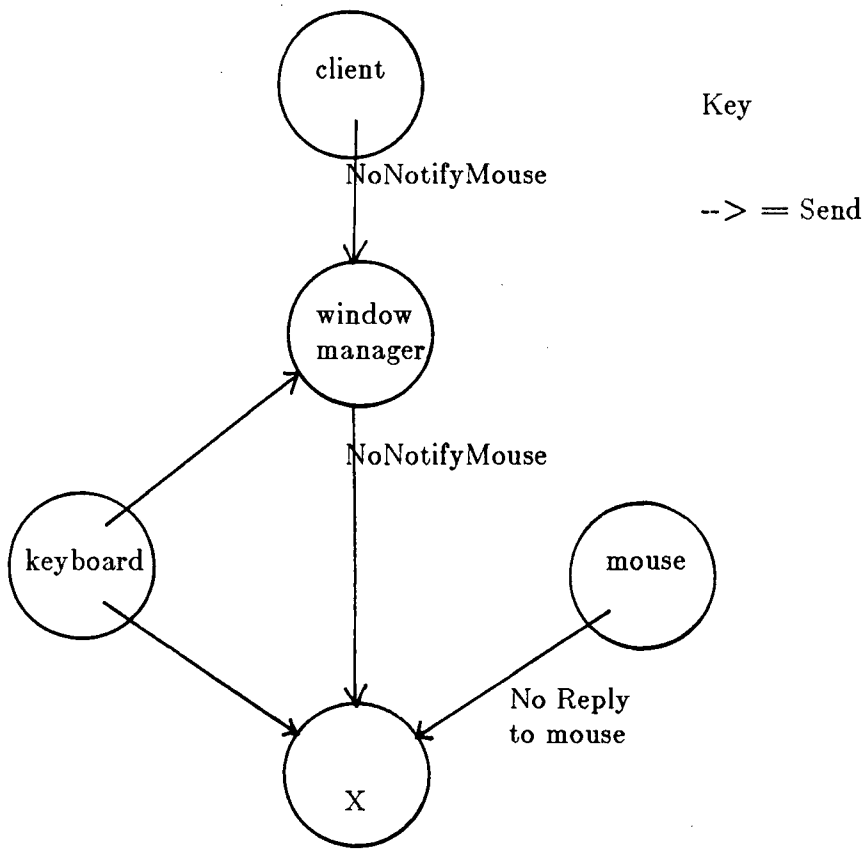


Figure 4.2. Thoth Switch Notification

demultiplexes the input and replies or queues it for the appropriate reader as before. After the client makes a **NoNotifyMouse** request to the window manager, the toggle for mouse i/p in X is changed to *off*. When the toggle is *off* X simply discards its input data from that device. Only when the client makes a **PleaseNotifyMouse** request to the window manager will the mouse data be sent to the level above. The masking of the mouse data can be pushed as low down as required with appropriate additions for the toggle at the client-server interface at each level. This solution has been implemented by the author in another context, that of reading <BRK> interrupts from a remote user over the X25 network. In that context, the toggle had to be reset after *every* data input, causing an overhead of 2 extra process switches on every normal case input. The author deemed it was too inefficient a solution, and

designed another solution for this problem employing an inter-process exception from server to client, which is described fully in Section 5.1. Thus the toggle switch tool provides an efficient and convenient mechanism for masking asynchronous events, *providing that the switching is performed only rarely with respect to the receipt of normal input items.*

4.1.3. Exception Notification in Procedure-oriented Concurrent Languages

In *procedure-oriented* languages (or, equivalently, *monitor-based*) such as Mesa [Mitchell 79], used to implement the single-user operating system PILOT [Lampson 80] and Modula-2 [Wirth 82], used for developing a real-time operating system called MEDOS on the Lilith computer, monitors were designed to provide a very cheap method of inter-process communication [Hoare 74]. In the Modula and Mesa languages the monitor approach often necessitates that a process finishing with a resource must explicitly **notify** (or **signal**) other processes waiting for it, so they will be awakened. These other processes are said to be waiting on a *condition variable.* Such use of a condition variable, or global flag, is unstructured and difficult to program correctly. The notification is easily forgotten as the **notify** is not an essential part of the releasing process's action; one might even criticise this as being a violation of the abstraction of the process -- such implementation details should not be the concern of the programmer at that level, but should be hidden by being implemented at the level below. To overcome these difficulties, the Mesa implementors placed a timeout on every condition variable so that missing **notify** signals would not block processes for ever. An experiment which increased this system-set timeout from 200 msecs to an effectively infinite value revealed that many such notifications were missing [Levin 82]. Further difficulties with the interaction of monitors, processes and exception signals have been described by Lampson in [Lampson 80].

Now a **notify** signal in a monitor will only wake up processes which are already waiting on that condition variable; it cannot be used for general-purpose inter-process communication. In Mesa, for inter-process exceptions the only signal is ABORT. Thus there are problems in using monitor-based languages for distributed systems where exception communication is required in general between unrelated processes. However, a program model to achieve this in Mesa is described in Section 4.2.2.

Wirth discusses the problem of multiplexing asynchronous input from a keyboard and a mouse in his book on Modula-2 [Wirth 82]. He suggests that the user execute a **BusyRead** which in contrast to the conventional **Read,** does not wait for the next keystroke on the keyboard, but instead immediately returns a special value if no character is found. The code for detecting whether input has occurred consists of a loop within which is a test to detect whether a character has just been read and the mouse has moved, followed by a call of **BusyRead** which returns either a character from the keyboard or the special value. Such a *Busy-Wait* loop is clearly only practical on a dedicated single-user processor as cpu cycles are unavailable for other work. Of course, one could then resume a background process after these tests. A better approach is to have a multiplexor which polls for the various events, run once every clock tick. The Macintosh uses this technique of one consolidated event queue, managed by the operating system. The user executes **get-next-event** and pauses there until an i/o event occurs [Macintosh 84]. Thus we observe that in procedure-oriented concurrent languages, one software tool useful for inter-process exceptions is an event queue managed by the operating system.

4.1.4. Exception Notification in Message-oriented Concurrent Languages

Andrews and Schneider define message-oriented high level languages as those which provide **send** and **receive** as the primary means for process interaction, and they assume no shared memory between processes. Message-oriented languages such as CSP [Hoare 78] and PLITS [Feldman 79] are intended for developing multi-user operating systems. The author observed that in the message-passing synchronization primitives the synchronous **Send-Receive-Reply** inter-process communication primitives of Thoth were not mentioned; that omission was remedied in a subsequent letter to *Surveyor's Forum* [Atkins 83c].

It has already been shown how the synchronous Thoth-like environment may be used to model the problem of multiplexing asynchronous input from two or more devices; however, Thoth processes on the same team may share memory. We must therefore ensure that the problem can be modelled without the use of shared memory, if this program design is to be applicable to a high level language such as PLITS or CSP.

The main use of shared memory is to save copying data, but data can obviously be transferred by messages if necessary. The second use of shared memory is in implementing synchronization between processes on a team; this can be achieved by a semaphore. Hence the solution outlined above for Thoth will work directly in message-oriented languages where no shared memory is available.

Nelson remarks in his thesis on *Remote Procedure Calls* [Nelson 81] that the need for asynchronous exceptions between processes has long been recognised and implemented in the message-passing world: PLITS, Medusa and RIG all provide some method of inter-process asynchronous exception. He continues, by noting that the RIG implementors, in particular, found this capability essential for designing reliable systems. However, it is shown here that

the toggle switch mechanism provides a suitable *synchronous* inter-process exception notification tool for message-based systems; thus *asynchronous* notification is not absolutely essential for good performance.

4.1.5. Exception Notification in Operation-oriented Concurrent Languages

Operation-oriented languages provide remote procedure call as their primary means for process interaction. ADA [USA 80] and SR [Andrews 82], intended for designing multi-processor systems, are examples of such languages. As in procedure-oriented languages, operations are performed on an object by calling a procedure. The difference is that the caller of an operation synchronizes with the so-called *caretaker* that implements it while the operation is executed. In ADA this synchronization is called a *rendezvous*.

A rendezvous presents problems for general inter-process communication as the server-task can only be communicating with one active process at a time, allowing for no asynchronous inter-process notification. For inter-process exceptions in ADA only the FAILURE signal can be given, so this mechanism cannot be used for general notification. A server process in ADA can multiplex calls by nesting **accept** statements that service the calls. This technique is not yet widely used, and as Andrews and Schneider point out, implementation of some of the concurrent programming aspects of ADA is likely to be hard.

SR (Synchronizing Resources) [Andrews 81,82], also uses the rendezvous form of the remote procedure call. However, both asynchronous and synchronous message passing are supported, and Andrews has achieved successful implementations of servers in SR using these operations. Only time will tell if the lack of specific software tools in ADA for inter-process exceptions will hinder its growth as a software language.

4.1.6. Exception Notification in Distributed Systems

In distributed operating systems where the user process may be located on a different host to the one the terminal is connected (e.g. during a remote login session), a mechanism must be available to divert the asynchronous interrupt to the remote host, as one of the special difficulties with asynchronous inputs in a distributed system is the lack of shared memory. This means that a hardware interrupt on one processor can't force a transfer of control to a process executing concurrently on another processor -- some kind of *software interrupt* facility is needed.

In communication protocols such as X.25 [CCITT 76], a special *out-of-band* message (an interrupt packet) is used to transmit such priority information to the remote host, and this must be received as soon as possible by the target process. An interactive terminal interface protocol is used on the remote host to simulate the terminal; however this may run at a much higher level than the local terminal server. Special facilities may therefore be required to achieve the desired effects on <BRK>. The author designed a mechanism, discussed in the ITI network protocol example in Chapter 5, to solve this problem in both Verex and UNIX.

4.2. Program Paradigms for Ignorable 1:many Exceptions

4.2.1. The Storage Allocator Example

In his thesis Levin proposes an important, interesting mechanism for inter-process exceptions, by designing into a high level language a means for the owner of a shared abstraction (typically a server) to notify ALL (or some number) of its clients that an exception has occurred. He calls such exceptions on a shared object *structure class* exceptions. All the processes that have used, and are still using the service may receive exception signals, as

specified in the server's functional description. Thus a module *exports* an exception to its users. This feature was not implemented by Levin, but we show here that it can be implemented, and that it could be extremely useful, particularly in distributed operating systems.

We develop several program paradigms based on the example problem of a storage allocator which may detect that its resources, while adequate to meet the current request, are running low. Such a POOL-SCARCE exception condition could be usefully propagated to all contexts where resources might conceivably be made available to the allocator. Yet there is no reason to suspend the current allocation request while the condition is being handled; the two actions are logically independent and should proceed (conceptually at least) in parallel. Hence the exception POOL-SCARCE is described as an ignorable asynchronous broadcast exception in our classification.

The allocator may also find it impossible to satisfy a resource request, and may raise a POOL-LOW exception with its clients, expressing its urgent need for storage. Naturally, the request will remain pending while this exception is being processed. POOL-LOW is also classified as an ignorable asynchronous broadcast exception. The allocator, even after the above, may not possess adequate resources to satisfy the request. In this case, it must raise a POOL-EMPTY exception, signifying its inability to meet the requestor's demands. POOL-EMPTY cannot be ignored by the client making the request; thus POOL-EMPTY is classified as a 1:1 synchronous exception which must-be-handled. Levin's proposed allocator is shown in Figure 4.3 below[1]. Because Levin's so-called *sequential-conditional* selection policy (described in detail later) could cause deadlock if the pool module is implemented as a moni-

[1] We have altered the notation from Alphard [Wulf 76] to a Mesa-like notation in which Levin's **condition** variables are replaced by **exception** variables to avoid confusion with monitor condition variables

```
StorageAllocator: MODULE =
begin
        p               : pool;
        free            : integer;
        exception       : POOL-LOW      policy sequential-conditional;
        exception       : POOL-EMPTY    policy broadcast;
        raises POOL-LOW on pool, POOL-EMPTY on Allocate;
        ...

        Allocate:       ENTRY PROCEDURE (p:pool, size: integer)
        returns (d:descriptor) =
        begin
                P(outer); P(inner);
                if  p.free < size  then
                begin
                        V(inner);
                        raise POOL-LOW until p.free >= size;
                        P(inner);
                        if p.free<size then
                        begin
                                V(inner); V(outer);
                                raise POOL-EMPTY;
                        end;
                end;
                p.free := p.free - size;
                < create descriptor d for segment of amount size >
                V(inner); V(outer);
        end;

        Release:        ENTRY PROCEDURE (p:pool, d: descriptor)
        begin
                P(inner);
                p.free := p.free + < size of segment referenced by d >
                < release space associated with d >
                V(inner);
        end;
        ...

end;
```

Figure 4.3 Levin's Storage Allocator

tor, Levin achieves mutual exclusion and condition synchronization using the two semaphores, **inner** and **outer**. The function of these semaphores is explained in detail by Levin, and also

by Black in [Black 82]. The author proposes a new approach which is implementable in existing systems, both monitor and message-based, by following the design techniques specified in the model. Program paradigms are presented for these implementations in Mesa, a language supporting monitors, and in Verex, a message-based operating system. The storage allocator is very simple. **Allocate** accepts a **pool** and **size** as parameters, obtains sufficient storage from the pool and returns a reference to it. If adequate storage is not available to do this, the POOL-LOW exception is generated. It is raised in turn with the users of the pool until either adequate resources become available, or there are no more handlers. If the handlers do not succeed in releasing enough store the POOL-EMPTY exception is generated. The **Release** operation is straightforward.

A pool user has the form described in Figure 4.4 below. The pool user is prepared to handle POOL-LOW at any instant. Except during the time when segments are being added to **m**, POOL-LOW is handled by the **squeeze** procedure. But during critical data-structure updates to **m**, invocation of **squeeze** is inhibited by a dummy (masking) handler for POOL-LOW that does nothing; in this situation, the POOL-LOW exception is lost.

Levin notes the differences in synchronization and communication requirements among the structure class exceptions POOL-SCARCE, POOL-LOW and POOL-EMPTY and their handlers, and proposes a new mechanism to solve the allocator problem. He defines a *selection policy* for each exception, and describes three such policies: *broadcast-and-wait*, *sequential-conditional* and *broadcast*.

Under the *broadcast-and-wait* policy, all eligible handlers are invoked in parallel. When all handlers thus initiated have completed, execution resumes immediately following the **raise** (like Mesa's SIGNAL) statement. Thus, while all eligible handlers execute in parallel (concep-

```
Userpool : MODULE =
begin
        p               : pool;
        m               : hairy-list-structure;
        exception       : NOSOAP  policy broadcast;
        raises NOSOAP on somefunc;

        somefunc:       PROCEDURE (<info>)
        begin
                d1, d2 : descriptor;

                d1 := Allocate (p,amt1)
                        [ POOL-EMPTY: raise NOSOAP; return];
                d2 := Allocate (p,amt2)
                        [ POOL-EMPTY: Release (d1); raise NOSOAP; return];
                < fill in  d1 and d2 >
                begin
                        add-to(m,d1);
                        add-to(m,d2);
                end [POOL-LOW: ]
        end;

        squeeze:        PROCEDURE ()
        begin
                < perform data-dependent compaction of m using
                 Release(p,d) to release any elements d removed
                 by compacting m >
        end;
end [POOL-LOW: squeeze() ]
```

Figure 4.4 A User of Levin's Storage Allocator

tually at least), the signaller is suspended until they have *all* completed. We will call this a **Reliable Broadcast Send**. The server, or the operating system supporting such servers, needs a list of all its clients as it must delay until all the handlers are done.

In his example, Levin does not use this policy for raising the POOL-LOW exception -- instead he uses the *sequential-conditional* policy. This policy has a predicate associated with the **raise** statement. The predicate is evaluated, and, if TRUE, execution continues following

the **raise** statement. If the predicate is FALSE, an eligible handler is selected and initiated. When it completes, the predicate is again evaluated.[2] The **raise** statement terminates when either the predicate becomes TRUE, or the set of eligible handlers is exhausted. Thus in his example, the exception is raised only as many times as needed to satisfy the current request -- it is therefore more efficient for any single request. However, in many situations, if the pool is low for one request, there is a high probability that the next request will also find the pool low. It would seem more advantageous to use the *broadcast-and-wait* policy, so that all the clients will squeeze their data. This may save subsequent shortage, in the same way that *page-look-ahead* attempts to reduce page faults by anticipating subsequent requests. But the server then has to wait for *all* the handlers to complete before it can continue; this can delay the current request unnecessarily if say, the first handler to complete released sufficient resources for satisfying the current request.

Levin also proposes a variation of *broadcast-and-wait* simply called *broadcast* which relaxes the completion requirement. All eligible handlers are initiated in parallel but the signaller does not wait for any of them to complete. It merely continues execution following the **raise** statement. Thus all handlers and the signaller execute in parallel. Levin's formal proof rule for this policy requires that the signaller and handlers act nearly independently. These are exactly the semantics he intended -- the broadcast policy should be used normally to report to users of an abstraction a *purely informational condition.* (He notes that you can relax the proof rules somewhat, but then we risk serious interactions with parallelism and synchronization mechanisms.) This selection policy could be used when the storage falls below a certain threshold; the server sends a POOL-SCARCE exception to a broadcast communica-

[2]making sure each time a different handler is selected

tion channel to which all clients subscribe. On receipt of the notification, each client takes appropriate action to release excess storage.

This selection policy, unlike the others, does not require that the server keep a list of client handlers, as the signaller acts independently of the handlers. If the signaller broadcasts to all subscribers, and some do not receive the message, the signaller is not directly affected. Thus we can use a so-called **Unreliable Broadcast** to implement Levin's *broadcast* policy, and the interesting feature is that it can be implemented *without a centralised list of subscribers at the signaller.* Instead, in a distributed system, all the system needs is to guarantee to try to deliver the message to all hosts in the system. Hence the server does not have the burden of being responsible for maintaining a list of clients' handlers, nor is concerned with garbage collection of clients which are no longer interested in the exception.

As several local area networks provide an efficient broadcast feature, we would like to exploit it for such broadcast exceptions. An unreliable broadcast has been implemented by Cheriton for the V-kernel [Cheriton 84], and a multicast feature has recently been implemented in UNIX 4.2BSD [Ahamad 85]. But to use such an unreliable broadcast in this problem, first we need to show that we can redesign Levin's storage allocator to exploit a *broadcast* signal, instead of the *broadcast-and-wait* signal which he used. We then discuss how we could use the Medusa, Mesa and V-kernel operating systems as virtual machines to provide tools to support this high level language construct.

