Exception Processing in Computer Systems

By

Bruce Gilbert Lindsay

A.B. (University of California) 1966
M.A. (University of California) 1971

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved:

Committee in Charge

.....

Exception Processing in Computer Systems

Bruce Gilbert Lindsay

Abstract

An exception is the reported failure of an operation to produce its specified results. An exception processing facility operating in an environment of cooperating, but mutually suspicious subsystems is developed. The exception facility is driven by exception signals from the failed operation and is responsible for selecting and activating a handler for the exception. It also manages the resumption of normal processing when the handler terminates.

The means used to specify exception handlers and the rule used to select one of the handler specifications are of critical importance in the design of the exception processing facility. In order to protect the subsystems of the computation and to avoid introducing dependencies between otherwise independent subsystems, the response to an exception should be controlled by handler specifications which are associated with the invoker of the failed operation. It is shown that other schemes for selecting the handler tend to break down the logical and physical isolation of the protected subsystems.

The exception facility supports programmer supplied local handler specifications, system supplied default handlers, and system imposed exception handlers. Default handlers supply a "standard" response when no other handler specifications are given. Imposed handlers allow the system to supply the response to system sensitive exceptions. It is shown how handler specifications can be associated with the subsystem during

the program preparation process. Installing handlers during program preparation and selecting a handler using only the invoker's handler specifications simplify program verification and comprehension because the response to an exception does not depend on other subsystems or on the dynamic state.

An implementation of the exception processing facility is presented. The implementation is partitioned into a set of low level operations for initiating and terminating exception episodes, and a subsystem which manages the selection and activation of the handler. An interesting aspect of the implementation is the ability of the exception processing subsystem to process its own exceptions using the same handler specification and selection protocols which are used for normal exceptions.

Acknowledgements

I would like to thank my thesis advisor, Professor R.S. Fabry, for providing advise and support during the preparation of this thesis. I am also indebted to the other members of my committee, Dr. James H. Morris and Professor Charles Prenner, for their critical reading and comments on the thesis.

Most of all I would like to thank Paul McJones who provided early interaction, valuable advice, and careful reading of early drafts of this thesis. I have also profited from discussions with Dave Redell, Bill O'Connell, Jim Gray, and John Barlycorn.

I would also like to thank the National Science Foundation for their support under Grant MCS75-23739 and IBM Corporation for its support during the final preparation of this thesis.

Table of Contents

Table of Contents

List of Figures

List of Figures

List of Figures

Chapter One

Introduction to Exception Processing

1.1 Introduction

The organization and construction of large programmed systems has been one of the continuing challenges faced by the computing community. Over the years, the availability of increasingly powerful processors, equipped with ever larger on-line stores, has been accompanied by demands for increasingly complex programmed systems to control and exploit these information processing resources. Considerable experience has been accrued in the construction of large programs such as operating systems, data-base management systems, and telecommunications applications. But, as Randell points out, we are willing "to design and implement systems of a level of complexity which challenges, and often defeats, our ability to comprehend them" [Randell 71]. Despite the enormous efforts which have been expended, large programs continue to be very difficult to produce and, once produced, they are unreliable and difficult to maintain or understand. The purpose of this thesis is to investigate and develop system level facilities for responding to and recovering from the detected failure of program or system components.

The "programming problem" has been attacked on many fronts. High level programming languages allow the specification of a computation to be encoded using a notation which has been adapted to the application area of the computation. The notation produces useful redundancy while

suppressing irrelevant details of the computation. Recently, so called "structured programming" techniques [Dijkstra 76] have been advanced as aids to understanding the program and the computation which it evokes. One structured programming technique is to restrict the mechanisms available for expressing the sequencing of the computation so as to make the lexical structure of the program reflect the sequencing structure of the corresponding computation. The structured programming discipline tends to reduce the complexity of the sequencing structure of a program by restricting the repertoire of sequencing constructs which can be used in the program.

Programming languages and structured programming techniques are concerned with the notation for expressing the computation. Other approaches to the programming problem endeavor to characterize the meaning of the notation. In conjunction with research into the formal semantics of programming languages [Hoare 73], program verification methods are being developed [Wulf 76, Ambler 77, Euclid 77]. The goal of these approaches is to prove formally that the program evokes the desired computation. The techniques and programming habits necessary to produce certifiably correct programs are not fully developed. The possibility of knowing that a program is correct is the chimera that motivates this line of research.

Instead of considering issues of representation and meaning in programs, one can study production management techniques. These techniques attack the traditional program production cycle of specify, code, debug, and maintain at one or more of the production stages. Design specification and documentation procedures range from the ad hoc [Boehm 75] to

more rigorous specification techniques [Parnas 72, Robinson 75]. The programming team has been studied [Baker 72, Brooks 75], and even the special psychological needs of programmers have been analysed [Miller 73, Weinberg 71]. Program testing methodologies continue to be advanced [Goodenough 75b, Panzl 76], despite the fact that "program testing can ... show the presence of bugs, but it is hopelessly inadequate for showing their absence." [Dijkstra 72]

1.1.1 Abstraction and Modularity

Another avenue of attack on the programming problem, and the one that we shall be following, is the pursuit of modularity in the program specification and implementation. Modularity is manifested through the use of procedural and data abstraction mechanisms. Procedural abstraction is implemented by providing a primitive operation which evokes a separately specified computation sequence. The "call" operation in conjunction with the "return" operation permits the specification of composite operations which can be invoked as if they were primitive operations.

> "We should recognize the closed subroutine as one of the greatest software inventions ... because it caters for the implementation of one of our basic patterns of abstraction." [Dijkstra 72]

Procedural abstraction mechanisms permit the description of a composite operation to be separated from the description of how it is employed. This organization allows us, in principle, to separate reasoning about the implementation of a procedure from reasoning about the use of the procedure.

If the atomic operations within one procedure interact freely with operations in another procedure, we can no longer separate the reasoning about the two procedures. The most common form of interaction is thru the use of shared data. If access to particular data structures is limited to a few procedures, then reasoning about the state and the well-formedness of these data structures can be restricted to reasoning about the procedures which directly manipulate the data structures. In order to achieve modularity, one must not only partition the program specification (into procedures), but also restrict the interactions among different procedures by partitioning direct access to data structures.