4.2.2. Storage Allocator Using Unreliable Broadcast Notification

Black discusses various alternatives to Levin's storage allocator in his thesis (pp 120-129), and proposes a solution in CSP [Hoare 78] which uses a reliable broadcast from the server to a list of its clients. His solution *persists,* in that if an **Allocate** request cannot be

satisfied immediately it continues to signal the POOL-LOW exception to all the user processes until enough store is available. During this time no further **Allocate** requests are accepted, but **Release** requests will be handled correctly. We consider this an undesirable solution; a timeout is necessary so that the client will not be suspended for arbitrarily long periods should no client be ready to execute *Release*.

We propose using monitors, condition variables, and their associated WAIT and SIGNAL operators which were designed [Hoare 74] for handling synchronization and data access in such situations to implement the allocator. Now Levin and Black dismiss the notion of using a monitor for achieving the necessary features because of deadlock. If a call of **Allocate** generates a POOL-LOW exception and a client handler attempts to call **Release,** the handler will be suspended awaiting completion of the **Allocate** request. However, **Allocate** cannot complete until at least one of the handlers completes. This problem arises specifically because of the semantics of Levin's *sequential-conditional* policy, whereby a signaller can not continue until an eligible handler has *completed.* This problem of managing resources in monitors was already recognized, and has been described by Lampson and Redell in their implementation of PILOT [Lampson 80] in Mesa. They resolved it by use of a WAIT on a condition variable which releases the monitor lock.

Our solution requires that the POOL-LOW exception obeys the *broadcast* selection policy, because then a monitor can be used for managing the storage pool. This solution is portrayed in Figure 4.5, where a Mesa-like condition variable **MoreAvailable** with an associated timeout is used[3].

[3]Note that the semaphore **inner** is not needed here, as the monitor protects the critical data from concurrent access.

```
StorageAllocator: MONITOR =
begin
        p               : pool;
        free            : integer;
        exception       : POOL-LOW      policy broadcast;
        exception       : POOL-EMPTY    policy broadcast;
        condition variable : MoreAvailable, Squeezedone;
        Boolean         : REQUEST-PENDING = FALSE;
        ...
        Allocate:       ENTRY PROCEDURE (p:pool, size: integer)
        returns (d:descriptor) =
        begin
                if REQUEST-PENDING then WAIT Squeezedone;  | P(outer)
                if p.free < size  then
                begin
                        REQUEST-PENDING = TRUE;
                        raise POOL-LOW;
                        while p.free<size do
                        begin
                                WAIT MoreAvailable;
                                if <timeout on MoreAvailable fired> then
                                begin
                                        raise POOL-EMPTY;
                                        REQUEST-PENDING = FALSE;
                                        NOTIFY Squeezedone;     | V(outer)
                                        return;
                                end;
                        end;
                end;
                p.free := p.free - size;
                < create descriptor d for segment of amount size >
                REQUEST-PENDING = FALSE;
                NOTIFY Squeezedone;                             | V(outer)
        end;

        Release:        ENTRY PROCEDURE (p:pool, d: descriptor)
        begin
                p.free := p.free + < size of segment referenced by d >
                < release space associated with d >
                NOTIFY MoreAvailable;
        end;
        ...
end;
```

Figure 4.5 Storage Allocator Using a Monitor and Broadcast Exceptions

In our program paradigm, the **Allocate** procedure raises the POOL-LOW exception signal to all its clients when storage is inadequate, and then executes a WAIT on the **MoreAvailable** condition variable, with an appropriate timeout. The WAIT releases the monitor lock, thus enabling handlers to execute **Release.** Whenever some store is released to the pool, a NOTIFY signal on the **MoreAvailable** condition is executed. The **Allocate** procedure can then try again to satisfy the request. If the timeout fires[4] the procedure **Allocate** can raise the POOL-EMPTY exception and return as before. **Allocate** requests made during squeezing are delayed on the condition variable **Sqeezedone,** which acts like the semaphore **outer** in Levin's model.

Now that we have designed a storage allocator monitor which uses a notification broadcast to all its clients, we develop program models for its implementation in various languages and systems.

4.2.3. Storage Allocator in Mesa

The allocator example as it stands is almost exactly implementable in Mesa, except for the problem of raising the broadcast exception POOL-LOW. We have already seen one solution in the Mesa cooperating files example, where each client provides a procedure parameter to the server. The server invokes each client's procedure in turn until enough storage has been released. Unfortunately this requires the server to maintain a list of clients and their call-back procedures.

The alternative in our program paradigm is for each client to WAIT on a new condition variable **PoolLow** which is signalled by the server when the exception is detected with a

[4]remember we do not have a list of clients, so we cannot tell when ALL the handlers have completed

NOTIFY BROADCAST **PoolLow.** Unfortunately the clients would be unable to handle any other business while they were being blocked on the **PoolLow** condition variable. This could be resolved by each client providing a peer exception handler process which waits on the POOL-LOW condition variable and which executes *SQUEEZE* synchronously whenever the condition is signalled. This solution also has the effect of increasing concurrency.

The implementation of condition variables in Mesa still involves a queue of waiting processes to be held in the monitor, but this list is maintained by the operating system rather than by the monitor's programmer.

Software tools for efficient implementation of this system are required for the cooperating processes, so that when a client process dies, its peer exception handler dies also, and is (eventually) removed from the monitor's queue. This process death notification should be made only when (or if) the condition variable was notified, otherwise the monitor would receive arbitrary unwanted interrupts from the operating system whenever a client process died. Furthermore, condition variables as specified by Hoare would not be adequate, as the Mesa condition variables are implemented with timeouts and the signal on a condition variable signifies a notification hint -- this implementation of monitors is required for correct operation. It is not clear how efficient such a mechanism could be in Mesa, particularly when some clients are executing on remote machines. Presumably the overheads are quite high, as the implementors of the **PleaseRelease** cooperating files did not use this approach. However, the machinery is present and could be used.

There is just one problem with this scheme for synchronous notification, and that is in ensuring the mutual exclusion of access to shared data of the client and the exception process while it is squeezing the data. Atomic actions, or shared memory and guaranteed progress of

a process until it voluntarily relinquishes the cpu (such as for Thoth team-members) would be sufficient to provide mutual exclusion; various designs are discussed later in Section 4.3.

4.2.4. Storage Allocator in Medusa

Medusa defines *external condition reports* which are based on Levin's proposed structure class exceptions, which can be raised on shared objects. An external report is raised by any activity with access to a shared object which incurs an *object exception.* For each shared object in the system there are 8 classes of object exception, some of which are signalled by the utilities (e.g. parity failure may occur on memory pages), and others may be defined and signalled by user activities. When an object exception is signalled, all other activities with access to the object receive an external report of the exception. These external reports are made using *flagbox* objects which are managed by Medusa's exception reporter utility. Each activity can read the flags synchronously in the flagbox when it wishes, or it can request to be interrupted when a particular flagbox becomes non-empty. In this latter mode, an external report is similar to a UNIX *software signal* interrupt.

The major disadvantage of the Medusa external mechanism is that backpointers are needed from every shared object to its users. As we have already shown, such tight coupling is neither necessary nor desirable; in many applications an unreliable broadcast notification from the server to its (unknown) clients is sufficient.

4.2.5. Storage Allocator in V-kernel

In Thoth-like message-passing environments, an extension of the server-client notification schemes already proposed in Section 2.3.1 requires each user registering an exception-process to be notified. The server is responsible for maintaining such a list, and for garbage-collection

of clients who are no longer interested in the exception. This solution suffers from the same disadvantage as Mesa's and Medusa's, that back-pointers are needed from the server to its users. Our new solution uses a multicast inter-process communication primitive, which we call a **Broadcast** (or Multicast) **Send (BSend)** [Cheriton 84] This has been implemented in the V-kernel [Cheriton 83c] and in UNIX 4.2BSD [Ahamad 85]. A server can inform clients of a *group-id* on which exceptions can be sent. The server uses the non-blocking **BSend** to the group-id to notify the exception, and clients can chose to receive from that group-id either with a separate exception-process, or by their main process (if the client is itself a server waiting on a **Receive**).

A program model for a storage allocator server in the V-kernel is shown in Figure 4.6.

```
pool-server()
begin
  integer     id, group-id, CLIENT_ID, AMT, reply, THRESHOLD = 100;
  Boolean     REQUEST-PENDING = FALSE;
  message     msg[MSG-SIZE];
  pool        p;
      ...
  repeat
  begin
      id := Receive(msg);

      select (msg.REQUEST-CODE)
      begin
            case Allocate:
            begin
                  if REQUEST-PENDING then
                        begin
                              queue-request(id,msg);      |
                              reply := NO-REPLY;          | P(outer)
                              continue;                   |
                        end;
      Alloc:        if p.free < msg.SIZE then
                        begin
                              REQUEST-PENDING := TRUE;
                              BSend (< message POOL-LOW>, group-id);
                              CLIENT-ID := id;
                              AMT := msg.SIZE;        | Implements WAIT, as
```

```
                        reply := NO-REPLY;      | no reply to client
                        < start timer >
                        continue;
                    end;
                p.free := p.free - msg.SIZE;
                satisfy-allocate (id, msg.size);    |makes reply to id
                reply := NO-REPLY;                  | no need to reply
            end;
            case Release:
            begin
                < update p.free>;
                if REQUEST-PENDING then
                begin                               | Implements NOTIFY
                    if p.free >= AMT then
                    begin
                        satisfy-allocate(CLIENT-ID,AMT);
                        REQUEST-PENDING := FALSE;
                    end;
                end;
                reply := OK;                        | to releaser
            end;
            case Timeout:
            begin
                if REQUEST-PENDING then
                begin
                    Reply ( <message POOL-EMPTY>, CLIENT-ID );
                    REQUEST-PENDING := FALSE;
                end;
                reply := NO-REPLY;                  | to timer
            end;
        . . .
        end;
        if reply ≠ NO-REPLY then
         begin
            msg.REPLY-CODE := reply;
            Reply (msg, id);
         end;
        if (( ¬ REQUEST-PENDING) ∧ (requestq ≠ NULL)) then
         begin
          dequeue-request(id,msg);                  |
          goto Alloc;                               | V(outer)
         end
    end;
end;
```

Figure 4.6 Pool-server in V-kernel

The server uses **BSend** to notify any interested clients of the POOL-LOW exception. Clients obtain access to groupid (on which the exception is broadcast) through some initial request to the server, not shown here. The procedure **satisfy-allocate(id,size)** creates a descriptor for the requested segment of *size* resource units, and then executes a **Reply** to the process *id*. The procedures **Queue-request(id,msg)** and **Dequeue-request(id,msg)** save and restore any **Allocate** requests which are received during a *REQUEST-PENDING*. Thus the queue of waiting **Allocate** requests is explicitly maintained by the server, instead of by the monitor's condition variables. One advantage of this approach is that the server can make arbitrary scheduling decisions easily, rather than be constrained by the mechanisms provided by the monitor implementation. Another advantage is that the implementation of WAIT and BROADCAST NOTIFY are explicit here; there are no hidden overheads in operating system implementation of the condition synchronisation primitives. However, this event-driven server is longer than the monitor version in Figure 4.4 as it requires extra variables (REQUEST-PENDING and the queue-variables) to encode its state and manage its flow control. But this example provides a useful program paradigm for message-based systems which do not provide a monitor construct or separate ports for different types of server requests.

A corresponding pool user is shown in Figure 4.7.

An alternative approach is to assume that a new **Allocate** request occurs only infrequently when the server is in the state *REQUEST-PENDING*. The server can make an exceptional *BUSY* reply to the client instead of queueing the request explicitly, and the client must take appropriate action such as delaying before retrying the request. This server does not implement the same scheduling semantics, but by designing the client to maintain the extra state information *RETRYING*, the server code is simplified. This is the approach advo-

```
userpool()
begin
  extrn             hairy-list-structure m;
  extrn             semaphore          M-UPDATING;
  integer           id, pool-id, reply;
    ...
  repeat
  begin
       id := Receive(msg);

       select (msg.REQUEST-CODE)
       begin
            ...
            case somefunc:
            begin
                 descriptor d1,d2;
                 Send(<Allocate-msg>, pool-id);
                 if msg.REPLY = POOL-EMPTY then
                           begin reply := NOSOAP;        | to caller
                                 continue;
                           end;
                 d1 := msg.PTR;
                 Send(<Allocate-msg>, pool-id);
                 if msg.REPLY = POOL-EMPTY then
                           begin Send(<Release-msg d1>, pool-id);
                                 reply := NOSOAP;
                                 continue;
                           end;
                 d2 := msg.PTR;
                 < fill in d1 and d2 from pool-id's message >

                 P(M-UPDATING);
                 add-to(m,d1);
                 add_to(m,d1);
                 V(M-UPDATING);
                 reply := OK;                            | to caller
            end;
       if reply = NO-REPLY continue;
       msg.REPLY-CODE := reply;
       Reply (msg, id);
  end;
end;

exception-handler()
begin
```

```
extrn           hairy-list-structure      m;
extrn           semaphore          M-UPDATING;
repeat
begin
 ...
    id := Receive(msg);
    select (msg.REQUEST-CODE)
    begin
        case POOL-LOW:
        case POOL-SCARCE:
        begin
            P(M-UPDATING);
            <squeeze by performing data-dependent compaction of m
            using Send(<Release d message>, pool-id)
            to release any elements d removed by compacting m>
            V(M-UPDATING);
        end;
    end;
 end;
end;
```

Figure 4.7 A User and Exception Handler of the V-kernel Storage Allocator

cated in the general model for designing event-driven servers which maintain minimum state, and for the message-based systems like Thoth this approach considerably simplifies the server code.

4.3. Exception Handlers as Processes: Mechanisms for Mutual Exclusion

If systems are designed so that exceptions are handled by a separate process, the problem is encountered in synchronization of the exception process and the other processes involved. This is particularly acute if the exception process executes *non-atomically,* where a process is defined to execute *atomically* if it completes in its entirety or not at all, with no visible intermediate states.

For an example, consider the V-kernel storage allocator described previously. Mutual exclusion is necessary on the data structure **m;** while USER is updating it, the exception handler must not be **squeezing** it. There are several alternatives for inhibiting **squeeze**

while **m** is being updated:

(1) reduce concurrency by allowing only the one process, USER, to update **m**

(2) use a simple P-V semaphore

(3) hardware test-and-set a shared variable

(4) temporarily remove **squeeze** from the server's list of handlers while USER is updating **m**

(5) temporarily mask the action of squeeze in EP while USER is updating **m**

(6) rely on non-preemptive process execution -- while the exception process executes, USER cannot obtain read/write access to **m**

(7) define **m** to be atomic data which can only be updated or accessed atomically, by so-called *atomic transactions*.

These solutions are considered in turn below.

(1) First, mutual exclusion could be ensured by making the USER process receive the exception notification from the server, and execute **squeeze** only when it is free to do so. The server would **BSend** the POOL-LOW exception, which USER would **Receive** and act on synchronously. This could be an effective solution when the USER acts as a server itself (and thus is usually ready and waiting on a **Receive**) and when there is no need to gain concurrency by having a separate exception process. To receive the broadcast exception message, the client USER would subscribe to the server's *group-id*. This would complete a **SendReceiveReply** loop between USER and the server. Although this is considered undesirable for regular **Sends**, a broadcast Send in the V-kernel where the server does not expect any **Reply** has been implemented so it is safe to complete such a loop. Reasoning about deadlock in such systems is thus the same as if the broadcast

Send were like a **Reply;** the difference is that the client can pick up this kind of **Reply** with a regular **Receive** statement.

The remaining solutions all provide a greater degree of parallelism, whereby the process executing **squeeze** (EP) is distinct from the process updating **m** (USER).

(2) In Figure 4.7 the P-V semaphore M-UPDATING was used to force mutual exclusion between USER and EP. This technique has the overhead of two extra process switches for each P-V operation. Ideally a solution is required which involves no extra process switches for a normal-case data update (when no other process is accessing the critical data).

(3) Use a shared variable, M-UPDATING, initialised to *FALSE,* which may be used in an atomic action such as provided by a hardware test-and-set. The USER and EP processes test this before accessing the critical data thus:

```
while M-UPDATING do delay(1);
M-UPDATING := TRUE;
        critical code
M-UPDATING := FALSE;
```

This has little overhead in the normal-case -- just two statements in USER and EP to test and reset the boolean flag. In our solution, if an access conflict is found, the second process loops in a **delay** statement until the resource is freed by the other process. This solution has the obvious drawback that it is not crash-resilient, nor is it fair. In the exceptional case of a data access conflict between USER and EP, the simple solution imposes some extra delay on USER and EP, although this may well be balanced by the lack of process switching. This solution therefore follows the principle of minimising the normal-case overhead (by having no context switches), at the expense of some overhead in extra delay and cpu cycles on handling the exceptional case of data-access conflict.

However, hardware support is required for the indivisible test-and-set operation.

(4) This solution can only be used if a list of clients is held at the server, and a **Reliable Broadcast** is used to notify the clients. This solution imposes an overhead of 4 process switches on each normal-case data update -- two to notify the server of the change in the list of eligible exception handlers, and two more to reset the list -- plus the maintenance of the list.

(5) Masking the effect of squeeze in EP when USER is updating the data-structures, again requires 4 process switches. First USER executes a **Send** to EP to set the mask, and then after the critical update, the USER has to **Send** again to EP to reset the mask. This solution has the advantage over the previous method, that the server is independent of its users, so it can be used with the unreliable broadcast.

(6) Exploiting non-preemption between processes on a team to achieve mutual exclusion is available in Thoth-like operating systems, and has been used successfully for many applications. The idea is that a process cannot be interrupted by any other process *on the same team*; thus if a page fault occurs, only a process on another team can execute, not another process on the same team. This contrasts with the Medusa activities in a task force, which are designed explicitly to execute in parallel. However, when the actions necessary for data updates are complex, involving procedure calls, there are no means of knowing, or specifying, which process actions are atomic (i.e. do not yield the processor by making system calls) and which ones are non-atomic (i.e. do yield the processor, leaving an intermediate state of computation visible to another process). Hence a procedure which happens to yield the processor must not be called. In large systems where modularity is essential, it is not known which procedures yield the processor; thus such restric-

tions are intolerable and unmaintainable.

(7) Much research has been done on *atomic data updates* -- all-or-nothing computations [Lomet 77], [Moss 82]. Liskov and Scheifler described a high level language approach in their paper *Guardians and Actions: Linguistic support for Robust, Distributed Programs* [Liskov 83]. They promise some high level language tools for specifying atomic actions in their integrated programming language and system called ARGUS. Their approach has centered on a class of applications concerned with the manipulation and preservation of long-lived, on-line data such as airline reservations. In ARGUS, the user can define *atomic data* which is accessed by processes which are called *actions* (or, equivalently, **transactions).** The action may complete either by *committing* or *aborting.* When an action aborts, the effect is as if the action had never begun: all modified atomic objects are restored to their previous state. When an action commits, all modified objects take on their new states.

Let us see how such ideas could be used in the storage allocator. USER and EP share access to *atomic data* **m.** If EP obtains a write permission on the data, and then blocks (maybe for some debugging i/o) and then USER tries to execute, USER will be prevented from accessing the atomic data, as EP has a write lock to it. Ideally, USER will be delayed until EP has finished -- just like waiting on a P-V semaphore. But in this case, the programmer merely has to specify that the data **m** is *atomic;* the implementing system then takes care of the queue on the semaphore. This linguistic tool presents a real advance in concurrent high level language design, and should certainly be incorporated into new languages. Its implementation in other operating systems has not been studied as the ARGUS project is an integrated system.

The author has designed an implementation of ARGUS's transactions under the UNIX operating system, exploiting the principles of structuring programs for normal-case efficiency, which is described in full in the next Chapter. This provides a tool for implementing exception processes, with due regard for process synchronization and serializability.

To conclude, for efficient implementation of exception handlers as processes, efficient tools for mutual exclusion are needed such as provided by hardware test-and-set, or by software atomic actions.

CHAPTER 5

Programs Illustrating the Model

This chapter describes two programs which illustrate the use of the model.

First, two structures used for exception handling in multi-process implementations of the X.25 and X.29 protocols in Verex [Lockhart 79] are described. Both these protocols are hierarchic and layered. The X.25 protocol [CCITT 76] is a multi-layer protocol for interfacing computer equipment to a packet-switched network. Each X.25 protocol layer -- the physical, link level or packet level -- can be described as a finite state machine (FSM) event manager making appropriate state transitions on receipt of packets or timeouts [Bochmann 78]. The CCITT Virtual Terminal Protocol X.29 [CCG 78], here referred to as Datapac's Interactive Terminal Interface (ITI) protocol, establishes a data structure that is viewed as representing the terminal. A remote terminal user may therefore interact with a host-side application program over the Datapac network via the host-side X.25 and ITI protocols. These protocols were studied because although each protocol layer is implemented as a collection of cooperating processes, the protocols are structured to reflect differences in the type of exceptional situations that may occur. The author implemented the ITI protocol on the Verex operating system [Atkins 83b], by following the design principles established in the model.

Deering's implementation of the X.25 protocol [Deering 82] is structured to reflect the logical flow of events. Within each of the *lower* protocol layers of X.25, one process takes the form of a Finite State Machine server. This process provides all the necessary synchronisation of transmitting and receiving, leading to efficient handling of all transitions. Thus in these lower level protocols, where it is difficult to identify *normal* transitions because the

probabilities of the events are not known, there are no special exception cases; all events are treated in a similar way. The resulting software is produced quickly and models closely the protocol descriptions contributing to its understandability and maintainability.

For *higher* level protocols such as the X.29 protocol, exception events can be identified which occur with low probability. The author has structured the ITI protocol so that transmitting and receiving are performed by two independent filter processes, and exception handling is performed by an auxiliary exception process which is invoked as needed [Atkins 83b].

This implementation of ITI illustrates designing a program to achieve all three objectives of the model (see Figure 3.3).

(1) Objective A: decreasing the average run-time, is achieved by reducing the normal case handling time (cost b of Level 2 of Figure 3.3). An alternate process configuration is used to reduce inter-process communication for the normal events (read/write requests). This is shown as feature 2 of Level 3 of Figure 3.4.

(2) Objective B: decreasing the exception processing time, is achieved for the <BRK> exception event, by reducing the inter-process exception notification time. This is shown as cost b of Figure 3.5.

(3) Objective C: increasing the program's functionality in a modular way, is achieved by structuring the program to use an ignorable exception notification from the server (X.25) to the client (ITI), for network reconnection. This design method is shown in Figure 3.6.

This implementation of ITI as a filter is compared with the multi-process server approach used for X.25, and it is shown that the filter structure matches the functionality of the ITI protocol, and results in efficient software.

In Section 5.2, a mechanism for implementing nested atomic transactions to perform atomic data base updates is described. The need for such a mechanism has already been discussed in the previous Chapter in Section 4.3 concerning mutual exclusion mechanisms. The author's implementation illustrates objective A of the model, by reducing the average run-time, shown in Figure 3.4. This is achieved by reducing both cost b -- the normal case handling cost -- and cost c -- the context-saving cost -- of level 2 of the model. To reduce the latter context-saving cost incurred while handling normal events, the approach is to assume that transactions COMMIT by default, and therefore to assume that an ABORT is an exceptional event. Context-saving is thus kept to a minimum while processing normal events, but at the added expense of exception handling when an exception occurs.

Another technique to reduce the normal case handling costs, is to use broadcast exceptions to reduce the network traffic over a Local Area Network (feature 3 of Level 3 of the model in Figure 3.4). A hybrid scheme of reliable-unreliable broadcast inter-process communication is used to achieve this. This design gives efficient performance and is easy to program.

5.1. Exceptions and Protocol Implementations -- a New Approach

5.1.1. The ITI Protocol

ITI acts mainly as an i/o filter, transmitting read/write requests from its client (called the application, APPL) through to X.25. Now if it is structured as a finite state machine server like the X.25 server, then the ITIMain process requires two worker processes, a reader (ITIReader) and a writer (ITIWriter), to handle delays in reading from and writing to X25Main, as shown in Figure 5.1.

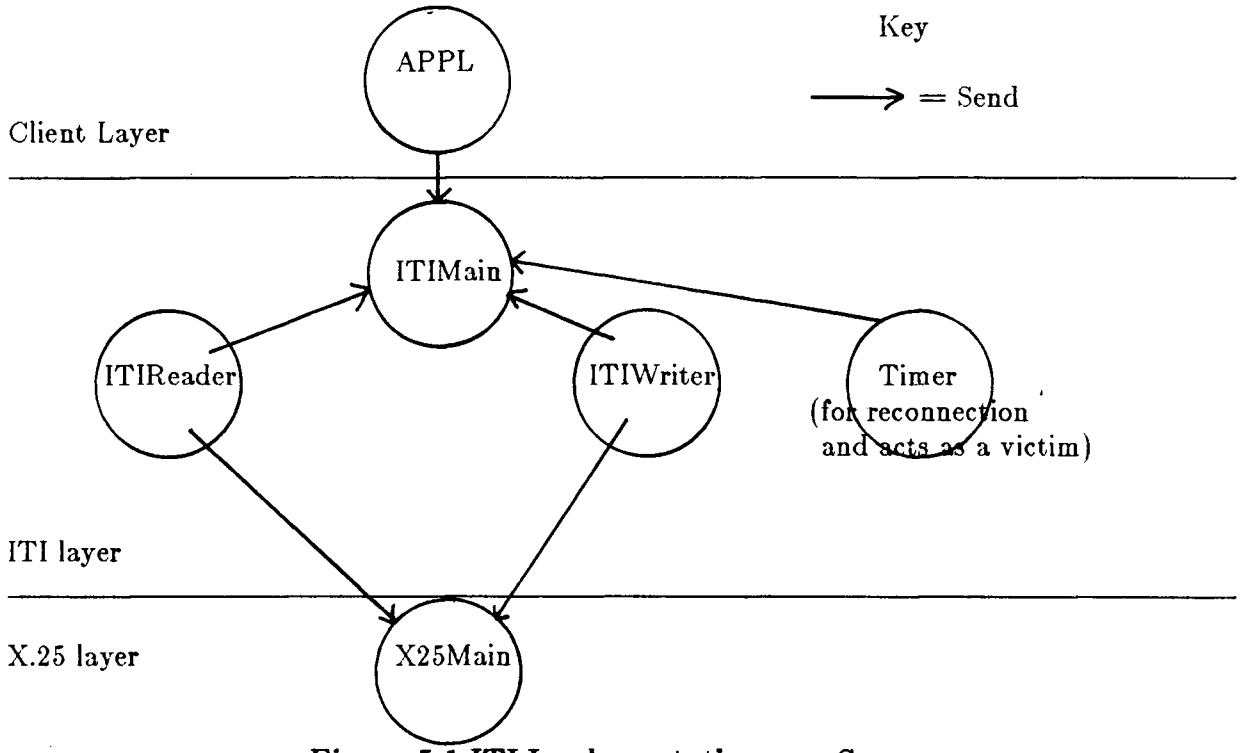


Figure 5.1 ITI Implementation as a Server

The author has analysed such an implementation, and finds that each read and write request from the client involves 6 more process switches at the ITI layer -- from client to ITIMain to ITIReader to X25Main and back again. Thus for a normal case read from the network, there is a total of 7 **Send-Receive-Reply** messages (including the processing within X.25), or 14 context switches. On the TI990 computer, each **Send-Receive-Reply** sequence takes 3 msecs (though on faster machines with hardware support for context switching it takes about 1 msec). Therefore it takes at least 21 msecs to process the context switches for each read request. At full speed on a 9600-baud Datapac link, 1200 bytes/sec are received, in data packets of 128 or 256 bytes. Thus up to 10 packets must be processed per second; the context switches for this take 210 msecs. If the machine is dedicated to the task, it can easily keep up with the data arriving. Then the argument for readability and understandability of multiprocess structuring compensates for the context switching overhead. However with three

X.25 virtual circuits going full speed, or with one other user on the system the computer could hardly keep up.

Following the model, it was noted that a large number of context switches were being made for normal-case processing, so the author decided to restructure the ITI protocol to reflect its major function -- that of an i/o filter (rather than a general-purpose server). By using a different process configuration, normal case read/write requests can be handled more efficiently. But some consideration must also to be given to handling the exceptional situations which might occur, such as network failing or a user hitting <BRK>. The X.25-ITI interface must be designed so that inter-process exceptions can be communicated efficiently and conveniently.

First, an analysis of the behaviour of the protocol on the <BRK> exception was made.

5.1.2. The Interrupt Protocol

Since their inception, communication protocols have employed asynchronous interrupt signals, but typically require that interrupts be synchronised with the parallel data channel. The X.25 protocol specifies a complex sequence of data transfers over the network to handle the interrupt exception signal <BRK>, which occurs when a remote user presses the interrupt key on his terminal. A total of 5 packets must be exchanged across the network through the link layer of X.25 on a <BRK> signal. These are shown below in Figure 5.2.

To transmit an interrupt caused by a remote user pressing <BRK>, an *interrupt expedited data* (INT) packet is sent by the remote X.25's client, CLREMOTE, which must eventually be acknowledged by an *interrupt confirm* (INT-CONF) packet from the local X.25's client, CLOCAL. Subsequent interrupts must be refused by CLREMOTE until the acknowledgement arrives. The INT packet may overtake ordinary data packets in transit.

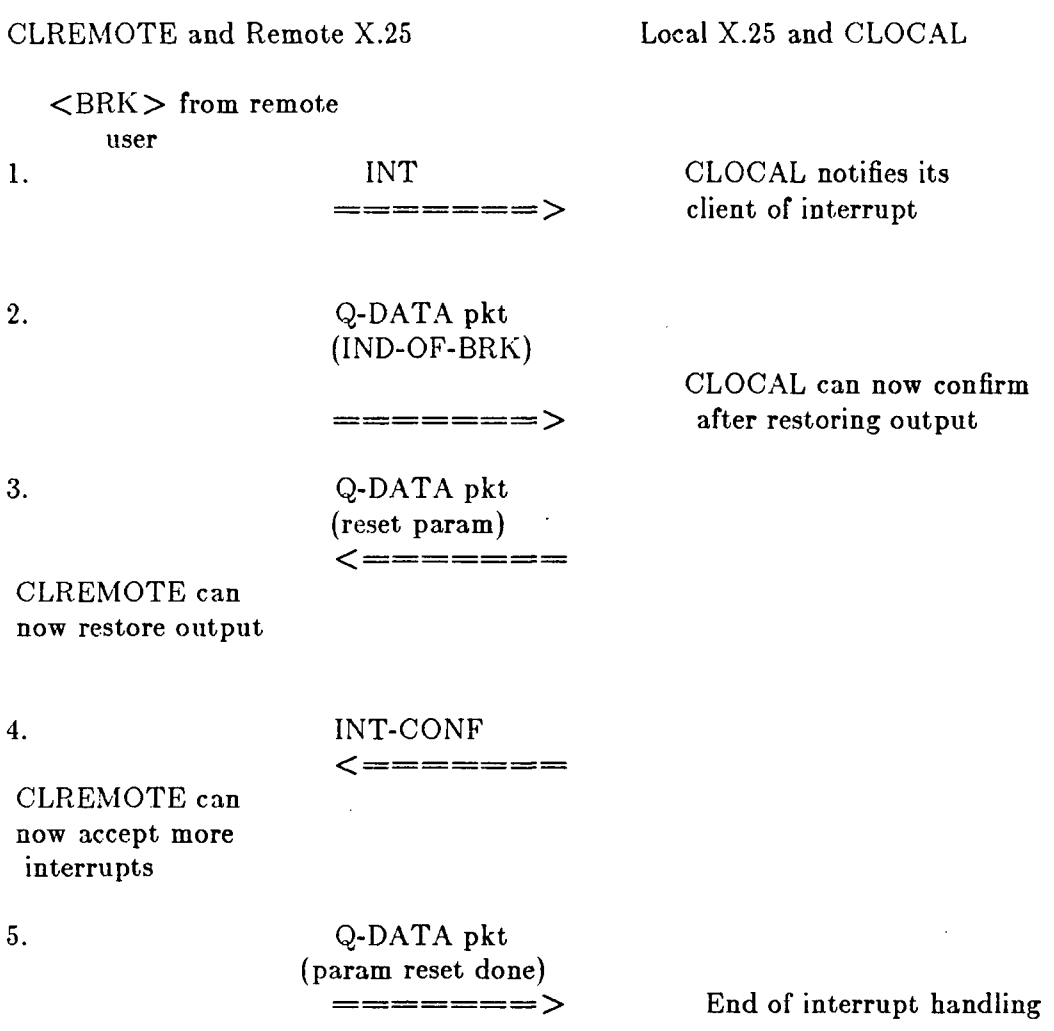


Figure 5.2 Network Packets to be Exchanged on an Interrupt

Thus CLREMOTE also inserts a *Q-DATA* packet with the code INDICATION-OF-BREAK (IND-OF-BRK) into the data stream at the point of the <BRK>. CLREMOTE also has an option to discard all subsequent output, which it may exert after a <BRK>.

When CLOCAL receives the INT packet from X25Main, it takes appropriate action to notify its client that an interrupt has occurred (possibly by destroying the client). Then

CLOCAL must read data packets until the IND-OF-BRK packet. Commonly, packets passed-over in flight are discarded at the receiver. Of course, if the remote host was outputting, or was quiescent at the time of the <BRK>, there will be only the IND-OF-BRK packet to be read.

When CLOCAL receives the IND-OF-BRK, it must send a *Q-DATA* packet back with a parameter set to a code to tell CLREMOTE to stop discarding output (if it was), and then the INT-CONF packet can be sent. As the last stage in handling an interrupt, CLREMOTE sends the *Q-DATA* packet back to CLOCAL, with the parameters reset to indicate that output has been resumed.

5.1.3. The ITI Implementation in a Thoth-like Operating System

We observe that although the network can fail at any time, it is not necessary for the higher levels to be informed of this event immediately (i.e. asynchronously). Instead, the application may continue to execute normally, unaware of any disruption to the i/o service until a read or write operation to/from the network is requested. When this occurs, the ITI reader process may receive, synchronously, the exceptional information concerning the failure of the underlying network layer.

This synchronous scheme cannot however be used to communicate a <BRK> interrupt from the remote user, as notification cannot always wait until the user requests an i/o operation. For example, if a user initiates a long compilation, then notices an error and presses <BRK>, a response is required within a short finite period, say half a second. In such a situation, the client is *deaf* to the server; thus another solution must be found.

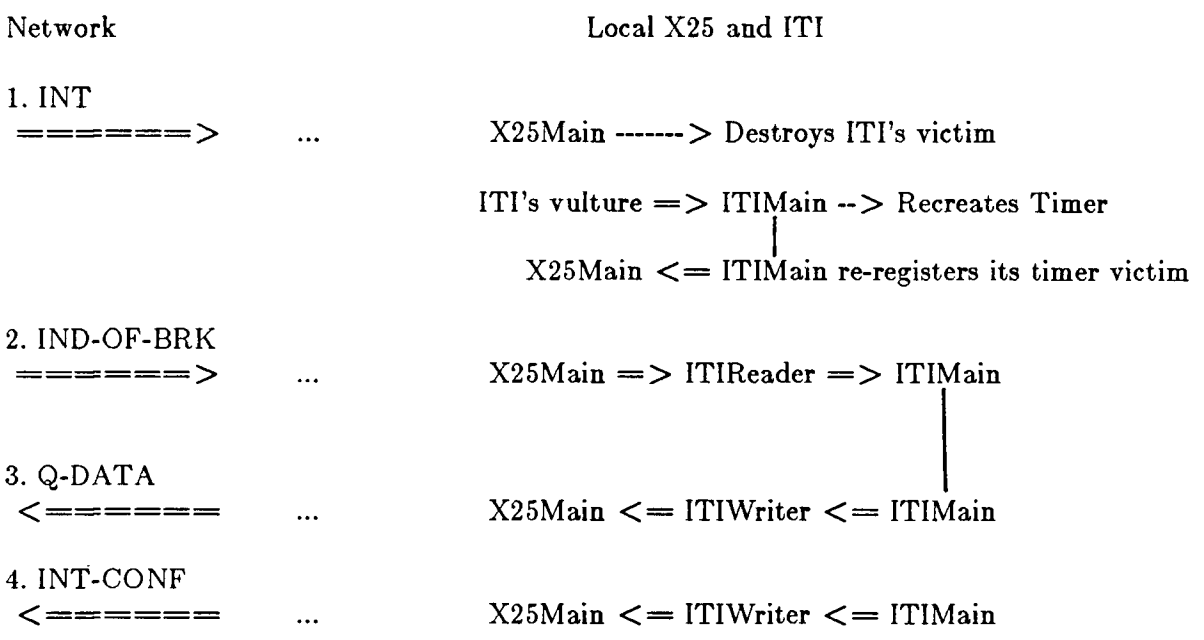
We have already discussed tools for such inter-process exception notification such as process destruction (Section 2.3.2.3) and the toggle switch (Section 4.1.1). In order to chose

between these alternatives, an analysis of their behaviour on the <BRK> exception was made.

5.1.3.1. Notification of Inter-process Exception Using Process Destruction

The process switches involved in processing an interrupt using process destruction are shown below in Figure 5.3[1]. Each inter-process **SendReceiveReply** message exchange causes

---



where ... = additional process switches within X.25 to handle each packet

Figure 5.3 Process Switches to Handle an Interrupt Using Process Destruction

---

[1]assuming that the ITI protocol is structured as a server with two workers to handle delays in communication with X.25.