Data abstraction is achieved by grouping together the procedures which maintain the data structures representing the states of abstract objects. The procedures and data structures defining a data abstraction are combined to form what is called a subsystem, cluster [CLU 75] or class [Dahl 70]. In order to restrict interactions among subsystems, the execution environment of a subsystem must be isolated from other execution environments. In order to control the passage from one environment to another, a guaranteed interface between subsystems is necessary. The guaranteed interface enforces changes to the data accessing environment in response to the transfer of control from a procedure in one subsystem to the beginning of procedure in another subsystem and in response to the return of control to the calling procedure. Enforced isolation permits us to reason about local program issues without considering the actions of programs in other subsystems. The guaranteed subsystem interface defines the limits within which we must reason about the actions of programs in other subsystems.

The partitioning of large computations into interacting subsystems is not a new idea. However, the enforcement of isolation and the provision of a guaranteed interface between subsystems has received attention only relatively recently [Schroeder 72, Redell 74, Jones 73, Walker 73]. Efforts on this front lie in programming language developments (e.g. CLU, Alphard, and Euclid) and in operating system and machine architecture studies (e.g. CAL, HYDRA, CAP, and Plessey).

Programming language advances, such as Simula [Dahl 70], CLU [CLU 75], and Euclid [Euclid 77] have illuminated isolation and interface issues at the programming language level. Operating systems and machine architecture studies approach the partition of large programs at a lower level of system organization. At the system level, the interactions among arbitrary machine language programs must be controlled. Enforcing the isolation of programs belonging to different subsystems while, at the same time, permitting controlled interactions between subsystems is a protection and architectural problem. It is at the architectural level that we shall be discussing isolation and interactions among subsystems.

1.1.2 Introduction to Exceptions

With suitable isolation and interface mechanisms, the distinction between primitive and non-primitive operations (or subsystem calls) can be minimized. In either case, the invoking program selects the operation, designates the parameters, and accepts the results produced by the operation. The computation specified by a program is elaborated by executing a sequence of primitive and non-primitive operations which update

the state of the system. As long as each operation performs as advertised, the computation can proceed smoothly towards completion. Occasionally, an operation may fail or be unable to produce its specified results and effects. If some operation fails to perform as advertised, the execution sequence must reflect the operation failure. The reported occurrence of an operation failure is called an exception. The reaction of the computation to the occurrence of an exception is called the handling of the exception. The failure of an operation to compute its specified result should not necessarily ruin the larger computation which called for the failed operation to be executed. Many exceptions can occur quite legitimately and ought to be handled gracefully.

Exception handling has long been one of the gray areas of programming in which ad hoc and special case techniques have been applied to deal with specific run time failures. Recently there have been efforts to design uniform exception handling mechanisms at the programming language level [Goodenough 75, CLU 75]. At the system level, however, there have been few serious efforts to design mechanisms and policies for controlling the response to run time exceptions and for recovering from the effects of operation failures.

In practice, the subsystem implementor must specify the response to run time exceptions. The programmer should assume that every operation may fail. Such a defensive programming style can be encouraged by supplying convenient language and system level facilities for specifying the response to an exception. By assuming that every operation may fail, one is forced to consider how to proceed in case of the failure. Typical responses to reported failures include attempting alternate com-

putations, proceeding to the next task, or reporting failure to the calling level of the system. A uniform, system level exception processing facility can help the programmer to think about exceptions and provides a framework within which it is fairly easy to specify the response to an exception.

The variety of exceptional events which may occur in sophisticated systems composed of interdependent and interacting subsystems requires flexibility in the response to operation failures. Typical exceptions range from such primitive operation failures as page faults (an addressing failure) or arithmetic overflow to subsystem operation failures caused by improper parameters or resource limitations. A uniform exception reporting mechanism, coupled with sophisticated exception processing policies, can be used to initiate and control the response to the various kinds of run time failures.

In an environment composed of interacting subsystems, the exception processing mechanisms and policies must protect the integrity of the individual subsystem. Since subsystem boundaries may be crossed in dealing with an exception, exception processing will interact with the guaranteed isolation between subsystems. Protection problems involving the relationships between the subsystems which cause, detect, and process an exception must be resolved in the design of exception processing mechanisms and policies.

The failure of an operation, whether it is a primitive, machine level operation or a composite operation initiated by a subsystem call, can often be overcome by appropriate recovery and retry sequences. A uniform exception processing mechanism permits the treatment of hardware

and software detected exceptions to be placed on the same footing. Various exception processing policies can be exploited to tailor the response to the requirements of the program which invoked the failed operation. Often the invoking program can specify alternative actions which can be taken to recover from or minimize the deleterious effects of the failure. For example, the failure of an attempted write at the beginning of a non-existent file might be resolved by creating the file and then retrying the failed write operation.

Examples of exception processing applications include a number of facilities traditionally integrated into the system kernel. Many of the supervisory functions provided by the operating system are initiated by the failure of an attempted operation. Virtual memory systems are driven primarily by addressing failures (page and segment faults), as are dynamic linking facilities. Virtual machine monitors [Popek 74, Goldberg 73] respond to the failure of "sensitive" instructions in order to simulate their effect on the state of the virtual machine.

The overall control of a computation composed of a society of interacting subsystems is another supervisory function which can make use of exception processing facilities to obtain control and enforce compliance with "high level" user and system decisions. Controlling a set of mutually suspicious subsystems presents interesting and difficult problems in which the interests of individual subsystems must be protected at the same time as the subsystem is being forced to comply with global decisions made by the system or by the owner of the computation [Needham 71].

Exception processing protocols can also be exploited to implement extensions to low level system facilities. This application is characterized by situations in which the lower level supports a restricted domain of operands or does not provide recovery procedures for unusual situations [Parnas 76]. When the lower level fails, exception processing at higher levels may be able to correct the cause of the failure or to simulate the effects of the failed operation. In this way, higher levels can effectively extend the domain or the functionality of a lower level. The occurrence of low level failures and the recovery by the higher level can be made transparent to the user thru the use of appropriate exception handling policies.