2 process switches. The number of **SendReceiveReply** messages to handle each packet are given in column 1 of Table 5.1 below.

Thus a total of 20 + nX.25p[2] = 33 **SendReceiveReply** message exchanges are needed to handle the <BRK> exception, which takes at least 99 msecs. It takes even longer for ITI's client to make its response (such as a message <IO-BRK>). When this system was implemented, timesharing with another user we observed response times of between 1-4 seconds for the <BRK> to be handled. This poor performance arose because the other user could obtain a full cpu quantum between each context switch.

5.1.3.2. Notification of Inter-process Exception Using a Toggle Switch

Here, ITI puts itself in the position of being always ready to receive a message from the lower level, regardless of whether or not the application has a read request outstanding. Thus the client is never deaf to the server below. The author has implemented this by adding

Table 5.1 Number of SendReceiveReply Messages on <BRK>

packet no.	ITI as a server no. of context switches	ITI as an I/O filter no. of context switches
1	9*2	2*2
2	3*2	3*2
3	2*2	3*2
4	3*2	2*2
5	3*2	3*2
Total number of context switches	20*2	13*2
Total number of SendReceiveReply exchanges	20	13

[2]nX.25p is the number of additional process switches within X.25 necessary to handle every packet, = 3 for inbound packets and 2 for outbound packets.

another state variable to both the ITI and X.25 servers to act as a toggle switch to exert control on the movement of data from the lower level X.25 to the higher level ITI, as described in Section 4.1.1. When the switch is *on*, X.25 returns both data and interrupts; when *off*, X.25 returns <BRK> signals only, and X.25 queues normal data as before until the client above makes another read request. Unfortunately, this scheme incurs the overhead of 4 extra process switches on each read request - 2 to set the switch *on* before reading data and 2 to set it *off* afterwards.

This overhead can be reduced to 2 extra process switches for every normal-case read by setting a default for X.25 to set the switch *off* automatically after satisfying every read request. Then the ITIMain process controls the switch setting so that when there is an application level read request, it requests data from X.25 by setting the switch *on*.

This default however is not the correct one for application programs such as file readers which do always have an outstanding read to the level below, and which therefore are not deaf to the server. Such programs do not need the toggle switch, which should always be *on*. The default should enable such programs to run with no normal-case overhead. These two aims are incompatible. Thus, if the ITI overhead is reduced to 2 process switches by using a default *off* after every read has been satisfied, then every other client of X.25 must be aware of this default, and must set the switch *on* after every read request. This was considered to be an unacceptable solution, so the author implemented the former design, which increases the context switches on a normal-case read from 14 to 18, a significant amount. At full speed (9600 baud) over the Datapac network, the computer could no longer keep up with one other user using cpu cycles.

5.1.3.3. ITI as a Filter and an Exception Process

Following the model, the author decided to use an inter-process exception notification from X.25 to a separate exception handler process at the ITI layer above for notifying exceptional asynchronous events encountered by ITI from the remote client.

When the server detects an exception, it checks to see if an *exception handler process*, EH, exists, and is ready. If so, the server makes a **Reply** to the EH, renders the EH *unready* and continues processing exactly as before. Thus from the server's point of view, it exports an exception merely by making a non-blocking **Reply** to a registered, ready EH, and then it *unreadies* the EH. If there is a registered EH which is *unready*, the server queues the **Reply** message until the EH makes a **Send** to indicate it is ready again. This approach is taken so that if there is no exception process *ready*, the server only has to queue up a **Reply** message with minimal information. If the queue becomes full, the server can drop subsequent exception replies. The server therefore needs to make no assumptions about the EH at the higher level, thus preserving modularity requirements.

The author observed that the exception process could also aid in structuring ITI as a filter, by allowing the exception process to take control as necessary on a <BRK>, and initiate further read requests (for the IND-OF-BRK packet) and write requests (for the Q-DATA and INT-CONF packets) until the exceptional situation has been dealt with. EH notifies X25Main with a **Send** when it is ready again to handle further exceptions. Exceptional events occurring across the X.25-ITI interface during this processing are queued up in X25 until there is a ready registered exception handler process ready to deal with them. ITI can now be structured as a reader filter (RFIL), a writer filter (WFIL), a timer and an exception process EH, as shown below in Figure 5.4, and the author successfully implemented ITI in this

way.

This structure reflects the major functionality of ITI, that of an i/o filter, better than the server model, and it is structured more efficiently for the normal case (no interrupts or network failures) by *reducing* the number of process switches by 2 on each read/write request, from 14 to 12. Here, the run-time overhead for normal cases has been reduced by a significant margin through restructuring the conventional i/o server by two i/o filters and an exception process. In this implemetation, on <BRK>, the context switches are as shown in Figure 5.5.

Recall that each inter-process **SendReceiveReply** message exchange causes 2 process switches. The number of **SendReceiveReply** messages to handle each network packet are now given in column 2 of Table 5.1.

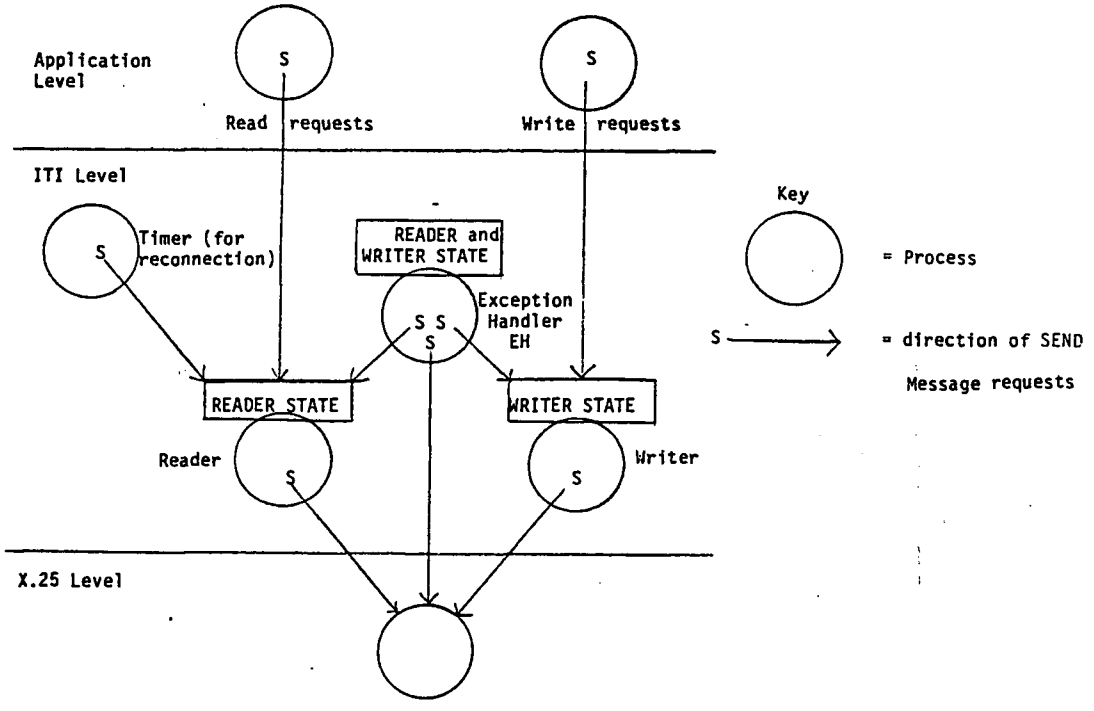


Figure 5.4 ITI as an I/O Filter

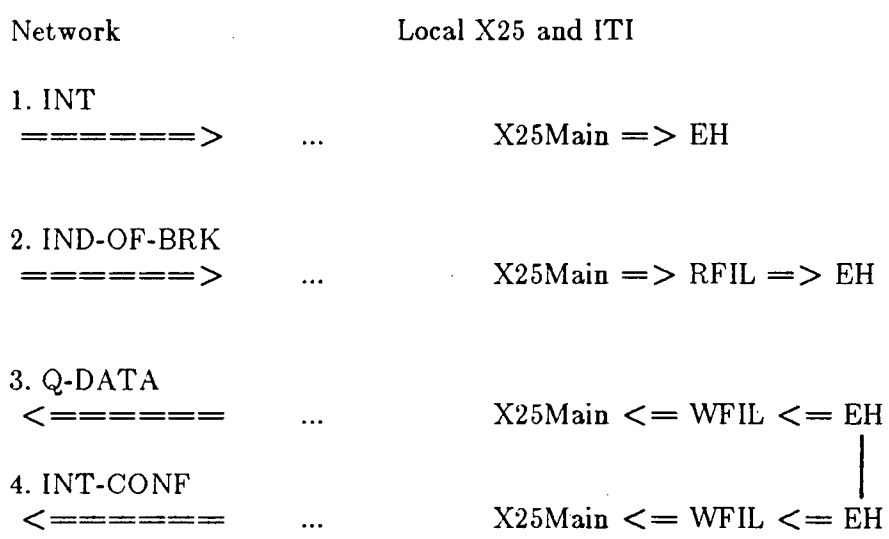


Figure 5.5 Process Switches on <BRK> with a Separate Exception Process

Thus a total of 13 +nX.25p[3] = 26 **SendReceiveReply** message exchanges are used to handle the <BRK> exception, compared with 33 using process destruction with a ITI as a server. When the system was timesharing with another user we observed response times of around 1 second for the <BRK> to be handled. Thus the major effect of reducing the context switches is not only in the actual context switch time (a difference of just 7x3 = 21 msecs), but in the reduction of cpu time slices made available to other users during <BRK> processing.

[3]remember nX.25p is the number of additional process switches within X.25 necessary to handle every packet, = 3 for inbound packets and 2 for outbound packets.

5.1.4. Problems with Synchronisation

The exception handling on <BRK> is *non-atomic* in that intermediate states during exception handling are visible; the exception handling involves several inter-process message exchanges which take a finite time, during which other processes may observe and intervene with the only partially completed handling. This causes problems in synchronisation.

There are at least 2 alternatives for EH to coordinate the 5 packet exchanges on <BRK> previously mentioned. We could either allow the EH to **Send** directly to X25 to read and write the packets, or we could make EH use the services of RFIL and WFIL. The author took the latter approach, as either RFIL or WFIL would probably be interacting with X25Main at the time of the exception, and EH would need to coordinate their actions anyway until the exception handling was completed. If EH was allowed to **Send** to X25Main directly for the packet exchange, RFIL and WFIL could service concurrently (and incorrectly) other requests from their clients. The concurrency during exception handling causes complications.

If the <BRK> occurs during a quiescent period (while X25Main had a deaf client) then the synchronisation appears to be straightforward, as RFIL and WFIL are waiting on a **Receive** for i/o instructions. The following stages should occur during exception handling:

(1) EH is *readied* by X25Main executing a **Reply** to EH.

(2) EH makes a **Send** to RFIL for the INT packet.

(3) RFIL makes a **Send** to X25Main .... and eventually a **Reply** to EH.

(4) EH makes a **Send** to RFIL for the IND-OF-BRK packet.

(5) RFIL makes a **Send** to X25Main .... and eventually a **Reply** to EH.

(6) EH makes a **Send** to WFIL to dispatch the Q-DATA packet

(7) WFIL makes a **Send** to X25Main .... and eventually a **Reply** to EH.

(8) EH makes a **Send** to WFIL to dispatch the INT-CONF packet.

(9) WFIL makes a **Send** to X25Main .... and eventually a **Reply** to EH.

(10) EH considers the job is done, and makes a **Send** to X25Main, indicating it is ready to handle another exception.

However, even in this simple scheme complications arise if ITI's application program (APPL) intervenes with a R/W request during these non-atomic message exchanges. For example, if APPL executes a read request to RFIL during stage 3 above, then after RFIL has made its **Reply** with the INT packet to EH, RFIL executes **Receive** again, without stopping, and at this time only the message from APPL is in RFIL's message queue, as EH has not yet had a chance to execute since being unblocked by RFIL at the end of stage 3. So RFIL will accept APPL's read request and will (eventually) return the IND-OF-BRK packet to its bewildered client APPL instead of to EH.

Thus after a <BRK> exception we need to be able to delay any further interaction of APPL with the filters, until the exception handling is finished. This can be achieved in several ways; the easiest is by setting global flags between EH and the filters, as they are all on the same team and share memory.

But even with global flags, there are situations which are not easily resolvable without the ability to be able to give EH the highest priority on the team. This can arise for example when there is already an outstanding read request from APPL when the exception occurs. The server, X25Main, unblocks EH with a **Reply,** and also, without stopping, unblocks RFIL with a **Reply** with its data -- the INT packet. Both EH and RFIL are ready to execute, and

if RFIL executes first, it can pass the INT packet to APPL as data; APPL can swallow it, and make another read request of RFIL. RFIL can execute next and **Send** again to X25Main, which makes a **Reply** with the IND-OF-BRK packet. RFIL could then return the IND-OF-BRK packet to APPL, before EH even begins to execute.

If we cannot control process priorities, then we must make RFIL aware that an INT packet is special, and subsequent data packets for exception processing must be handled by EH. Such an approach violates modularity, and is very difficult to program correctly. Whereas if it is possible to specify that EH executes first if more than one process on its team is ready, then the problem can be ameliorated. EH executes first, and makes a read request to RFIL to pick up the IND-OF-BRK packet. Meanwhile, RFIL is unblocked with the INT data packet which it gives to APPL as before. Now RFIL can accept EH's read request for data, and pass the IND-OF-BRK packet correctly to EH.

But another complication arises if both EH and APPL make read requests to RFIL while it is busy fetching data. Unless RFIL accepts any messages from EH in priority, then again RFIL could send the IND-OF-BRK packet to APPL.

Recall from Section 4.3 that in Verex a process on a team executes atomically with respect to other processes on a team, till it voluntarily relinquishes the processor, and that a higher priority process always executes before a lower priority one on the same team, even through involuntary time-slice and page-fault deschedules. Both process priority and message priority are available in Verex so the author was able to achieve this complex synchronisation without altering RFIL.

A major problem in writing an exception handler as a separate concurrent process, arises in synchronisation. Although the technique allows clean separation of normal-case code, and

is conceptually modular and well-structured, if non-atomic action is needed during exception handling, special features are needed to write clear, correct programs. Three such features are global (shared) memory, process priority, and message priority for handling the exception process's messages. This latter was also recognised by the implementors of RIG; emergency messages run at a higher priority than regular messages.

In general, we need to be able to specify a partial ordering on events, so that the client is not serviced during the exception processing. Various mechanisms for synchronization in concurrent programming have been succinctly described in [Andrews 83]. The article states that ensuring the invisibility of inconsistent states during an atomic action is an active research area. Some of the most promising results for our problem have been obtained by Andrews in his language, SR (for *Synchronising Resources*) [Andrews 82], and by Liskov's atomic actions, described in the next Section 5.2. The exception handling could be readily specified in SR by using global Boolean flags such as INTERRUPT-PENDING and READ-REQUEST-PENDING, plus scheduling and synchronizing constraints. Use of global flags, although unstructured, does lead to some easier proofs of program correctness than with message-passing alone, as Schlichting and Schneider have shown [Schlichting 82].

Thus we have shown that the use of a separate exception process to handle asynchronous exceptions in the Thoth message-passing environment is both practical and applicable.

To show that there are situations exploiting a separate exception process which are relatively easy to program, we describe in the next Section how an exception process is used to handle another exception: the end-of-file exception which occurs when the virtual circuit is arbitrarily and abruptly terminated.

5.1.5. Recovery after network failure

As described in Section 3.4.2.1., we wish to provide an ITI implementation and session-layer service which would hide a failure in the underlying network from the application layer until either a timeout fired, or the same user logged in again. In the latter case, the application program will be available to the user just as if no disconnection had occurred (except for a brief RECONNECTED message).

The major problem with providing reconnection is that the state of the suspended protocol must be saved; yet in order to allow the remote user to log in again for reconnection, a working version of the protocol must be available. The old and new virtual circuits must then be merged after a successful login.

The author observed that the exceptional event of failure of the underlying network could also be efficiently handled using an exception handler process. As explained in Section 5.1.2, this end-of-file (EOF) failure is communicated synchronously to ITI on its next read or write request to X.25. The **Reply** contains the EOF information.

The exception process is used as follows. There is one exception handler process, EH, for each X.25 virtual circuit. If a network failure on a virtual circuit is recorded, the EH associated with the failed virtual circuit is notified immediately by a **Reply** from the server below. EH tears down the virtual circuit and suspends itself by executing a **Send** request to the process which would be invoked to handle a reconnection. If a timeout occurs before the remote user has reconnected, the timer destroys the suspended session, causing the user to be logged out.

If the user attempts to reconnect via a new X.25 virtual circuit within the timeout period, another EH process is created for this new circuit. The reconnection sequence involves

a check of the remote user's status, and if it is suspended, the old EH associated with the torn-down virtual circuit is unblocked with a **Reply.** The old EH links its session to the new X.25 virtual circuit, and wakes up the reader and the writer, completing reconnection before destroying the new EH. The reader and writer then continue exactly as if the underlying network had not failed, replying with new data to the suspended application.

This application was very straightforward to implement and had no synchronisation problems. The author made one minor change to RFIL so that on receiving an EOF **Reply** from the server below, RFIL checked if an exception handler process was ready to handle the EOF. If so, RFIL voluntarily blocked, instead of immediately terminating the virtual circuit by committing suicide. This reconnection feature provides a very useful addition to the remote login-in service, and has been measured as being invoked by 5% of remote login sessions.

5.1.6. Summary

Two mechanisms for handling exceptions in multi-process protocol implementations have been described. The first mechanism reflects the use of combining the role of a server with a finite state machine. This structure is particularly efficient for the lower level protocols such as X.25, where it is difficult or impossible to define a normal flow of control.

For higher level protocols where exceptional situations occur less frequently, it is more efficient to structure according to the normal flow of control. The server model is not so efficient in these situations - instead, the protocol is structured as an i/o filter with a special exception handler process.

This structure has the following advantages over the server structure:

(1) it reduces overhead in the normal case by reducing the number of process switches on each read/write request.

(2) the normal-case code is separated from the exception code, leading to good modular programming so that, for example, the client and server can be debugged without any exception handling, as the normal-case code is completely separated from the exception-handling code

(3) it provides a clean mechanism for asynchronous server-client notification

(4) it provides a clean queueing mechanism for the exceptions; they are not lost

(5) the increased concurrency allows exception handling to be done in parallel if that is possible (though not in this example)

(6) it can be generalised to other operating systems.

The disadvantage is that the synchronisation of the exception process with other processes may be difficult -- the increased concurrency may be hard to handle.

The author implemented two program paradigms for non-standard exception notification from server to client for synchronous Thoth-like operating systems

(1) using a toggle switch for rendering deaf clients able to hear.

(2) by using an exception process which registers with the server below

The advantage of the latter approach in a hierarchy of servers is that the client (itself a server) can be restructured as an i/o filter, with corresponding increase in performance for normal case event processing, whereas the former toggle-switch solution actually increases the run-time for normal case handling in this example, although it can be used to advantage when the client actually *wants* to be deaf to a device for an extended period, as described in Section

4.1.1.

5.2. A Nested Transaction Mechanism for Atomic Data Updates

5.2.1. Introduction

Mechanisms for achieving mutual exclusion have already been discussed in Section 4.3; one approach is to declare *atomic data* which is operated on by *atomic transactions.* An *atomic transaction,* as defined by Mueller [Mueller 83] is a computation consisting of a collection of operations which take place indivisibly in the presence of both failures and concurrent computations. Atomic transactions differ from non-atomic transactions in that they are apparently indivisible - no intermediate states are observable by another transaction. They are also recoverable in that the effect is *all-or-nothing* - either all the operations prevail (the atomic action **commits),** or none of them do (the atomic transaction **aborts)** so that all the modified objects are restored to their initial state.

There is much current literature on atomic transactions, and, more recently, on *nested transactions* [Liskov 83], [Mueller 83], [Moss 81]. Nested transactions provide a hierarchy of atomic transactions, useful for composing activities in a modular fashion. A transaction which is invoked from within a transaction, called a *subaction* appears atomic to its caller. Subactions can commit and abort independently, and a subaction can abort without forcing its parent to abort. However, the commit of a subaction is conditional: even if a subaction commits, its actions may be subsequently aborted by its parent or another ancestor. Further, objects modified by subactions are written to stable storage only when the top-level actions commit. Thus a nested transaction mechanism must provide proper synchronisation and recovery for subactions.

The implementation of such a mechanism is discussed here, because the implementation is based on the general model for designing software. Liskov's ARGUS project [Liskov 83] has been taken as the specification for nested transactions. In ARGUS, Guardians and Actions are provided as linguistic support for robust distributed programs. The ARGUS project was conceived as an integrated programming language and operating system, but we show that the language can be conveniently and efficiently supported by other distributed operating systems which provide certain features.

We describe a possible implementation of a nested transaction mechanism, structured to exploit exceptions, for distributed operating system kernels such as the V-system [Cheriton 83]. We compare this with the conventional approach to implementing nested transactions as exemplified in the LOCUS Distributed UNIX project, and show how structuring the program (in this case, the nested transaction mechanism) to exploit the exceptions improves the efficiency.

5.2.2. Overview of the ARGUS Model

5.2.2.1. Atomic Objects

ARGUS provides for *atomic abstract data types* linguistically declared as *stable* objects. Only activities which have read or write access to such atomic objects are treated as atomic activities. The implementation of atomic objects is based on simple read and write locks with the usual constraints on concurrent access:

(1) multiple readers allowed, but readers exclude writers

(2) a writer excludes readers and other writers.

When a write lock is obtained, a **version** of the object is made and the action operates on this version. If the action commits, the version is retained, otherwise it is lost. All locks are held till completion of the top-level action, and the commit of a top-level action is irrevocable.

5.2.2.2. Nested Actions

To keep the locking rules simple in nested actions, a parent cannot run concurrently with its children. The rules for read and write locks are extended so that

(1) an action can obtain a read lock if every action holding a write lock is an ancestor.

(2) an action may obtain a write lock provided every action holding a read or write lock on that object is an ancestor.

When a subaction aborts, its locks are discarded. Liskov also states that when a subaction commits, its locks are inherited by its parent. We will show that this is not necessary for correct operation of the mechanism, and in this point our model differs from Liskov's.

5.2.2.3. Guardians

In ARGUS, a distributed program is composed of one or more *guardians*. A guardian provides access to atomic objects through a set of *handler operations,* much like a file server provides access to files. We use the word *guardian* alone, to refer to the location where the atomic object is kept -- equivalent to the *Transaction Scheduling Site* (TSS) of LOCUS [Mueller 83]. In contrast, *guardian handlers* refer to subactions, as described below.

Liskov states that when a handler is invoked, the following steps occur:

(1) A new subaction, SP1, of the calling action is created.

(2) A message containing the arguments is constructed.

(3) The system suspends the calling process and sends a message to the target guardian. If that guardian no longer exists, subaction SP1 aborts and the call terminates with a *failure* exception.

(4) The system makes a reasonable attempt to deliver the message, but the call may fail.

(5) The system creates a process and a subaction SP2 (of subaction SP1) at the receiving guardian to execute the handler. Note that multiple instances of the same handler may execute simultaneously. The system takes care of locks and versions of atomic objects used by the handler in the proper manner, according to whether the handler commits or aborts.

(6) When the handler terminates, a message containing the results is constructed, the handler action terminates, the handler process SP2 is destroyed, and the message is sent to the caller SP1. If the message cannot be sent the subaction SP1 aborts, and the call terminates with a *failure* exception.

(7) The calling process continues execution.

A guardian makes these resources available to its users by providing a set of operations, called handlers, which can be called by other guardians to make use of these resources. Given a variable x naming a guardian object, a handler h of the guardian may be referenced as $x.h$. A handler can terminate in either normal or exceptional condition. A handler executes as a subaction, so it must either commit or abort - either normal exit or exceptional exit may be committed or aborted. If the guardian crashes, all the active handler calls are aborted and its atomic data must be retrievable from stable storage.

Liskov's nested transactions are similar to the nested transaction model of LOCUS, however, in LOCUS, the only atomic data supported is of type *file*. To aid in comparison of our approach with that of Liskov and LOCUS, we use atomic data only of type *file* in our examples.

5.2.3. Implementation Considerations

5.2.3.1. Introduction

In the ARGUS model of nested transactions, and in the LOCUS implementation, the relationship between client and guardian is not just that of a one-off request for data -- instead, close links are retained so that the system can notify guardians immediately a lock-holder commits or aborts. Parents inherit their children's read/write locks on termination -- the list of such locks is called the *participant file list*. Their implementations are structured so that if a subaction is successful it must explicitly execute a commit at termination. This causes COMMIT messages to be sent to every participant. Thus each action must maintain a list of participants. The list of participants locked by the committing child is then transferred to its parent. The parent transaction thus becomes aware of its childrens' deeds, which violates modularity concepts, both in the language ARGUS, and in the supporting operating system. We claim to preserve these boundaries in our implementation.

Further, their implementations are structured for functionality rather than efficiency, as there is a high overhead in the case of no aborts and no R/W conflicts. This is counter to the principles for exception handling, and our implementation which follows the model is considerably more efficient.

It is assumed that

(1) subactions commit more frequently than abort

(2) all atomic data is frequently available for R/W access; conflicting R/W requests are rare.

The principle followed in our implementation is to structure the system to minimise the run-time by reducing the inter-process communication in the statistically dominant case, and also to reduce the context saved in the normal case, by maintaining the minimum context necessary to achieve correctness. Thus when exceptions do occur, the exception-handling is costly. However, by making the normal case more efficient we reduce the chance of conflicting data accesses and this in turn may further reduce the number of aborts.

We now describe the minimum context necessary to achieve correctness, before presenting an overview of the features of our implementation. We then make a comparison of the efficiency of our implementation with others.

5.2.3.2. Requirements for correct implementation

To handle aborts, each guardian needs the ability to detect whether any of its atomic data has been updated by the aborted process, and if so, to restore a valid version of the atomic data, prior to that changed by the aborted process.

To handle R/W conflicts, each guardian needs the ability to detect whether its atomic data are free, and if not, whether any write-lock owners are not ancestors of the requesting action.

To handle aborts correctly, each guardian of atomic data needs to keep a stack of versions and list of actions holding each lock. For conflict resolution, each guardian needs to keep a current version and a list of actions and its ancestors holding each R/W lock.

We assume that each transaction is uniquely identified in the network by its *transaction unique identifier* (Tid). We assume also that it is possible to determine from a transaction's Tid both the home site of the transaction (the site where the transaction begins executing) and the Tid's of all the transaction's superiors. We thus encode the transaction's ancestors in its identifier, and can merge the version list with the ancestor list, so that each guardian needs just one volatile data structure, called a *t-lock* (after LOCUS) to hold the locking and recovery information for each atomic object.

The t-lock structure is as follows:

```
Tlock          = Struct [ ReadRetainers, Top]
ReadRetainers  = List [ Tid]
Top            = Struct [ Vstack]
Vstack         = Stack[version, Tid]
```

Another requirement for correct operation is the existence of a communication facility which can be used by the guardians to receive *ABORT* and *TOP-COMMIT* messages reliably. We discuss mechanisms for achieving this in Section 5.2.4.

However, we show in our implementation, that guardians need not be *immediately* informed of sub-action commits for correct operation, provided the guardian can determine whether a transaction holding a R/W lock is still alive or not. So although the guardian's version-owner may be out-of-date (can occur when the committed subtransaction T dies), the guardian can still operate correctly, in that it does not use out-of-date versions. This is achieved because the guardian keeps the current version on the top of the version stack, together with what it thinks is the current owner. The version stack is updated whenever there is need to resolve a potential R/W conflict, with the youngest living ancestor of T replacing T in any owner's list. Alternatively, the guardian can use a background process to update its t-lock periodically.

5.2.4. Features of the Implementation

A *transaction group* and a *transaction group leader* (GpId) are used to implement nested transactions correctly and efficiently. Each new top-level transaction creates a new process, called the group leader, and all its subtransactions use the same GpId. Any guardian of atomic data accessed by any transaction, notifies its group leader, so as to receive ABORT and TOP-COMMIT messages. Such messages are sent by transactions to the group leader, who in turn reliably notifies all the guardians in that group. The group leader may use a simple 1:1 scheme for notification, or alternatively may use a multicast or broadcast feature. We digress briefly to describe such a mechanism which can be used advantageously.

An unreliable multicast feature is available in the distributed V-kernel [Cheriton 84]. The multicast function is achieved through a Group-id. A process which subscribes to the Group-id will receive messages sent to it. No group membership list is maintained, and so members of a group are only loosely coupled, and messages are not reliably delivered. We show that this may be extended simply to a reliable multicast which fulfills the requirements. The algorithms used by the group leader, and their performance, are discussed fully in Sections 5.2.4.7- 5.2.4.9.

The implementation is structured so that parents do not inherit their committed children's locks automatically. Indeed, when a child commits, which it can do only on its death, the waiting parent is merely informed of that fact, and the participating guardians are not informed of the child's death. However, if a subsequent attempt is made to access such a guardian's atomic data, the guardian must then be able to detect the status of its lock-holders, and update its t-lock data accordingly.

To handle the commit of a top-level transaction, the action must send a TOP-COMMIT message to the group leader. All guardians to which the transaction group has, at any time, held a lock, then receive the TOP-COMMIT message. Each guardian must then commit to stable storage any current versions held by that transaction group.

In the event of network partitioning, the problem is the same as that discussed in the LOCUS paper, and no top-level transaction can commit. We provide a user override for these cases, as *any* failed component can prevent commitment. Orphan transactions will not commit; they should eventually detect the decease of their top-level ancestor and should duly abort.

Many message-based operating systems provide the software tools necessary for efficient and correct implementation; namely that transactions may be implemented as processes, and that parent transactions may await replies from their children subactions. Processes should be cheap to create and destroy dynamically so that each invocation of a guardian handler can be efficiently implemented as a new process, as specified in ARGUS. Further, context switching between processes should not be too expensive. These criteria are all fulfilled by Thoth-like operating systems, and even UNIX 4.2 processes communicating over datagram sockets can be used for functional correctness, if not efficiency. A synchronous **Send-Receive-Reply** inter-process communication facility is needed -- again, this is available in the Thoth derivatives, and can be simulated with UNIX datagrams, as we show in our examples, details of which are now given.

5.2.4.1. Transaction Invocation

The *calling process, P* may invoke another transaction T either as a nested top-level action (only if P is also top-level), or as a subaction, usually by making a handler call to a

guardian, viz. *g.h.*

The following algorithm is employed:

(1) At the site of P, a new process, pid P1 is created for the new transaction T. (If P1 is to be run at a remote site, an appropriate remote pid is generated). If the invoked transaction is a top-level transaction, it creates a new group leader, whose unique process-identifier is GpId.

(2) P1 is added to the list of P's children.

(3) The complete ancestor list, and the GpId is passed to P1 and P1 is readied.

(4) The parent P awaits completion of P1 by executing a Receive(specific) from P1, or if several children are initiated concurrently, the parent must arrange to be notified of each one's death and its status (COMMITTED or ABORTED).

5.2.4.2. Atomic Data Access

Any transaction T wishing to obtain R/W access to atomic data must perform the following algorithm. Note that this algorithm will usually be executed by the atomic data's guardian upon receipt of an OPEN request.

(1) If a t-lock for the data does not exist, one is created, and a top version stack entry is made from the state maintained in non-volatile storage.

(2) If the request is for Read and Write access, the request is denied if any other living transaction holding a lock is not an ancestor. If the request is granted, a new version stack entry is made for T containing a copy of the top of the version stack.

(3) Otherwise if the request is for Read access, the request is denied if any other transaction holding a write lock is not an ancestor of T. If the request is granted, an entry for T is

added to ReadRetainers.

(4) For successful requests, the guardian checks to see if this transaction has a GpId different from the GpId of any other reader or writer. Note that all writers must have the same group leader, by the locking rules, but readers may be in different groups. If this is a new GpId, the guardian notifies the group leader so it can receive ABORT and TOP-COMMIT messages.

5.2.4.3. Subaction Commit

No special action is taken. The parent receives information that the child died, committed, and the parent continues normal execution.

5.2.4.4. Top-level Commit

The committing toplevel action, T, must send a TOP-COMMIT message to the group leader. The group leader executes an algorithm described below in Section 5.2.4.8. to send COMMIT messages to each guardian which has joined the group. All guardians to which any transaction of the group has, at any time, held a lock, receive the COMMIT message (eventually). In the case of network partitioning, not all guardians may immediately receive the COMMIT message. This problem may be solved as in the LOCUS description, and will not be considered further here. As parents cannot run concurrently with children, all children must by now be dead (or unable to receive the message). Each guardian must then commit any current versions held by that group. The group leader uses a standard 2-phase commit protocol such as described by Moss [Moss 81]. After writing data to stable storage and committing, the guardian removes the t-lock structure together with all the version lists.

5.2.4.5. Transaction Aborts

To handle a transaction ABORT, all the guardians in the aborting transaction's group must receive the ABORT message and the transaction-id P of the aborting process. The mechanism for the receipt of ABORT is via the group leader, as for TOP-COMMIT. The guardian checks whether P is a ReadRetainer of a version, then performs the following algorithm for each atomic data item.

(1) If P is a ReadRetainer of a version, or in the ancestor list of a ReadRetainer, P is removed from the list; goto step 4.

(2) Otherwise, the guardian checks, starting at the bottom of its version stack, whether P is a WriteRetainer, or in the ancestor list of a WriteRetainer of a version.

(3) If a match is found with a version, say v_i, the oldest version prior to v_i is restored, and the owner is set equal to P's ancestor.

(4) If there are no remaining R/W retainers, the t-lock is removed.

To clarify this, an example of the formation of a version stack with nested transactions before and after a transaction aborts is shown in Figure 5.6 below.

Thus, ABORT causes the version stacks and the ReadRetainers and WriteRetainers to be updated[4].

5.2.4.6. Transaction Read/Write Conflicts

To handle conflicts correctly, the guardian must check for each atomic data access attempt, if there is a R/W lock on the data.

---

[4]Note that we must check to see if the owner held several versions -- if so, the oldest must be restored. Therefore the version stack must be checked from the bottom-up for completeness, even though

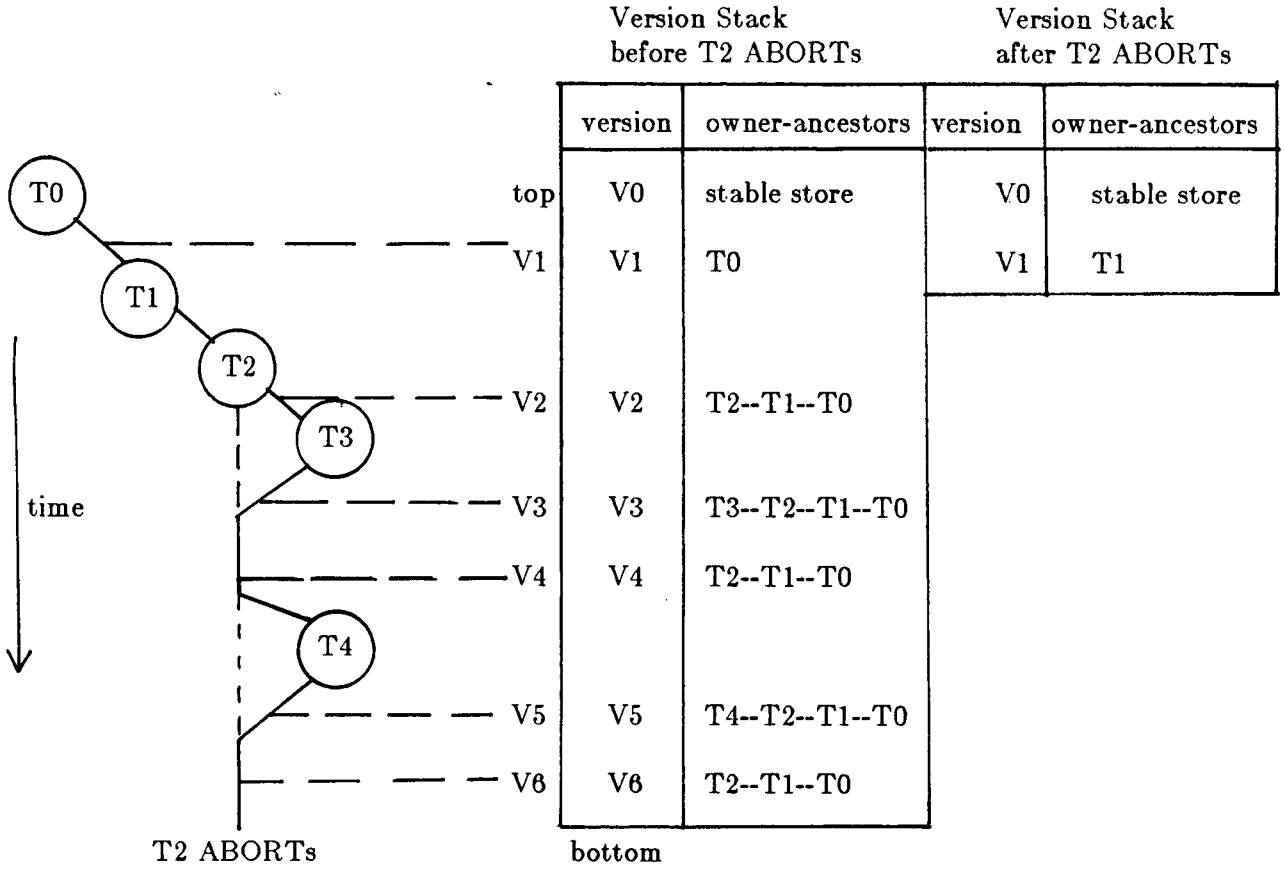


Figure 5.8 Formation of a Version Stack with Nested Transactions

(1)

If no R/W locks are held, access is permitted and a normal data access is performed as described above.