1.1.3 Thesis Plan

The goal of this research is to investigate exception processing policies and mechanisms. We develop a uniform exception processing mechanism for controlling the response to exceptions at all levels of the system. The primitive exception processing mechanisms are extended by exception processing policies which can deal with a variety of exceptional situations. In particular, policies for dealing with exceptions in a process composed of cooperating but mutually suspicious subsystems are developed and discussed.

In this chapter, we present an introduction to the topic of exception processing. A discussion of the causes of exceptions is followed by a walk thru of an exception episode. An exception episode includes exception detection, reporting, processing, and returning to the main line computation.

In the second chapter we discuss a broad range of issues related to exception processing. General issues associated with the organization of the exception episode, naming of exceptions, exception processing overhead, and the relationships among the subsystems participating in the exception episode are disposed of first. The issues associated with specifying the response to particular exceptions are handled before the interesting questions surrounding the design of the handler selection policy. The choice of a handler selection policy is the most critical issue addressed in Chapter Two. Chapter Two also deals with the problems of how to terminate an exception episode so as to reflect the outcome of the handler's recovery attempts.

In Chapter Three, a processor model supporting exception processing operations in an environment of protected, mutually suspicious subsystems is presented. The processor model illustrates how the base system can implement protected subsystem interfaces and how the basic exception processing actions can be decoupled from the policy decisions which select handlers and control the resumption of normal processing.

Chapter Four continues the implementation of an exception processing facility by describing how the policy module responsible for selecting and activating handlers and controlling the evolution of an exception episode can be implemented as an ordinary subsystem. An interesting aspect of the implementation of the exception processing policy module is that the programs of the exception processing subsystem are themselves subject to exceptions. Exceptions which occur while processing an exception can, for the most part, be taken in stride without getting involved with special protocols or unusual processing. The last

chapter of the thesis summarizes the key points discussed in the body and states the conclusions of the research.

1.2 Anatomy of an Exception

In this section we first discuss the relationship between programmed abstractions and isolated or protected subsystems. The remainder of this section is a discussion of the several phases of an exception episode. Exception causes as well as exception detection and reporting are discussed briefly. After introducing exception reporting and handling, we discuss the termination of an exception episode.

1.2.1 Abstractions and Subsystems

Recent developments in the art of system design have emphasized that complex systems should be designed and implemented as an interacting set of abstractions [Parnas 72, Wulf 76, Lampson 76, Dijkstra 68]. Each abstraction defines an object type (or types) in terms of a set of operations which may be performed on instances of objects of the appropriate type. Starting with a set of primitive abstractions, it is possible to construct a hierarchy of abstractions in which complex abstract objects are implemented in terms of previously defined objects. Among the advantages of this approach is the fact that the functionality at each level of abstraction can be described in such a way that the design can be understood and evaluated without considering the internal details of other levels of abstraction.

The use of abstractions permits the development of complex systems using a "building block" approach. Operating systems are a prime example of complex systems which can profit from a systematic decomposition and hierarchical implementation. When composing abstractions, it is assumed that the programs which make use of an abstraction do not need to take into account the internal mechanisms used to implement the operations of the abstraction. The complementary criterion, that the implementor of an abstraction need not make assumptions about the programs which may call upon his/her abstraction, implies that the correctness of the implementation (program text, data structures, etc.) will not depend on how the abstraction is used [Wulf 76, Lampson 69]. An organization in which implementation details are concealed from the user of an abstraction [Liskov 74, Parnas 72] facilitates the composition of independently implemented modules and conforms with the principles of programming generality [Dennis 68].

The specification of an abstraction describes what the abstract level does without detailing how it is accomplished [Parnas 72, Robinson 75]. The implementation of an abstraction will include programs and data structures designed to simulate the specified effects of each of the operations of the abstraction. The implementation of an abstraction is known, for our purposes, as a subsystem. Each of the operations of the abstraction is associated with a gate or entry point to the subsystem. We will often refer to subsystem gates as "operations of the subsystem" to emphasize their relationship to the abstract operations they simulate. The programs implementing an abstract operation may discover, during execution, that they cannot produce the specified effects of the operation being simulated. The reported failure of an operation

invocation to produce the specified effects of the operation is called an exception * . An important part of the specification of an abstraction is the list of the exceptions which might be reported by each of its abstract operations.

1.2.2 Exception Causes

Operation exceptions can occur for various reasons. One source of exceptions is the existence of partially defined operations. If the effect of an operation is undefined for some values of the input parameters or for some states of the data base associated with the subsystem or for some combinations of parameter values and data base states, inappropriate use of the operation would cause an exception. These exceptions are sometimes called domain failures [Goodenough 75].

Resource limitations can also lead to exceptions if the implementation does not have, or is unable to obtain, sufficient resources (memory, I/O devices, etc.) to achieve the effects of an operation. For example, arithmetic overflow can be thought of as occurring because the adder and accumulator are not big enough to perform the operation. Resource exceptions are particularly difficult to circumvent. Attempts to guarantee sufficient resource availability at all times for all subsystems leads to unwanted dependencies between otherwise unrelated parts of the system. Insufficient resources can effectively prevent the suc-

---

* While other terms, such as error, condition, or undesirable event [Parnas 76], have been used to describe run time failures, we prefer the more neutral term which reflects the, hopefully, infrequent occurrence of failures and the fact that failures can be anticipated and may eventually lead to desirable results.

cessful completion of many operations.

The implementation of an abstraction will normally make use of primitive and non-primitive operations provided by the base level (hardware) and by other subsystems. The failure of an operation used by a subsystem implementation may prevent the invoking subsystem from producing its specified effects. While the failure of an invoked operation does not necessarily imply that the invoking subsystem must fail, it is often the case that there is no reasonable recovery action which would enable the calling subsystem to produce its specified effects and results. The propagation of failures from lower levels of the hierarchy of subsystems/abstractions is a frequent cause of failures at higher levels of the system. Of course, if the system is to be reliable, the buck must stop somewhere.

Several authors [Zilles 74, Parnas 76, CLU 75] have distinguished between exceptions caused by anticipated anomalies and exceptions resulting from unforeseen developments. The latter form of exception is sometimes termed a "failure of mechanism" and usually manifests itself by causing the system to enter an inconsistent or "impossible" state. Programming mistakes (bugs), hardware failures, or corrupted data can cause failures of mechanism. Like other exceptions, failures of mechanism can be detected and, sometimes, corrected.

1.2.3 An Exception Episode

Regardless of its cause, an exception episode can be broken into several phases. Exception processing begins with the detection of a

condition which prevents the successful completion of an operation invocation. Once the operation failure has been discovered, its occurrence must be reported to the rest of the system. Only after the exception has been announced to the exception processing facility can the proper response be initiated. The exception facility selects and initiates a handler for the exception. The handler initiated by the exception processing facility supplies the response to the exception. The exception handler will attempt to recover from the operation failure. When recovery actions have been completed, the exception episode must be terminated. Depending on the outcome of the recovery actions of the handler, different continuations of the suspended main line computation will be in order. Exception detection, reporting, handling, and the termination of the exception episode comprise the standard exception processing scenario. Of course, further exceptions may be reported during an exception episode. An exception during exception processing causes a new exception episode to be initiated. When the second level exception episode terminates, the first exception episode can be continued.

Three subsystems are involved in an exception episode. The subsystem invoking the operation which detects and reports the exception is known as the invoker. The invoker and the invoker's environment will determine how the exception is to be processed. The subsystem which detects and reports the exception is called the signaller. For primitive operations, the signaller is the hardware or the kernel. The subsystem which is called to deal with the exception is the handler of the exception. In general the invoker and the signaller will be different subsystems. The handler may or may not be part of the invoking

subsystem. The relationships between these three subsystems play important roles in the processing of an exception.

1.2.4 Failure Detection

The timely detection of operation failures is extremely important. "We know that the only way to avoid error is to detect it, that the only way to detect it is to ... inquire." * Also, "the aim should be that all components have a reliable mechanism for error detection, if not for error recovery." + The discovery of exceptions during execution implies that the failure detection algorithms must be invoked at the appropriate moments. Many failures are easily detected by checks embedded in the subsystem program. Range and consistency checks on the parameters of a subsystem invocation can be used to detect improper use of the subsystem. The compatibility between a contemplated action and the current state of the subsystem's data base can also be checked dynamically.

One of the most generally applicable mechanisms for detecting exceptions is the use of redundancy. Error detection can be based on redundancy and consistency checks within the data structures maintained by the subsystem. Internal check sums, "checkable" pointers, and range checks are among the techniques which use redundant information to verify the correctness of stored data. Data structures designed to make use of consistency controls which can detect and recover from "impossible" states are called robust data structures [Lampson 74]. Dynamic

---

* J. Robert Oppenhimer, quoted in Cities in Flight by J. Blish.

+ Randell 71, p. 107.

verification of data consistency can not only detect errors, but also limit their propagation once they have occured.

Another failure detection technique is based on the independent verification of the results of the subsystem operation [Fabry 73, Horning 74]. Instead of integrating the checking code into the subsystem, the operation results are checked by independent algorithms (a form of algorithmic redundancy). These verification algorithms can recalculate the results using different methods or by checking that the results meet certain specifications (e.g. SQRT(x) * SQRT(x) = x $\pm$ epsilon). One problem with relying solely on this approach is that many valuable consistency checks are closely related to the function being performed. They are often heuristic tests which are intimately associated with the logical and physical structure of the data and algorithms used to implement the operation. Outside of the execution environment of the subsystem activation, information about the initial and intermediate states of the subsystem and its data structures is not available to the error detection algorithms.

Execution time parameter checking can be deferred under certain circumstances. If a parameter to one subsystem is to be used by that subsystem as a parameter to another subsystem, the parameter checking can often be left to the second subsystem. Of course, the first subsystem must be prepared to field the exception signalled by the second subsystem if the parameter is rejected by the second subsystem. The trade off between redundant parameter checking and the added complexity involved in recovering from the parameter exception signalled by the second subsystem must be considered in the design of the exception

detection algorithms of the subsystem.

The importance of timely and effective failure detection mechanisms stems from the fact that no response to the exception can be undertaken before it is discovered. Early detection prevents the propagation of the failure to other parts of the system and pinpoints the cause of the exception. The subsystem designer should take into account the means by which satisfactory subsystem operation can be verified and unsatisfactory behavior detected.

1.2.5 Exception Reporting

Once it has been discovered that a subsystem operation cannot be completed, the failure of the operation must be communicated to the exception processing facility. Note that if the problem can be corrected by the subsystem which detects the anomaly, the subsystem operation has not failed and there is no need to signal an exception. Sometimes an impending failure can be circumvented locally by forcing parameters to acceptable values, by repairing "robust" data structures, or by applying alternate algorithms to achieve the effects of the operation. If, on the other hand, a locally uncorrectable error has been detected, the attempted operation must be abandoned. Before relinquishing control to the exception processing facility, the signaller should return its data structures to a consistent state.

Ideally, a failed operation should have no (observable) effects [Parnas 72]. Conceptually, one can imagine an infallible oracle which is consulted before the operation is undertaken to find out whether the

operation would fail for any reason * . If the operation would not fail, execution commences and completes successfully (of course). Otherwise, the operation is not attempted and the failure is signalled to the exception processing facility. It appears to the invoker of the failed operation that the operation was not invoked.

In order to eliminate the side effects of failed operations, the data structures maintained by the subsystem should be restored to their state before the invocation of the subsystem. Restoring the state may be difficult if the exception is detected after the subsystem's data structures have been updated. Not only must local data structures be restored, but changes made by sub-operations called before the exception was detected must also be undone. For example, files opened by the failed operation should be closed before the exception is signalled to the exception processing facility.