(2) If a R/W lock is held on the data, we search to see if the current version owner is an ancestor of P, by checking whether the owner is still alive. In doing so, the top of the version stack is updated. If there is a conflict, access is denied.

---

the most likely ABORT is the current version holder.

5.2.4.7. The Group Leader's Role

Each group leader maintains a list of participants by accepting information from a guardian whenever a transaction in a new group receives access to some of the guardian's atomic data (i.e. by gaining a R/W lock). The group leader also accepts TOP-COMMIT and ABORT messages from transactions in the group. The structure of this communication is shown in Figure 5.7 below.

On receipt of an ABORT or TOP-COMMIT message[5] from a transaction, the group leader has to notify the participants in its list. For this purpose, the GL uses a worker process on the same team, whose role is to provide two-way communication as described previously in Section 2.3.2. The worker has access to the participant list maintained by the GL, and exploits this fact to handle the TOP-COMMIT protocol, unaided by GL, thus avoiding extra context switches and ipc between the worker and the GL. Only when the TOP-

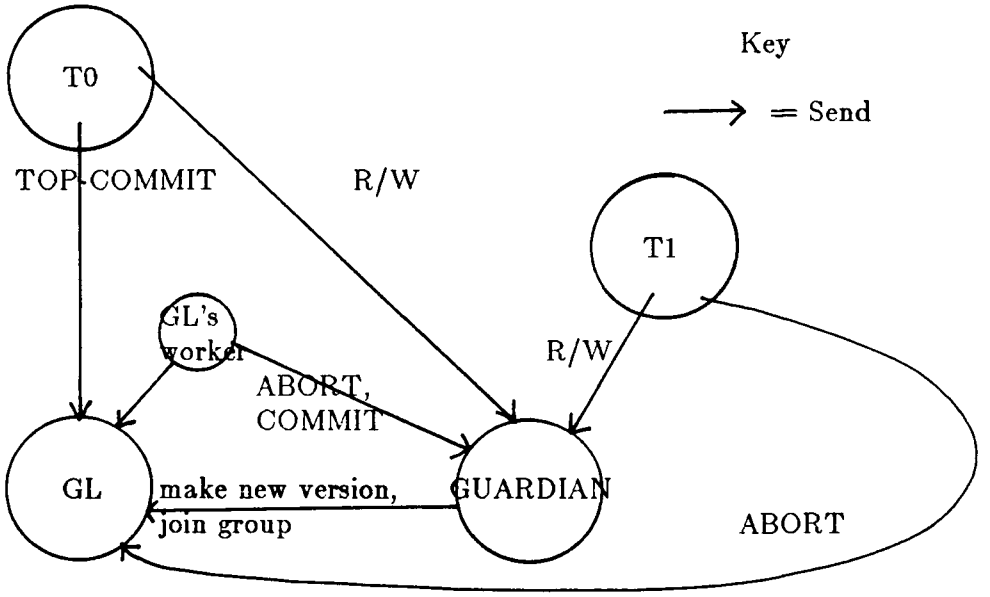


Figure 5.7 Group Leader Communication Structure

[5]Note that ABORT messages must always take priority over TOP-COMMIT

COMMIT is resolved does the worker re-Send to its GL. The worker thus acts like an exception process described earlier. There are no synchronisation problems between the worker and its GL because the TOP-COMMIT message should be the last from the group.

Similarly, during the processing of ABORT the worker only re-Sends to its GL when the ABORT is completed. If more ABORT's for the group are received by GL while its worker is processing an earlier one, the GL must save them up.

To avoid multiplexing and possible delays, we specify one GL per transaction tree group. Usually it is created on the same host as the top-level transaction, although for efficiency in execution of the 2-phase COMMIT protocol the GL should be on the host which has most participants. Thus all group leaders execute the same code, first creating their worker to manage the 2-way communication between participants and the GL.

5.2.4.8. Group-Leader's TOP-COMMIT Algorithms.

The GL executes the following algorithm on receipt of a TOP-COMMIT message from the group's top-level transaction T.

(1) Reply **TOP-COMMIT(T)** to worker.

(2) Wait to receive next message.

The worker executes the following algorithm when there is no multicast or broadcast ipc mechanism.

(1) Send to GL for instructions

(2) On receiving the Reply **TOP-COMMIT(T)**, obtain the list of participants.

(3) For each participant, the worker must Send a **PREPARE(T)** message.

(4) If the participant replies **ABORT(T)**, then the whole commit must fail, and all the transactions must be aborted. Go to step 8. If the participant does not reply within a timeout period, the worker can either ABORT (goto step 8), or retransmit the **PREPARE(T)** message. If the participant responds **PREPARED(T)** then goto step 3 unless each participant has been notified.

(5) All the participants have responded **PREPARED(T)** at this point. Record in permanent memory a list of the participants along with a notation that transaction T is now *COMPLETING*.

(6) For each participant, Send a **COMMIT(T)** message. The participant *must* respond **COMMITTED(T)**. If there is a participant which has not responded after a timeout period, the message **COMMIT(T)** is retransmitted.

(7) When all participants have responded, erase the list of participants and associated memory from permanent memory. Reply **COMMITTED(T)** to GL. Done.

(8) Send the message **ABORT(T)** to each participant. When all participants have responded, reply **ABORTED(T)** to GL. Done.

The participants act as follows:

(1) If a **PREPARE(T)** message is received, and **T** is unknown at the participant (it never ran there, was locally aborted, or was wiped out by a crash), then respond **ABORT(T)**. Otherwise, prepare the transaction locally as follows. Write the identity and the new state of any objects *write-locked* by T to the permanent memory. However, the old state of the objects is not overwritten -- the point is to have both the old and the new state recorded in permanent memory, so that the transaction can be locally committed (by installing the new states) or aborted (by forgetting the new states) on demand,

regardless of crashes. This write to permanent memory must be performed as a single atomic update. When a transaction is prepared, its **read** locks may be freed immediately, but **write** locks must be held until the transaction is resolved. The participant must then reply **PREPARED(T)** to the GL.

(2) If an **ABORT(T)** message is received, then locally abort T. If T is *PREPARED* erase from permanent memory the potential new states of objects modified by it (using a single atomic write), and then perform the usual version restoration associated with transaction abort.

(3) If a **COMMIT(T)** message is received, then T must have been locally prepared, and is either still prepared, or already committed. If it is prepared, install the tentative new states of objects modified by T in both permanent and volatile memory, discarding the old states of these objects[6]. Finish committing the transaction by releasing all its locks? If T is no longer locally prepared, then it has already been committed, and no special action is required. In either case (T is locally prepared or not), respond **COMMITTED(T)** to the GL when done.

(4) If a transaction has been prepared for a long time and nothing has been heard from the GL, then ask it for the state of the transaction. The GL replies **PREPARE(T)** if it is still preparing, **COMMIT(T)** if it is committing. If the GL is not running, or there is no record of the transaction, then the node where the GL ran should respond **ABORT(T)**.

Moss argues [Moss 82] that this algorithm is correct and will always work eventually. If there is a crash at a participant after the first phase, the permanent memory contains the

[6]again, ensure that the update is achieved atomically with a single write

relevant information for recovery. If a transaction was not yet completed, then the crash will wipe it out, and the transaction will be ABORTED. Further scenarios are explained in Moss, but only the case of the coordinator failing may cause some participants to lock data forever. A manual override must be used in such cases - the data is, however, correctly preserved on permanent storage.

Alternative algorithms may be used by the worker if a multicast or broadcast ipc mechanism is available, which reduces the ipc and message traffic still further.

(1) The GL broadcasts or multicasts (Bsends) a **PREPARE(T)** multicast message, and waits till all the replies have been picked up.

(2) If the GL receives an **ABORT** reply, then go to step 7.

(3) If the GL receives all **PREPARED(T)** replies within the timeout period, it records in permanent memory a list of participants along with a notation that transaction T is now *COMPLETING*. Otherwise it can retransmit **PREPARE(T)** messages on a 1:1 basis as in step 4 of the previous algorithm, and if the timeout still expires, the transaction must be aborted. (go to step 7).

(4) All the participants have responded **PREPARED(T)** at this point. Now Bsend a **COMMIT** message.

(5) As step 6 before.

(6) As step 7 before.

(7) BSend an **ABORT(T)** message. When all participants have responded, reply **ABORTED(T)** to GL. Done.

5.2.4.9. Group Leader's ABORT Algorithms

The GL executes the following algorithm on receipt of an ABORT message from any transaction T of the group. It is similar to the COMMIT algorithm described above.

(1) Reply **ABORT(T)** to worker.

(2) Wait to receive next message.

The worker executes the following algorithm when there is no multicast or broadcast ipc mechanism.

(1) Send to GL for instructions

(2) On receiving the Reply **ABORT(T)**, obtain the list of participants.

(3) For each participant, Send an **ABORT(T)** message.

(4) If the participant does not reply within a timeout period, the message is retransmitted. If the participant responds **ABORTED(T, NONE)** then that participant has indicated there are no more transactions in that group holding R/W locks, so the worker can remove that participant from its list.. If the participant responds **ABORTED(T, MORE)** then that participant has indicated there are more transactions in that group holding R/W locks, and the worker cannot remove that participant from its list. Goto step 3 until each participant has been notified.

(5) All the participants have responded **ABORTED(T)** at this point. Reply **ABORTED(T)** to GL. Done

Extension of the algorithm may be made if a broadcast or multicast feature is available, as for the 2-phase COMMIT described above.

5.2.5. Efficiency of Mechanisms

A comparison of the LOCUS and V-kernel implementations is shown in Table 5.2. In LOCUS, for each subaction opening n atomic files and later committing, there are 4+2n messages to parent and TSS. For top-level commit a 2-phase protocol involving 4n messages (to TSS) is executed.

Thus if T -> T1 -> T2 -> T3 -> n atomic files there are 3*(4+2n) messages between T,T1,T2,T3 and the TSS, plus 4n messages for 2-phase top-level commit. This is shown in column 1 of Table 5.2.

Thus, contrary to the claim in [Mueller 83], the 2-phase commit does not dominate at 3 or more levels of nesting of transactions.

The number of *message events,* where a message event is described as a message being sent or received, is equal to 2 * (no. of messages) in this case, as each message is 1:1.

As the process context switching is often a dominant time-cost, we use this measure, which is equal to the message events. So in the above example, in the case of no aborts or

Table 5.2 A Comparison of Locus and V-kernel Nested Atomic Actions

no. of ipc messages NO ABORTS	LOCUS	V-kernel no broadcast	V-kernel broadcast
Level m subtrans. opening n files	m*(4+2n)	4+2n	4+2n
Level 3 subtrans.	3*(4+2n)	4+2n	4+2n
TOP COMMIT	4n	2+4n	2+2n
TOTAL messages	12+10n	6+6n	6+4n
ABORTS			
Level m subtrans. aborts	2n	6+4n	7+3n
Level 1 trans. aborts	12+8n	6+4n	7+3n

data conflicts, the LOCUS system requires 24 + 20n context switches.

For our scheme, for each subaction commit there are no context switches. Therefore the only messages for the above model are to create the GL, to establish the participant list and to execute the 2-phase commit protocol. The cost of establishing the GL is the overhead of 2 process creations (for GL and its worker) = 4 ipc messages in V-kernel. Each participant in the group, P, does a Send to the GL. Therefore there are 2n messages for n participants, shown in column 2 of the Table.

For top-level commit (no broadcast) the 2-phase protocol involving 2 messages from GL to its worker, and 4n messages between the worker and the participants is executed. Thus, after the 2-phase top-level commit for successful update of n files (without broadcast), (4+2n) + (2+4n) = 6+6n ipc messages, requiring 12 + 12n context switches have been executed, as shown in the Table. This compares favourably with LOCUS' 24 + 20n context switches.

If a multicast algorithm is used to broadcast the TOP-COMMIT message in the V-kernel, only 2+2n messages instead of 2+4n are required, as shown in column 3 of Table 5.2.

Let us consider what these figures mean in real time. If n=5 and the context switching time is 10msecs (as it is for a Send-Receive-Reply between two UNIX 4.2BSD processes on a VAX 11/750), LOCUS takes 124*10 msecs =1240 msecs to commit (ignoring message transit time), whereas our system takes 10(12+12*5) = 720 msecs (no multicast), and with multicast takes 10(6+20) = 260 msecs. Thus our algorithm is considerably faster in the normal-case of no aborts or conflicts.

We now consider the exceptional case of a transaction aborting. In the LOCUS system, if a subtransaction ABORTs, a message is sent to each of the aborting subtransaction's participants (including those inherited from its committed children). If the transaction T has n

participants, and if the lowest subtransaction ABORTs, LOCUS uses 2n messages. If the highest subtransaction ABORTs, LOCUS uses 3*(4+2n) + 2n = 12 + 8n messages.

In our system, the subtransaction sends an ABORT message to the GL which in turn sends ABORT to each participant before replying to the aborting transaction. Thus in our system, if the lowest transaction ABORTs we have the following:

On ABORT, 2 + either 2n (no multicast) or 1+n (if multicast) messages

So the TOTAL messages (1 ABORT, no timeouts) = 6 + 4n (no multicast) or 7 + 3n, as shown in the Table. This is the same for any level of transaction aborting.

Thus our system is faster than LOCUS to handle an ABORT *if the aborting transaction is more than 2 levels higher than the file user.* This situation would commonly occur in practice: the file-opener would be the lowest-level transaction, and it might well be aborted by a grandparent. Thus even the exception-handling can be faster in our system, and the normal-case always is faster.

CHAPTER 6

Conclusions

This thesis is another step towards the time when concurrent distributed programming is commonplace. Its contribution is in identifying exceptional events in the context of systems software, and in the development of a design model illustrated by many examples and program paradigms which show how systems programs may be improved by exploiting exception mechanisms -- either by making the programs more efficient at run-time (according to certain criteria), and/or by making the program code easier to write through employing incremental program design. These software engineering techniques are applicable to all event driven systems where the relative probability of the events is known *a priori*; such systems include hierarchic systems servers, communications protocols and utility programs. Thus the model is an extremely useful tool for solving a wide range of systems software design problems.

6.1. Summary of results

The thesis establishes a set of general principles for exploiting exceptions in systems programs by proposing a general model to achieve improved performance and functionality according to desired objectives. The thesis also describes program models to carry out these objectives, and gives two extended examples which show the practicality of the design methodology.

It is appropriate here to summarize these results, in the light of achieving the goals of the thesis, which are to show how to design efficient and modular systems programs exploiting exception mechanisms. The goals are elaborated as follows: to characterize the nature of

exceptions in operating systems, to develop a model for designing efficient and modular software in a multi-process environment, and to provide program paradigms which employ the design principles by exploiting exception mechanisms.

The goals are examined separately in the following subsections.

6.1.1. Characterize the Nature of Exceptions

An event-oriented view of systems has been taken, where an *event manager* responds to events, which may be externally or internally generated. This thesis shows that there are many systems programs which may be treated as event driven. Systems servers (or monitors), i/o library routines, and, less obviously, some data-driven utility programs come under this description. The exceptions encountered by an event manager may be handled either by the manager itself -- called peer exceptions -- or by the client at the level above -- called server-client exceptions. In multi-process systems, where a client is a process separate from the resource which it is using, exceptions from the event manager (called a server) to the client are called *inter-process exceptions*. Such exceptions, where the handler is at a different level to the detector, have been classified along three dimensions: viz. ignorable v. exceptions which demand attention, broadcast v. 1:1 exceptions, and asynchronous v. synchronous exceptions.

These distinctions occur because the exception mechanisms may be different for the various classes of inter-process exceptions. For example, notification of ignorable exceptions may be different to that for exceptions which must be handled in a distributed system where a cheap unreliable message delivery may be exploited for ignorable exceptions. Similarly, broadcast exceptions from a server to several clients may be used to develop further cooperation between the client processes, and may therefore be handled differently to the exceptions which

are 1:1 from server to client.

This classification of exceptions therefore provides a very useful basis for research into exception mechanisms, and for multi-process systems design exploiting exception mechanisms.

6.1.2. Model for Systems Design

For event-oriented systems in which the relative probabilities of the events are known *a priori*, an appropriate design objective is chosen. The thesis model considers three objectives, namely: minimising the average run-time, minimising the exception handling time, and (less obviously) increasing the program's functionality. This latter objective was chosen as a new approach to systems design, which is particularly useful and appropriate in a multi-process environment.

Programs which are designed to minimise the average run-time should use the following design principles:

(1) The events are divided into 2 groups; the so-called *normal* events, and the other, *exceptional* events.

(2) Cost-benefits may be made in the detection, handling, or context-saving of the normal-case events depending on the number and probabilities of the events.

(3) The program should be designed to minimise the run-time by either reducing the detection cost of normal events, reducing the handling cost of normal events, or reducing the context-saved while processing normal events. This design may mean that the program is structured according to *functionality* rather than its logical purpose. In other words, it may pay to cut the program into vertical slices of function, rather than horizontal slices of modularity. One approach is to structure the inter-process communication to minim-

ise the message-flow in the statistically dominant cases, by recognizing the most common events in the system, and ensuring that they are processed with the minimum number of context switches. This can be achieved by choosing the event manager process which eventually handles most common events to handle all the events first, by adding request exceptions. This event manager processes the common requests quickly without any extra context switches, and it forwards the exception requests to other processes for handling. Another approach is to execute exception-handling code in parallel where possible by distributing the exception-handling over multiple processes.

A general principle which leads to better design in a multi-process or distributed system, is to consider exception handlers as separate processes. The notion of *ignorable* exceptions, in the sense that if they are not handled they have no effect on the correct operation of the program, may also be used to improve program functionality. These two tools used together enable a system to be developed without an exception handler, and then its functionality can be increased by the addition of a handler. Further, broadcast exceptions can be used to enhance cooperation between the clients of a server, and cheap unreliable message delivery can be safely used for ignorable exceptions.

The many examples illustrate that applying the model enables multi-process programs to be structured simply and in a modular way.

Thus the model provides a very powerful software engineering tool for systems design methodology.