In complex situations a recovery log can be maintained to record the sequence of actions to be undone in order to reverse the effects of subsystem execution. A checkpointing facility, such as the recursive cache associated with the recovery block scheme [Horning 74], can be employed to restore the state of the subsystem following the detection of an exception. If messages to the outside world have been sent or accepted (e.g. fire the rocket), some form of "compensation" [Bjork 72] should be undertaken to nullify or mitigate their effect.

---

* This model for explaining the detection and reporting of exceptions is due to Paul McJones who also implemented a micro coded APL interpreter which conformed to the model [McJones 73].

Although it is less desirable than undoing all the effects of a failed operation, it will be necessary, on occasion, to retreat from the no effects policy. It is sometimes possible to define intermediate states of an operation which may be retained after a failure is detected. If the failed subsystem has reached a state which could legitimately occur in response to operations of the subsystem, the subsystem can be said to be in an interface consistent state. An interface consistent state is one which could be observed, under normal circumstances, at the subsystem interface.

If an operation is left partially completed, the invoker of the operation should be provided with information which indicates how much of the operation has been completed [Parnas 76]. This can be achieved either by providing operations which return the necessary information or by updating one of the operation parameters which serves as a progress indicator. The Move Character Long (MVCL) instruction of the IBM 370 system [IBM 370] is an example of how partially executed operations can be terminated and continued. A partially executed MVCL instruction cannot be undone because of the destruction of the target string or re-executed because of possible overwriting of the source string. However, the parameters to the MVCL (length, and string pointers) are updated by the operation to indicate how many characters have been copied. This permits the invoker to continue the operation after it has been interrupted by, say, a page fault.

After restoring the subsystem state following the detection of an exception, the exception is signalled to the exception processing facility. The signal operation terminates the activation of the signaller

and transfers control to the exception processor. The exception processor must select and invoke a handler to respond to the signalled exception. The selection of a handler for the exception can be based on the nature of the exception. To assist in the selection of a handler, the signaller should supply an exception name to identify the exception being signalled.

1.2.6 Exception Handling

Once the signaller has notified the exception processing facility, its participation in the exception episode is over. The subsystem operation selected by the exception processing facility to handle the exception is invoked by the exception processor. The rules used to choose the handler comprise the handler selection policy. Various policies will be discussed in Chapter Two. The handler may be part of the invoking subsystem, or it may be in a different subsystem.

The handler may be able to recover from the exception by simulating the effects of the failed operation using alternate algorithms designed to circumvent the problems encountered by the signalling subsystem. For example, the failure of an in-core sorting operation might be overcome by resorting to a sort-merge which uses secondary storage. Returning the largest representable number might be an acceptable simulation of an arithmetic operation which has signalled overflow. The Newcastle group [Horning 74, Anderson 76] is investigating this form of recovery in connection with their Recovery Block approach to exception processing.

Instead of simulating the effects of the failed operation, the handler may be able to correct the cause of the exception. There are many applications for this kind of recovery. Fetching the missing page into main memory will alleviate the cause of a page fault. The failure of an attempted write to the beginning of a non-existent file can be overcome by creating the file. Segment activation and initiation in Multics [Bensoussan 72] correct the cause of segment faults in that system. The copy-on-write rule in TENEX [Bobrow 72] is implemented by making a private copy of a shared page in response to the exception generated by an attempted update to the shared page.

It may be that the handler is unable to recover from the exception either by simulating the failed operation or by correcting the cause of the failure. This may mean that the invoker will be unable to produce the specified effects of the operation it was in the process of performing. If the handler determines that the invoker cannot complete its mission, an exception at the level of the invoker has been detected. The invoker's state must be restored and the exception signalled to the invoker's invoker. From the point of view of the invoker's invoker, it is the invoker which has failed. As the exception propagates from callee to caller, each level is given a chance to recover or to cleanup and signal the exception to its caller. Eventually, some subsystem will either be able to recover from the exception or, more likely, abandon the current task and proceed to some other task. For example, the highest level of a payroll program, when confronted with an exception caused by inconsistent data for one employee, could print an error report and then continue with the pay computation for the next employee.

1.2.7 Episode Termination

If the exception handler succeeds in recovering from the exception, either by simulating the effects of the failed operation or by correcting the cause of the failure, the invoker can be resumed. If the failed operation was simulated by the handler, the invoker can be continued as though nothing had happened. If the cause of the exception has been corrected, then the invoker can re-execute the failed operation.

The resumption of the invoker terminates the exception processing episode. The exception handler should return control to the exception facility with an indication of whether the instruction which led to the exception should be retried or should be considered completed. If the instruction in the invoker which led to the exception has been simulated by the handler, the handler may need to return the results of the operation to the invoker.

The handler may also report that it has not recovered from the exception. The handler may indicate that it was unable to recover but, the exception processor should to try to find other handlers for the same exception or for a different exception. In this case, the exception episode can continue with the activation of a new handler. The handler may also report that the exception has led to the failure of the invoker. In this case, the invoker will become the signaller of a new exception and a new exception processing episode will be initiated after the invoker has been terminated. Chapter Three discusses the mechanisms for terminating exception episodes while Chapters Two and Four discuss a variety of handler terminations and the exception handling policies they support.

1.2.8 Anatomy Summary

The processing of an exception proceeds in several phases and involves the subsystem which detects and reports the exception (the signaller), the subsystem which called the signaller (the invoker), and the subsystem which responds to the exception (the handler). Exception processing begins with the detection of the failure by the signaller. After restoring its state, the signaller reports the exception to the exception facility which selects and calls a handler for the exception. The choice of the handler is conditioned partially by the exception name provided by the signaller. Depending upon the outcome of the handler's recovery actions, the invoker can be continued or the exception can be propagated to the invoker's invoker. The overall control of exception processing is provided by the exception processor which selects handlers and manages the transfer of control to the handlers as well as the termination of the exception episode.

Chapter Two

Issues and Answers

2.1 Introduction

This chapter discusses a wide variety of issues related to exception processing. We begin with several general issues relating to the organization of the exception episode. These issues are concerned primarily with the beginning of an exception episode. The reporting and classification of exceptions, the relationships among the subsystems involved in an exception episode, and the the issues surrounding activation and communication with the handler are all discussed.

Section 2.3 isolates the issues surrounding the association of handlers with the programs on whose behalf they operate. Handler specifications must meet a variety of requirements in order to reflect and protect the interests of the programs to which they apply. Given a facility for specifying handlers, there must be a rule for selecting a particular handler specification from the set of applicable specifications. The handler choice rule which selects the handler to respond to an exception is of critical importance. In an environment supporting protected subsystems, the handler choice rule must not compromise the integrity of the subsystems involved in the exception episode.

Finally, this chapter discusses the various ways in which the handler should be able to control the continuation of the computation which encountered the exception. Different ways of terminating an

exception episode are needed to reflect the outcome of the handler's efforts to recover from the exception. Facilities for continuing or terminating the computation, along with ways of passing the buck to other handlers, allow the handler to indicate what should happen next.

2.2 Exception Episode Issues

This section discusses some of the differences and similarities between language level and system level exception processing and the separation of exception policy and mechanism. touched upon briefly. The issues surrounding exception reporting and the disposition of the signaller are examined along with the problems of exception processing overhead and efficiency. The classification of exceptions and communication with the handler are examined at the end of this section.

2.2.1 Language vs System Level Exception Processing

Exception processing can be discussed either in terms of linguistic constructs embedded in a programming language or in the context of the extended interface provided by the operating system kernel. While most of the issues raised when considering exception processing at the programming language level also apply to the processing of exceptions at the operating system level, the systematic treatment of exceptions at the system level imposes additional constraints on the design of the exception processing mechanisms and policies. These constraints stem from the fact that a general purpose operating system is charged with supervising the harmonious execution of independently generated machine

language representations of programs written by different people and translated by different compilers.

Protection is one important issue in system level exception handling which cannot be accomplished totally at the language level. Language level, compile time checking of the compatibility between program components is a useful tool for discovering inconsistencies in the program text. However, when programs which have been prepared by different language processors are to interact at run time, it is not always possible to verify that the interfaces between them are understood in the same way by all parties. This is especially the case in environments which support independent programming language systems and the dynamic binding of subsystem invocations to subsystem instances. If arbitrary machine level programs can be executed along with compiled high level language programs, interface compatibility cannot usually be verified prior to execution.

With respect to exception processing, the compatibility between exception signals in one subsystem and handler specifications in the invoking subsystem can be checked at compile time or dealt with during execution after a signal has been raised. Some modern programming language systems [CLU 75, Wulf 76] assume that the program modules to be combined are written in the same language and that the compiler can check the compatibility between different modules. Goodenough's proposals for exception processing [Goodenough 75] also require that the compiler be able to check that there is always a handler for a signalled exception. This sort of compile time checking is similar to checking the type compatibility of procedure parameters during compilation.

One problem with language level control of the compatibility between exception signals and handler specifications is that exception processing facilities outside the scope of the language cannot be included. Particularly, system supplied default handlers and handlers imposed by the system fall outside of the purview of the language processor. If programs prepared by uncertified translators are allowed, it may be impossible to tell which exceptions are signalled by these programs. The lack of global knowledge about which exceptions are possible makes it impossible to assure that signals and handlers are well matched.

At execution time, a system level exception facility should be able to find some way to continue. The ability of the system to do this in every case is required if the system is to allow reliable programs to be written. The show must go on because some subsystems may be in a state which requires their resumption to release resources and restore their data bases to a consistent state. Finding a reasonable continuation in the face of unanticipated exceptions, without compromising the integrity of the subsystems involved, is one of the problems which should be solved by the system exception processing facility.

At the level of the language processor, on the other hand, good design principles impose constraints on exception processing that would be unacceptable at the system level. It is common for the run time environment to deal with exceptions for page faults, hardware errors, virtual machine traps, resource exhaustion, and so on, and to shield the user from the complications of processing the exceptions. The result is typically a simplified set of exception processing facilities at the

language level which are designed for ease of correct use but which rule out certain techniques required for dealing with the more bizarre types of exceptions. While recognizing the importance of usable exception facilities at the language level, we focus instead on a more general facility which would form an appropriate base for implementing the simpler language level exception mechanisms.

Separately compiled, mutually suspicious subsystems require an exception processing facility which enforces orderly transfers of control in response to exceptions discovered by one subsystem, but handled by a different subsystem. The ability to combine mutually suspicious subsystems written in different programming languages implies that the management of subsystem interactions, including exception signals, must be implemented and enforced by operating system facilities which are independent of the language processors which help to prepare the executable machine level programs.

2.2.2 Policy/Mechanism Separation

In a hierarchically structured system, facilities implemented at lower levels of the system are used by higher levels. Intervening levels of the system may restrict or extend the ways in which the low level facilities are used by the higher levels. The intervening level is said to establish and enforce a policy restricting the manner in which the low level mechanisms are exercised. Of course, programs which use the policy controlled mechanism see the policy/mechanism combination as a facility or mechanism which can be further regulated. In a multi-level system, this organization can lead to a hierarchy of policies in which

one level's policy becomes the next level's mechanism. The multi-level design of the resource allocator in IBM's OS/VS2 [Lynch 74] is an example of a policy hierarchy, as is the memory management organization suggested by Dijkstra [Dijkstra 74].

The separation of the policy aspects of a facility from the use of the basic facility permits high level, decision making programs to be removed from the lower level. In the policy driven scheduling facility described by Bernstein and Sharp [Bernstein 71], the choice of a scheduling strategy for a given process is a policy decision made outside of the portion of the system responsible for short term processor scheduling. The dispatcher bases its scheduling decisions on a parameterized priority rule for each process. The policy level in that system is able to set the parameters which will cause the lower level to supply the appropriate class of service to each process. The thrashing control policy described by Shils [Shils 68] is another example of a policy which regulates an underlying mechanism.

Another reason for separating policy from mechanism is so that different policies can be defined in order to cater to different patterns of usage of the underlying facility or to tailor the facility to the needs of the user. Multiple linker policies described by Janson [Janson 74] permit the dynamic linking facility to be controlled differently in each domain of a Multics process. In HYDRA it is possible to define different scheduling policies for different classes of processes [Levin 75].