6.1.3. Program Paradigms for Systems Design

The design principles have been extended into usage models when the design alludes to a system dependent feature, such as in the inter-process exception notification scheme.

One example is given in several operating systems and high level languages, for multiplexing of asynchronous i/o events, where a server needs to notify a client of asynchronous input from more than one input device. Another example shows how cooperation can be achieved between the clients of a storage allocator server, using ignorable, broadcast messages, which may be unreliably delivered for the most efficient communication between processes. Finally, system dependent mechanisms are given for handling the mutual exclusion problems which may arise if the exception handler is implemented as a separate process, including the use of language-level *atomic actions*.

In this way, the properties are isolated which high-level languages should have for implementing the design ideas of the model. Hence the systems and language approaches to systems design have been unified.

6.1.4. Program Examples

Two programs are described which were designed by the author to test the applicability and effectiveness of the model. The programs have improved structure and function by exploiting exceptions. Both programs run more efficiently than their counterparts which do not exploit exceptions. These examples show that the principles and tools are both practical and effective.

6.2. Further Work

The trend towards distributed systems continues -- to loosely-coupled systems without shared memory, tightly-coupled multiprocessor systems with shared memory, and towards hybrid systems, reflecting the move towards computer systems as concurrent cooperating modules executing an algorithm. New techniques for managing these distributed systems are

required. The principles enunciated in this thesis for multi-process systems extend naturally into the domain of distributed systems.

One open question concerns the techniques for exploiting unreliable messages in networks of computers. The use of broadcast (or multicast) messages is very convenient in local area networks where all the nodes receive the same message, thus reducing network traffic over many 1:1 messages. However, in long haul networks, characterised by multiple hops between source and destination machine, the network traffic is usually increased for broadcast messages, because of the routing algorithms. Yet it is in precisely these networks that the reliable message delivery is most costly to provide (in the sense of elapsed time for acknowledgements and in protocol processing overhead). The tradeoff between broadcasting unreliable messages and providing reliable 1:1 message delivery should be examined in these contexts.

The use of broadcast/multicast inter-process communication is not yet well developed; there are open questions on the best semantics of a broadcast Send; e.g. should the sender be able to specify whether to wait for one, many or all replies. The use of broadcast Send for exception mechanisms impacts these semantics, as the idea behind exception mechanisms is to take the load off the normal case processing which can then be made as efficient as possible. Hence an unreliable broadcast Send in which the sender cannot specify, or need to know, the number of replies is considerably cheaper than a broadcast Send where a list of all recipients is needed for checking that all replies have been received. Performance estimates of these alternatives would be valuable in deciding the protocols and semantics of broadcast communication.

Another open question concerns the implementation of transaction scheduling sites (TSS) for atomic actions. The idea behind a TSS is to protect atomic data, (which is shared by

several processes), and is crucial for cooperation of access to shared data between exception handlers and other processes. Shared data between the processes should be specifyable as *atomic* for simple synchronisation, and such atomic data should be efficiently managed by a TSS. Further work on this could include enhancements to high level languages such as Modula-2 in which cooperating systems are written, to provide easy methods for specifying exceptions and exception handling in this way.

Efficient algorithms for expedient and fair selections of events should be developed for event managers; the task of deciding which ready event to handle next cannot be left to the operating system in a distributed environment.

New mechanisms are needed for distributed highly parallel computations, to reflect needs in both systems software and applications software written in high level languages. A higher-level programming problem which could employ our principles for good design, by the use of exception handling facilities in multiprocess and/or distributed operating systems, is the SMALLTALK system [Goldberg 84]. SMALLTALK is an important step towards message-based concurrent programming in AI applications. If it could be distributed over a local area network by enhancing the language with unreliable exception notification, its programming power might be considerably enhanced.

References

1. Ahamad, M. and Bernstein, M.J., "Multicast Communication in UNIX," *Proceedings 5th International Conference on Distributed Computing Systems*, pp. 80-87, May 1985.
2. Andrews, G.R., "Synchronizing resources," *ACM Transactions on Programming Languages and Systems*, vol. 3(4), pp. 405-430, 1981.
3. Andrews, G.R., "The Distributed Programming Language SR--Mechanisms, design and implementation," *Software P&E*, vol. 12(8), pp. 719-754, August 1982.
4. Andrews, G.R. and Schneider, F.B., "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15(1), March 1983.
5. Atkins, M.S. and Knight, B.J., "Experiences with Coroutines in BCPL," *Software P&E*, vol. 13(8), Aug. 1983a.
6. Atkins, M.S., "Exception Handling in Communication Protocols," *Proceedings 8th Data Communications Conference*, Oct. 1983b.
7. Atkins, M.S., "Notations for Concurrent Programming," *Surveyors' Forum, ACM Computing Surveys*, vol. 15(4), Dec. 1983c.
8. Beander, B., "VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger," *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Aug. 1983.
9. Bentley, J.L, "Writing Efficient Programs," in *Prentice-Hall Software Series*, New Jersey, 1982.
10. Black, A., "Exception Handling - the case against," in *University of Washington Tech Report*, 1982.
11. Bochmann, G.V., "Finite state description of communication protocols," *Computer Networks*, vol. 2(4/5), pp. 361-372., Oct. 1978.
12. Bourne, S.R., Birrell, A.D., and Walker, I., "ALGOL68C Reference Manual," *University of Cambridge Computer Laboratory,*, 1975.
13. Bron, C., Fokkinga, M.M., and Haas, A.C.M. De, "A Proposal for dealing with abnaormal termination of programs," *Twente University of Technology, Mem.Nr.150, The Netherlands*, Nov. 1976.
14. Brownbridge, D.R., Marshall, L.F., and Randell, B., "The Newcastle Connection or UNIXes of the World Unite!," *Software P&E*, vol. 12(7), pp. 1147-1162, 1982.
15. Bunch, S.R. and Day, J.D., "Control Structure Overhead in TCP," *Proceedings IEEE Trends and Applications: Computer Network Protocols, Gaithersburg, Maryland*, pp. 121-127, May 1980.
16. CCG,, "Datapac Interactive Terminal Interface (ITI) Specification," *Trans Canada Telephone System, Ottawa, Ontario*, Oct. 1978.
17. CCITT,, *Interface between DTE and DCE for terminals operating in the packet mode on public data networks*, March 1976.
18. Chanson, S.T., Ravindran, K., and Atkins, M.S., "Performance Evaluation of the Transmission Control Protocol in a Local Area Network Environment," *Canadian INFOR -- Special Issue on Performance Evaluation*, vol. 23(3), pp. 294-329, Aug. 1985a.
19. Chanson, S.T., Ravindran, K., and Atkins, M.S., "LNTP - An Efficient Transport Protocol for Local Area Networks," *UBC Computer Science Technical Report 85-4, University of British Columbia*, Feb. 1985b.
20. Cheriton, D.R., Malcom, M.A., Melen, L.S., and Sager, G.R., "Thoth, a portable real-time operating system.," *CACM*, vol. 22(2), pp. 105-115, Feb. 1979a.
21. Cheriton, D.R., "Multi-process Structuring and the Thoth Operating System," *UBC Computer Science Technical Report, University of British Columbia*, March 1979b.

22. Cheriton, D.R. and Steeves, P., "Zed Language Reference Manual," *UBC Computer Science Technical Report 79-2, University of British Columbia*, Sept. 1979c.

23. Cheriton, D.R., "Distributed I/O using an object-based protocol," *UBC Computer Science Technical Report 81-1, University of British Columbia*, Jan. 1981.

24. Cheriton, D.R., "The Thoth System: Multi-Process structuring and Portability," in *Elsevier North-Holland, New York*, 1982.

25. Cheriton, D.R., "Local Networking and Internetworking in the V-System," *Proceedings 8th Data Communications Conference*, Oct. 1983a.

26. Cheriton, D.R. and Zwaenpoel, W., "The Distributed V-kernel and its Performance for Diskless Workstations," *Proceedings of the 9th Symposium on Operating Systems Principles*, Oct 1983b.

27. Cheriton, D.R., "One-to-Many Interprocess Communication in the V-System," *Proceedings Fourth International Conference on Distributed Systems*, May 1984.

28. Clark, D., *Proceedings SIGCOMM 83*, 1983.

29. DARPA,, "Internet program protocol specifications -- Internet Protocol, Transmission Control Protocol," *Information Sciences Institute, USC, CA*, vol. RFC 791, 793, Sept. 1981a.

30. DEC,, "Digital Equipment Corporation - BLISS-11 Programmer's Manual," *Maynard, Mass.*, 1974.

31. Deering, S.E, "Multi-process structuring of X.25 software," *UBC Computer Science Technical Report 82-11, University of British Columbia*, Oct 1982.

32. Dijkstra, E.W., "Guarded Commands, nondeterminacy, and formal derivation of programs," *CACM*, vol. 18(8), August 1975.

33. Feldman, J.A., "High-level Programming for Distributed Computing," *CACM*, vol. 22(6), pp. 363-368, June 1979.

34. Gentleman, W.M., "Message passing between sequential processes: The Reply primitive and the administrator concept," *Software P&E*, vol. 11(5), May 1981.

35. Gentleman, W.M., "Using the Harmony Operating System," *National Research Council Canada, Division of Electrical Engineering, Ottawa, Ont.*, vol. NRCC/ERB-966, Dec. 1983.

36. Geschke, C.M. and Satterthwaite, E.H., "Exception Handling in Mesa," *XEROX PARC report, Palo Alto, 1977*, 1977.

37. Goldberg, A., "SMALLTALK-80 The Interactive Programming Environment," in *Addison-Wesley*, 1984.

38. Goodenough, J.B., "Exception-handling: Issues and a Proposed Notation," *CACM* , vol. 18(12), pp. 683-696, Dec. 1975.

39. Herbert, A.J., *CAP Operating System Manual*, University of Cambridge Computer Laboratory, 1978.

40. Hoare, C.A.R., "Monitors: An operating system structuring concept," *CACM*, vol. 17(10), pp. 549-557, Oct. 1974.

41. Hoare, C.A.R., "Communicating Sequential Processes," *CACM*, vol. 21(8), pp. 666-677, Aug. 1978.

42. IBM,, "PL/1(F)Language Reference Manual," *Form GC28-8201, IBM Corporation*, 1970.

43. Joy, W., *UNIX 4.2BSD Operating System*, 1983.

44. Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H., "On the Transfer of Conrol Between Contexts," *In Lecture Notes in Computer Science,vol.19,B.Robinet(ed.), Springer-Verlag,N.Y.*, pp. 181-203 , 1974.

45. Lampson, B.W. and Redell, D.D. , "Experience with Processes and Monitors in Mesa," *CACM 23(2)*, pp. 105-117, Feb. 1980.

46. Lantz, K.A., "Uniform Interfaces for Distributed Systems," *PhD Thesis, University of Rochester*, 1980.

47. Lauer, H.C. and Needham, R.M., "On the Duality of Operating System Structures," *Operating Systems Review*, vol. 13(2), pp. 3-19, April 1979.

48. Leach, P.A. and Levine, P.H., "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communications*, vol. SAC-1(5), pp. 842-856, Nov. 1983.

49. Lee, P.A., "Exception Handling in C Programs," *Software Practice & Experience*, vol. 13(5), pp. 389-405, 1983.

50. Levin, R., "Program Structures for Exceptional Condition Handling," *PhD Thesis, Carnegie-Mellon University*, 1977.

51. Levin, R., *Personal Communication*, May 1982.

52. Liskov, B., "CLU Reference Manual," *Computation Structures Group Memo 161, MIT Laboratory for Computer Science.*, July 1978.

53. Liskov, B. and Snyder, A., "Exception Handling in CLU," *IEEE Transactions on Software Engineering SE-5(6):546-558,November 1979*, 1979.

54. Liskov, B. and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, vol. 5(3), July 1983.

55. Lockhart, T.W., "The Design of a Verifiable Operating System Kernel," *UBC Computer Science Technical Report 79-15, University of British Columbia*, Nov. 1979.

56. Lomet, D.B., "Process structuring, synchronization and recovery using atomic transactions," *Proc. ACM Conf. Language Design for Reliable Software, SIGPLAN Notices*, vol. 12(3), pp. 128-137, March 1977.

57. Luckham, D.C. and Polak, W., "ADA Exception Handling: An Axiomatic Appoach," *ACM Transactions on Programming Languages and Systems*, vol. 2(2), pp. 225-233, April 1980.

58. Macintosh,, "Inside Macintosh," in *Apple Computers*, 1984.

59. Malcolm, M., Bonkowski, B., Stafford, G., and Didur, P., "The Waterloo Port Programming System," Technical Report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, January 1983.

60. Metcalfe, R.M. and Boggs, D.R., "Ethernet: distributed packet switching for local computer networks," *Communications of the ACM*, vol. 19(7), pp. 395-404, July 1976.

61. Mitchell, J.G., Maybury, W., and Sweet, R., "Mesa Language Manual, version 5.0," in *Rep. CSL-79-3, Xerox PARC*, April 1979.

62. Moss, J.E.B., "Nested Transactions: An APproach to Reliable Distributed Computing," in *PhD. Thesis, Masssachusetts Institute of Technology*, vol. MIT/LCS/TR-260, April 1981.

63. Mueller, E.T., Moore, J.D., and Popek, G.J., "A Nested Transaction Mechanism for LOCUS," *Proceedings of the 9th Symposium on Operating Systemas Principles*, Oct. 1983.

64. Nelson, B.J., "Remote Procedure Call," in *PhD. Thesis. Rep. CMU-CS-81-119, Department of Computer Science, CMU*, May 1981.

65. Ousterhout, J.K., Scelze, D.A., and Sindhu, P.S., "Medusa: an experiment in operating system design," *CACM*, vol. 23(2), pp. 92-105, Feb. 1980.

66. Parnas, D.L., "On a Buzzword: Hierarchical Structure," *Proc. IFIP Congress,North-Holland publ. Co.*, 1974.

67. Peterson, and Silberschatz,, "Operating Systems Concepts," *Prentice Hall*, 1983.

68. Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the 8th Symposium on Operating Systems Principles, ACM*, pp. 169-177, Dec. 1981.

69. Randell, B., "System structure for Fault Tolerance," *Proc. Int'l Conf. on Reliable Software, Los Angeles 1976*, 1975.

70. Randell, B., Lee, P.A., and Treleaven, P.C., "Reliability Issues in Computing Systems Design," *Computing Surveys*, vol. 10(2), pp. 123-165, June 1978.

71. Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., P.R.McJones,, Murray, H.G., and Purcell, S.C., "PILOT: an operating system for a personal computer," *Communications of the ACM*, vol. 23(2), pp. 81-92, Feb. 1980.

72. Reed, D.P., "Implementing Atomic ACtions on Decentralised Data," *ACM Trans. Computer Systems*, vol. 1(1), pp. 3-23, Feb. 1983.

73. Reid, L.G. and Karlton, P.L., "A File System Supporting Cooperation between Programs," *Proceedings of the 9th Symposium on Operating Systems Principles*, Oct. 1983.

74. Richards, M. and Whitby-Strevens, C., "BCPL -- The Language and its Compiler," in *Cambridge University Press*, 1979.

75. Richards, M., Aylward, A.R., Bond, P., Evans, R.D., and Knight, B.J., "TRIPOS -- a portable operating system for mini-computers," *Software P&E*, vol. 9, pp. 513-526, 1979.

76. Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System," *CACM* , vol. 17(7), pp. 365-375, July 1974.

77. Ritchie, D.M., S.C.Johnson,, Lesk, M.E., and Kernighan, B.W., "The C Programming Language," *The Bell System Technical Journal*, vol. 57(6), pp. 1991-2021, July-Aug. 1978.

78. Sammet, J.E., "Programming Languages: History and Fundamentals," *Prentice-Hall, Inc. N.J.*, 1969.

79. Scotton, G.R., "An experiment in High level protocol design," *University of British Columbia, Department of Computer Science, MSc. Thesis*, Dec. 1981.

80. Schlichting, R.D. and Schneider, F.B., "Using Message Passing for Distributed Programming: Proof Rules and Disciplines," *Carnegie-Mellon Technical Report*, vol. 82-491, May 1982.

81. Sindhu, P.S., "Distribution and Reliability in a Multiprocessor Operating System," *CMU Technical Report* , vol. CMU-CS-84-125, 1984.

82. Spector, A., "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM*, vol. 25(4), April 1982.

83. Defense, US Department of, *ADA Programming Language Military Standard*, MIL-STD-1815, Washington, D.C., Dec. 80.

84. vanWijngaarden, A., "Revised Report an the Algorithmic Language ALGOL 68," *Acta informatica*, vol. 5,Fasc.1-3, 1975.

85. Wilkes, M.V. and Needham, R.M., "The Campbridge CAP computer and its operating system," in *Elsevier North Holland*, 1979.

86. Wilkes, M.V. and Needham, R.M., "The Cambridge Model Distributed System," *Operating Systems Review*, vol. 14, pp. 21-29, 1980.

87. Wirth, N., "Programming in Modula-2," in *Springer-Verlag, New York*, 1982.

88. Wulf, W.A., London, R.L., and Shaw, M., "An Introduction to the Construction and Verification of Alphard Programs," *IEEE Transactions on Software Engineering*, vol. SE-2,4, pp. 253-265., Dec. 1976.

89. Zimmerman, H., "OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Commun.*, vol. COM-28, pp. 425-432, April 1980.

Appendix A

The putbyte program.

This appendix describes an example of minimising the average run-time by reducing the normal case detection costs in the **putbyte** utility program in the Verex i/o system [Cheriton 81]. This system provides, at the application level, a similar model of selecting input and output as is used in several implementations of BCPL [Richards 79a]. The Verex Z-language i/o library [Cheriton 79c] provides the routines **selectoutput** and **putbyte** to output a character. In Z, as in many C-like languages, the i/o library routines share global data. **Putbyte** and **selectoutput** share the global variable *selected-output*, from which the variables *selected-output.ptr* and *selected-output.bufcnt* are accessed. The routine **putbyte** may be conveniently be considered to have states *UNINITIALISED* and *RUNNING*. The events **putbyte** has to deal with are the request for initialisation, *initialise*, and the request to put a byte to the selected output, *putcharacter*. Initialisation must occur before any other calls to **putbyte** are accepted. The user must call **selectoutput** to cause initialisation of the global variable *selected-output* which influences the state of **putbyte.**[1] If the programmer tries to output a character when there is no selected output device, an exception occurs. This exception is a server-client type, which is detected by the server routine (in this case, **putbyte**), which may perform some handling before passing the exception on for handling by the client, **putbyte's** caller.

[1]In more modular languages, the global data and the routines manipulating it would be encapsulated in a module, and invocations of **selectoutput** and **putbyte** would be treated as request events upon which appropriate action must be taken: some events would cause state transitions and others would not. In the unstructured C-like languages, relevant global variables may similarly be treated as *state variables* and calls to routines which manipulate these, such as **selectoutput** and **putbyte** as events which cause state transitions.

Now the cost of handling a byte when the local buffer is full, is much more than when it is not full. Thus the execution cost of the routine depends on the buffer state as well as the event, so violating the assumption of constant-cost events. The event set is increased to reflect constant cost, by dividing **putbyte's** state of *RUNNING* into two new states, *BUFFER-FULL* and *BUFFER-NOT-FULL* so there is a constant cost of handling each event-state pair.

Thus there are 3 states, *UNINITIALISED, BUFFER-FULL, BUFFER-NOT-FULL*, and two external events, ***initialise*** and ***putcharacter***. By combining these into constant-cost state-event handling the events s_1–s_4 defined in Table A.1 below are obtained[2].

Table A.1 Execution costs for the putbyte routine

event	probability	cost of handling	cost of detection	expected execution cost	new detection cost	new expected execution cost
s_1= initialisation	0.00025	3000	0	0.75	0	0.75
s_2= putbyte-uninitialised	10^{-6}	1000	1	0.001	2	0.001
s_3= putbyte-with-buffer-full	0.001	1000	2	1.002	2	1.002
s_4= putbyte-with-buffer-not-full	0.999	2	2	3.996	1	2.997
Total expected costs/event				5.75		4.75

A regular implementation of **putbyte** might be of the following form:

The routine **error** handles the server-client exception *NO-SELECTED-OUTPUT* by printing a message and making a synchronous error return to the client. The routine **emptybuf** handles the peer exception *PUTBYTE-BUFFER-FULL* which is transparent to the user (except that the call to **putbyte** may take longer).

[2]For simplicity, the rare combinations of *initialise* with states *BUFFER-FULL* and *BUFFER-NOT-FULL* are ignored.

```
putbyte(ch)
 {
  if selected-output = NULL then error(NO-SELECTED-OUTPUT);
  selected-output.ptr++ = ch;
  if selected-output.bufcnt++ = 1000 then emptybuf();
 }

emptybuf()
 {
  write ( selected-output.buf, selected-output.bufcnt );
  selected-output.bufcnt = 0;
 }
```

This program is now analysed to see where its efficiency can be improved. To obtain the probabilities of each event, assume initialisation (i.e. a call of **selectoutput)** is performed just once for each set of data, and its cost is independent of the previous state of **putbyte.** If the average length of the data to be written is 4000 characters, then $p(s_1) = 0.00025$. Assuming that a request to putbyte with no selected output occurs very rarely, say 1 in 10^6 bytes written, and the buffer is 1000 characters, then $p(s_3) = 0.001$, so the remaining event, putbyte-with-buffer-not-full occurs with probability 0.999. Note that no kind of division is made here between the events.

Assuming that the routine **initialise** which handles the request exception call of **selectoutput** takes approximately 3000 instructions, the routine **emptybuf** takes 1000 instructions, and the routine **error** also takes 1000 instructions, and say just 2 instructions are needed to insert the byte into the buffer and update the counter for event s_4, then the handling costs for each event are as described in column 2 of Table A.1. Now the cost of detecting event s_1 in **putbyte** is zero, because initialisation is handled in a separate routine. Assuming each test costs one instruction, the detection costs are as shown in column 3 of Table A.1. Hence the expected average run-time cost = 5.75 instructions/event.

To minimise the run-time cost, the detection cost of the commonest event, s_4 should be reduced to just one test. Unfortunately this event cannot be tested explicitly; it is assumed to be none of the others. The approach is to follow the general principle of mapping several exception events onto one to reduce detection costs. Here, performance can be improved by mapping s_2 and s_3 together (as the initialisation exception s_1 is already separated out through a call to a separate routine).

The detection of the server-client exception *PUTBYTE-UNINITIALISED* can be mapped onto the detection of peer exception *BUFFER-FULL* by arranging that the uninitialised global variable ***selected-output*** points to a dummy file which has a full buffer. The two exceptional events s_2 and s_3 are then detected in just one test. Individual exceptions s_2 and s_3 are separated in the handler **emptybuf.** An appropriately modified program reads as follows:

```
putbyte(ch)
 {
  if selected-output.bufcnt++ < 1000 then selected-output.ptr++ = ch;
      else emptybuf(ch);
 }

emptybuf(ch)
 {
  if selected-output = dummy-file then error(NO-SELECTED-OUTPUT);
  selected-output.ptr = ch;
  write ( selected-output.buf, selected-output.bufcnt );
  selected-output.bufcnt = 0;
  selected-output.ptr    = selected-output.buf;
 }
```

The new program's detection and execution costs are given in the last two columns of Table A.1. The average run-time cost is reduced by 1/5.75 = 17.4%; achieved by dividing the events into two sets - $\{s_4\}$ corresponding to a *normal event* and $\{s_2, s_3\}$, corresponding to *exceptional events*, and by modifying the program so that the normal event is detected in just

one test.

Appendix B

The os program.

This appendix describes an example of minimising the average runtime by restructuring a utility program, **os,** to reflect the statistically dominant case rather than the logical flow. **Os** converts a formatted text file containing backspace characters for underlining and boldfacing, to a file achieving the same printed effect using overprinting of whole lines, and containing no backspaces. This data-driven utility can be treated as an event-driven program, where the input value of each character read represents an event.

In its original structure, the program uses multiple line buffers, one for each level of overprinting on the line. Each character except backspace and line terminators are placed in the highest level line buffer that is currently unoccupied in the current character position. Thus, its structure reflects the logical operation it is performing, translating a character stream into overprinting line buffers. However, its structure also means that the character it processed with the least overhead is the backspace character. Given that backspaces constitute about 0.5% of the characters in most text files, this program is structured inefficiently.

An initial version of the program is given in Figure B.1.

Using the same approach as in the **putbyte** example, the program is divided into several state-event pairs, each with constant cost over execution of the event. The main external event is reading of the next character from the text file. With the program structure described above, the cost of the event depends on what the character is -- *NEWLINE, BACKSPACE, END-OF-FILE,* and all the others. If the program has one state, *RUNNING,* the constant cost event-state combinations for analysis are as shown in Table B.1 below.

```
while ( byte ≠ END-OF-FILE )
 begin
   read( byte );
   if byte = NEWLINE then writeline( )[3];
   else if byte = BACKSPACE then cc--;
   else
   begin
     for (i=0; i< MAXOVERPRINTS; ++i )
     begin
       line = Lineptr[i];
       if line[cc] = BLANK then
                          begin
                            line[cc] = byte;
                            cc++;
                            Lastx[i] = max{Lastx[i],cc};
                            break;
                          end;
     end;
   end;
 end;

writeline( )
 begin
    for (i=0; i < MAXOVERPRINTS; ++i )
    begin
     last = Lastx[i];
     if last = 0 then break;
     line = Lineptr[i];
     for (j=0; j<last; ++j )
     begin
       put ( line[j] );
       line[j] = BLANK;
     end;
     Lastx[i] = 0;
   end;
   put(NEWLINE);
  cc = 0;
 end;
```

Figure B.1 The Initial Version of Os.

[3]Note that writeline() would be expanded as an in-line macro

Table B.1 Execution costs for the os utility

event	probability	cost of handling	cost of detection	expected execution cost	new handling cost	new detection cost	new expected execution cost
BACKSPACE	0.005	1	3	0.02	50W+160	2	.25W+0.8
NEWLINE	0.01	125W+382	2	1.25W+3.84	W+1	2	0.01W+0.03
END-OF-FILE	10^{-5}	0	1	10^{-5}	0	1	10^{-5}
REMAINING CHARS	0.985	7	3	9.85	W+1	2	.985W+2.955
Total expected costs/event				1.25W+13.71			1.24W+3.785

Assuming *BACKSPACE* occurs say 1 in 200 characters, $p(INITIALISE) = .005$; similarly, if the average length of the line is 100 characters, then $p(NEWLINE) = 0.01$. Assuming the average text file is 10000 characters, then $p(END\text{-}OF\text{-}FILE) = 10^{-5}$.

Furthermore, it is assumed the system function **read(byte)** takes R instructions, which are required to handle every event. Now the cost of handling *NEWLINE* depends on how many backspace characters there are in that line, so either more events should be generated to represent this, or the average handling costs over all lines should be taken. The latter approach is used here, noting that there is approximately one backspace character on every other line. The average handling cost per newline character is the average of a line without a backspace and a line with a backspace character half way along. This costs approximately ((100(W+3)+5) + (150(W+3)+10))/2 = 125W + 382, where the system function put(byte) takes W instructions, and the cost of executing a for-loop is 2 instructions per iteration[4]. From Table B.1 the expected cost of execution of the initial version = (1.25W + R + 13.71)

[4]Note that more characters are written than read, because of the extra blank characters in the overstrike buffers.

instructions/event.[5]

This program does NOT reflect any structuring for the statistically dominant case - indeed it processes one of the least likely bytes (BACKSPACE) most efficiently. The program can be rewritten to reflect the statistically dominant case of normal characters by writing it as a null filter copying its input to its output unchanged. Starting with this view, the exceptions to recognize are *END-OF-FILE*, a *server-client* exception, and a *BACKSPACE*, a *peer exception* transparent to the user. This leads to the following program structure:

```
while ( byte ≠ END-OF-FILE )
 begin
  read( byte );
  if byte = BACKSPACE then HandleException();
  else put( byte );
 end;
```

In the **HandleException** routine, characters are read and stored till the next newline into line buffers for overprinting, similar to the initial program which used buffers for *all* lines. This normal-case processing is not sufficient as it stands to handle the *BACKSPACE* exception because it is necessary to know the current character position on the line when a backspace is encountered. The final program is structured as above with this additional context-saving code, which takes the form of the insertion of the following lines in the **else** clause above:

```
if byte = NEWLINE then col = 0;
else col++;
```

This program was measured as twice the speed of the original on most text files, and, as Cheriton pointed out, this is contrary to the expectation that the gain would be small because the fixed file access overhead would dominate.

---

[5]for simplicity the constant cost R for each event is excluded from the Table

An analysis of this program reveals that the new execution cost is composed of three parts: the handling cost, the detection costs and the context-saving cost (= 1 instruction). These are shown in Table B.1, where the context-saving cost is merged with the handling cost. In this version the cost of handling the *BACKSPACE* exception in **HandleException** includes the cost of reading and storing characters till the next newline -- which means that the cost of handling an ordinary byte depends on whether a backspace has already occurred on a line, or not. To preserve comparison with the first **os** program, **all** the **extra** costs in processing the ordinary bytes on a line with a backspace, are ascribed to the BACKSPACE character. The cost of handling every ordinary character is then W + 1, and the cost of handling BACKSPACE is the extra cost in handling the line buffers. Assuming as before, that a BACKSPACE appears half way along a line, the expected cost of executing **HandleException** is the cost of reading and writing one line buffer of half the average length = 50(W+3) +10. Then the total expected cost of this version of the program is approximately (R+1.24W+3.8) instructions/event.

This final program fragment illustrates the use of a readback facility, as described in section 3.3.3.1, which would reduce context-saving costs in the processing of normal events, by enabling the last *NEWLINE* character to be located in the input document, thus obviating the need to preserve a column count.

```
while (byte != END-OF-FILE)
begin
  read(byte);
  if byte = BACKSPACE then HandleException();
  else put(byte);
end;

HandleException( byte )
begin
   col = 0;
```

```
    oldbyte = NULL;
    while oldbyte ≠ NEWLINE do
      begin
        readback( oldbyte );
        col++;
      end;
    newread(byte);
   <process-chars-as-before>;
end;
```

Appendix C.

The TROFF/TBL/EQN system.

In a Thoth-like operating system, a UNIX-like pipe system could be implemented using multiple processes, by using a **pipe worker process** to connect two systems processes, as shown below in Figure C.1.

The TBL, EQN and TROFF processes act like servers, waiting on a **Receive** statement for either data from the client, or for the WORKER-READY message from their pipe-worker. After processing a client request (usually a text line or less), if the worker is free, the server

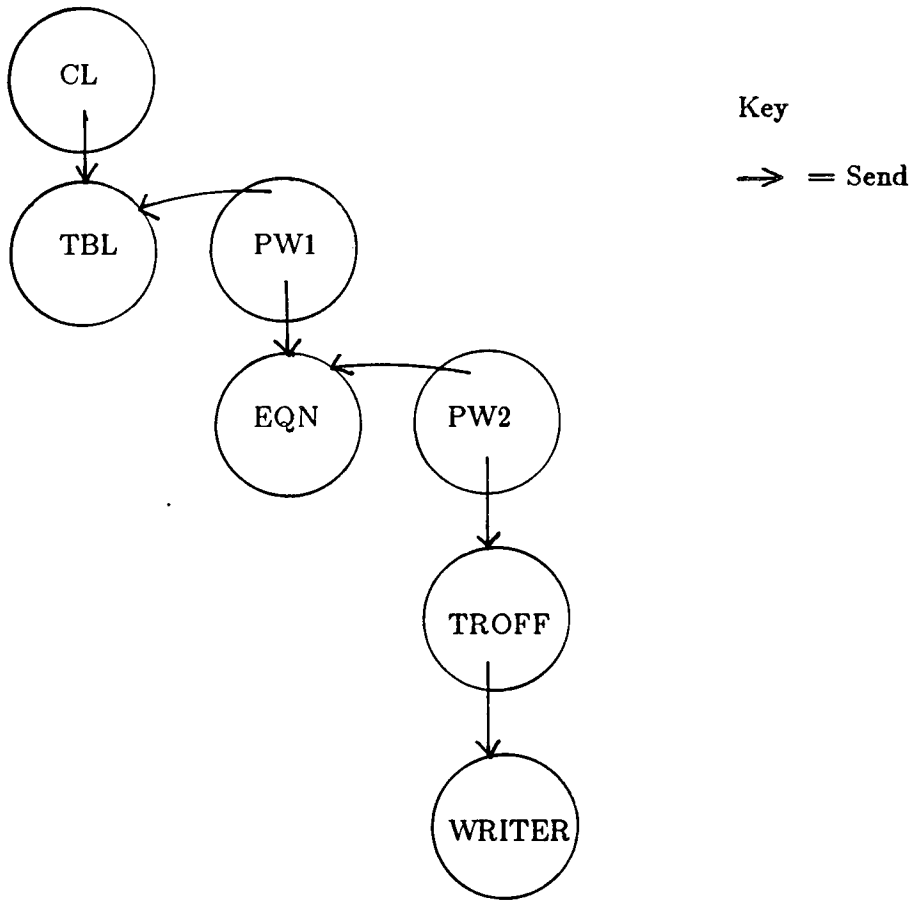


Figure C.1 Pipe Worker Processes

makes a **Reply** to the worker[6] before making a **Reply** to its client and completing its loop by awaiting another request. If a client request cannot be serviced immediately because the worker is busy, the server queues it (for later processing by the worker), and makes a **Reply** to its client, as before. The capacity of the queue can be set to the capacity of a UNIX pipe. If the queue becomes full, the server witholds the **Reply** to the client until the worker empties the queue. The worker processes the request by executing a **Send** to the server at the level below. When the worker becomes free it executes a **Send** to its master for more data.

Thus in the above system structured as a hierarchy of clients and servers, there are 6 x 2 = 12 process switches for each data item in the source file -- from CL to TBL to PW1 to EQN to PW2 to TROFF to WRITER -- and back again.

Suppose an average document has 5% of its text as tables to be processed by TBL, 5% as equations to be processed by EQN, and 90% as normal text to be formatted only by TROFF. Then this system structure does not reflect the statistically dominant case.

The system can be restructured so the inter-process communication is minimised for the statistically dominant case by treating lines with equations or tables as exceptional. All lines are sent to TROFF first. The user's document is read by a client process, CL, which executes a **Send** to TROFF, for each unit of data. TROFF executes a **Receive** and processes normal text, then executes a **Send** to the level below (say to process WRITER) before executing a **Reply** to CL, which then proceeds synchronously with the next item of data from the user's document. Thus for each item of normal text there are 4 process switches -- from CL to TROFF to WRITER to TROFF to CL. This is illustrated below in Figure C.2.

---

[6]Care must be taken to ensure that the data is copied into a safe place before the server makes a **Reply** to its client.

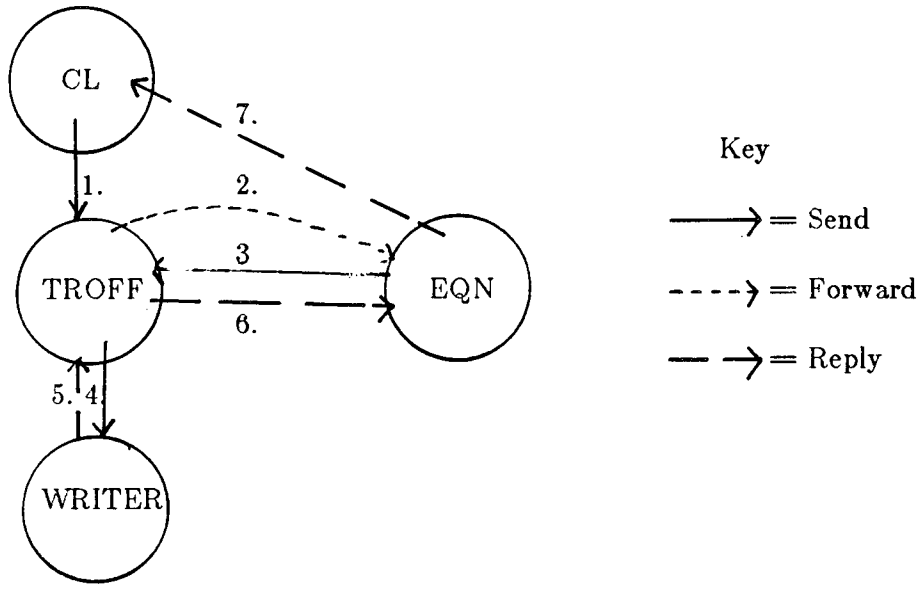


Figure C.2 Message Traffic for Handling an Equation

TROFF detects when a line is of the exceptional form EQN or TBL and handles it by forwarding to the appropriate server, where it is processed as before, and then re-sent to TROFF for further processing. Thus 7 process switches are incurred for equation text -- from CL to TROFF to EQN to TROFF to WRITER to TROFF to EQN to CL. TROFF forwards subsequent input to EQN, until EN occurs. A similar scheme can be used for TBL text and for mixed EQN and TBL text.

If an average document has 5% of its text as tables to be processed by TBL, 5% as equations to be processed by EQN, and 90% as normal text to be formatted only by TROFF, in a 1000-line document there are 4*900 + 7*50 + 7*50 = 4300 context switches, compared with 12*1000 = 12000 for the pipe system.