The means by which a policy program is able to enforce its decisions are usually related to the access control mechanisms of the

system. Privileges extended to the policy level permit it to exercise the lower level in ways denied to the user of the policy/mechanism combination. If some other subsystem could also exercise the policy prerogatives, it would be difficult to assure a consistent interface to the controlled mechanism. In order to support multiple policies controlling the same facility, the policy prerogatives must apply, not to the facility being controlled, but to instances of the objects supported by the facility. In HYDRA, for example, a scheduling policy module must have particular rights for the processes it schedules. [Levin 75].

The behavior of a policy controlled mechanism will depend on the policy currently being enforced. In many cases, the user of a policy/mechanism combination is not given any choice as to which policy is imposed upon the use of the underlying mechanism. For example, the system must often be able to select the policies which control the use of its resources and facilities. However, the user of a policy controlled mechanism may depend upon the behavior associated with a particular policy. If the choice of a policy is imposed on the user, the user must at least be able to inquire as to which policy is being imposed. If the imposed policy is not appropriate to the task at hand, the user's program can, at least, refuse run under that policy. Note that the mechanism by which the user learns the identity of the current policy must be implemented outside of the policy module. It must not be possible to lie to the user about which policy is in force as is the case for the JSYS trap mechanism in TENEX [Thomas 75].

Exception processing lends itself to a policy/mechanism decomposition. The mechanisms used to signal an exception, activate a handler,

and resume the main line can be separated from the policy which determines which handler is to respond to the exception. Signalling, activating the handler, and returning to the main line are subsystem transfer operations. The handler choice is a policy decision which can be implemented by a policy module. The policy module in this case is a distinguished subsystem -- the exception processor. The exception processor can be activated by the signal operation and will enjoy certain privileges with respect to the resumption of the main line and access to other environments. Different exception processors can control the processing of exceptions in different processes. Chapter Three discusses the implementation of the mechanisms for signalling exceptions and resuming the main computation. Chapter Four is concerned with the implementation of a particular exception processing policy.

Not only can the exception processing facility be given a policy/mechanism decomposition, but exception processing can also be exploited to help implement mechanism/policy separation for other system and user facilities. In order to implement a policy/mechanism separation, the policy module must receive control at the appropriate moments in order to enforce its control over the use of the mechanism. The exception processing facility can be exploited to transfer control to the appropriate policy module in response to exceptions signalled by the controlled mechanism. The exception processor and its handler selection rule can be relied upon to activate the appropriate policy module at decision points signalled by the mechanism level. For example, the selection of the name binding algorithm to be invoked in response to signalled linkage faults can be enforced by the exception processor [Janson 74].

2.2.3 Uniform Exception Reporting

In order to initiate exception processing following the detection of an exception, the signaller must somehow inform the rest of the system that it cannot complete the operation it was called upon to perform. In some systems the signaller may, depending on its own classification of the exception, initiate different sorts of exception processing. In CAL [Lampson 69, CAL 69, Sturgis 73], the signaller would execute a 'failure-return' if it thought that the operation could possibly be salvaged. If an error of usage was detected, the signaller performed a 'return-with-error.' In Multics, exceptions can be communicated by returning status codes to the caller or by using the Multics condition mechanism [MPM 75, Organick 72]. Similarly, in OS/VS2 exceptions are signalled either by returned status codes or by performing an ABEND [OS/VS2 75].

The problem with supporting multiple exception reporting modes is that the signaller must decide which mode is appropriate for the exception in hand. In general, the meaning of and response to the exception are defined by programs residing in different environments. The signaller is incompetent to make the decision as to which signalling mode is appropriate because the response to the exception depends upon why and for whom the failed operation was invoked. For example, the distinction between failures-of-mechanism or unanticipated failures, and specified or anticipated exceptions is not always clear cut [Melliar-Smith 77]. A uniform exception reporting facility permits the response to any reported failure to be handled without requiring the signaller to select one or another exception reporting protocol.

Uniform signalling or exception reporting should extend to hardware as well as software detected exceptions. Hardware detected exceptions should be processed in the same ways as software reported exceptions. This is especially true in systems in which the hardware/software boundary migrates on different models of the equipment (e.g. PDP 11 systems with or without floating point hardware and the IBM S/370 with or without VM assist). Many systems treat some or all hardware detected exceptions differently from software reported failures. The Multics [Organick 72] and IBM [OS/VS2 75] systems treat some hardware level exceptions specially. Some exceptions are processed using the system exception protocols, while others are given special treatment. The Cambridge system [CAP 76c] seems to provide a uniform exception reporting mechanism. The ability to turn hardware exceptions into exception processor calls is also available in Burroughs systems [Organick 71].

Decisions as to how an exception is to be processed should not be made by the signaller. A uniform exception signalling mechanism decouples exception reporting from exception processing. If all exceptions are reported in the same way, the response to any exception can be controlled by uniform mechanisms and policies.

2.2.4 Disposition of the Signaller

Following the detection of an exception, the signaller must report it to the system exception facility. What should happen to the subsystem activation of the signaller? The system response to the failure of an operation usually requires that the system backup to a consistent state from which alternative computations can be initiated to circumvent

the reported exception. If the signaller has determined that it cannot complete successfully, it should give up cleanly and raise the exception. A subsystem transfer primitive which raises an exception and terminates the signaller activation permits exception processing to proceed from a clean state.

The subsystem most directly affected by the failure of the signaller to perform as advertised is the signaller's caller. If the signaller cannot complete normally, it is the signaller's caller who must respond to the exception. Maintaining the activation of the signaller complicates the environment in which the invoker's handler must operate. To leave things in a clean state, the signaller should fold its tent neatly and leave the field. If the signaller could circumvent its own failure, then it should do so instead of reporting an exception.

If the signaller restores its state before reporting the exception, the handler can pretend that the failure was detected before the failed operation was initiated. This allows the handler to operate strictly on behalf of the invoker of the failed operation. If the signaller activation is not terminated in a clean state, the handler cannot ignore the signaller while it attempts to recover on behalf of the invoker. If the handler must be responsible to both the invoker and the signaller, protection problems are introduced and the recovery task becomes more complex.

Existing and proposed exception facilities dispose of the signaller in different ways. The recovery block scheme [Horning 74] is very careful to restore and terminate the signaller cleanly. PL/I, on the other hand, folds the signaller when the signaller is the system (e.g.

conversion, overflow, end-of-file), while maintaining the signaller and its state for software reported exceptions. Lampson's and Goodenough's proposals [Lampson 74b, Goodenough 75] opt for holding the signaller activation with controls on whether the signaller can be continued or must be folded later by a sort of multi-level exit from the handler.

One case in which the activation of the signaller might be preserved occurs when the signaller wishes to notify or seek advice from its caller. This application, discussed extensively by Goodenough [Goodenough 75], processes the exception on behalf of the signaller's caller (the invoker) so that the caller's handler can respond to the condition reported by the signaller. This use of the exception facility to provide a communication channel between a subsystem and its caller is orthogonal to the main purpose of the exception facility which is to provide notification of, and initiate recovery from failures of a system component to perform its specified task.

2.2.5 Exception Processing Overhead

An important consideration in the design of exception processing strategies is the main line overhead involved in preparing for eventual exceptions. One expects that many possible exceptions will seldom occur. The expected ratio of calls to exceptions is a good yardstick to apply to the design of the exception profile of subsystem operations. Simon's principle of complexity states that adjacent levels of a hierarchy should differ by an order of magnitude in the frequency of their activation [Simon 68]. If an operation exception occurs frequently, relative to the number of operation invocations, it should perhaps be recast

as a result or a returned status of the operation.

Exception processing overhead can be distributed between program preparation time (e.g. compilation and subsystem creation), run time (normal main line execution), and exception time. If exceptions are infrequent, any continuing run time overhead may be unacceptable. It is possible to exchange extra processing at program preparation time and at exception time for less run time overhead preparing for possible, but improbable exceptions. In most cases, the main line should not have to execute any extra instructions solely to test for, or prepare for exceptions which have not yet occurred [Lampson 74b].

Some exception processing strategies are ruled out by this overhead principle. The use of returned status codes in Multics [Organick 72, MPM 75] and IBM OS/VS2 [OS/VS2 75] produces main line overhead whether or not exceptions occur. The less expensive the operation, the greater becomes the relative cost of the status code checking. Returned status variables must be passed across all subsystem boundaries and must be checked on every call. Systems which rely on status variables to signal exceptions are filled with status variable manipulations and tests which are executed whether or not an exception occurs. The situation is not unlike the case for interrupts in the CPU; constant testing for some change of status can be replaced by facilities to automatically transfer control when the status changes.

Passing extra parameters to control exception processing [Parnas 72] also adds overhead in the normal case. To maintain uniform exception reporting, every operation would require the extra parameter(s). Setup calls, such as the PL/I ON statements [Nobel 68] and the OS/VS2

ESTAE and SPIE facilities [OS/VS2 75], add overhead by requiring the execution of run time code and calls to maintain and control the exception processing environment. If setup calls must be executed by every subsystem activation to control its exception processing environment, considerable overhead will be involved if subsystem calls are frequent. The condition mechanism in Multics requires that the procedure entry prologue update and thread "condition blocks" in the stack frame if the procedure is to enjoy its own exception processing environment * [Organick 72]. The CAL system [Sturgis 73, CAL 69] did not suffer from main line exception processing overhead. Lampson's and Goodenough's proposals [Lampson 74b, Goodenough 75] avoid main line overhead by compiling information to control the handler choice at run time.

2.2.6 Exception Naming

Once an exception has been reported, the exception processing facility must select a handler for the exception. Unless the exception is classified in some way, it is impossible for the exception facility to provide an exception specific response. The exception name identifies the exception so that the exception facility can select an appropriate handler. Several issues are raised when we consider how an exception ought to be identified.

To begin with, it must be possible to add new exceptions to the system. An expandable exception set permits the definition of new exceptions to characterize the failure modes of newly implemented

---

* This may be a case in which the sins of the system programming language have been visited on the operating system.

subsystems. Many systems severely limit the number of distinguishable exceptions. In IBM's OS/VS2, the SPIE exception handling facility permits only fifteen different exceptions to be recognized while the ESTAE form of exception handling supplies a uniform response to all, or almost all, ABENDs [OS/VS2 75]. The CAL system [CAL 69, Gray 72] grouped exceptions into thirty-two error classes to control the selection of an exception handler. UNIX [Thompson 74] recognizes only twelve different exceptions. The CAP system allocates and controls the use of 256 different error names [CAP 76a, CAP 76b, CAP 76c]. If new exception names are not available, exceptions signalled by newly implemented subsystems may interfere with existing exception handling protocols and procedures.

Besides, providing for an extendible set of exception names, it must be possible to avoid ambiguities in the exception identification. If the name which identifies an exception is freely assigned by the signaller, conflicts will occur when two signallers choose the same name to identify otherwise unrelated exceptions. The exception processing facility and exception handling procedures require an unambiguous indication of what happened in order to provide a precise response. PL/I [PL/I 74] and Multics [Organick 72, MPM 75] do not distinguish between identical condition names chosen independently by different procedures.

Finally, some controls on the use of the exception names is necessary to prevent conflicts between independent subsystems. It should not be possible for a subsystem to signal an exception which is associated with the operations of another subsystem. If the signaller can fake an exception, the handler may need to validate the occurrence of the exception before proceeding. Misleading a handler which has special

privileges may cause the handler to misuse its privileges.

To avoid ambiguities and to validate the source of the signal, exception names can be centrally allocated, as in CAP, or the identity of the signaller (signaller-id) can be combined with an exception code supplied by the signaller to form the exception name. Of course, the signaller-id which disambiguates and validates the exception must be unique and unforgeable. Using the signaller-id avoids the implementation of a global exception name allocation mechanism and also adds useful structure to the exception name. The difference between centralized exception naming, with controls on the use of different exception names, and the use of the identity of the signaller to avoid ambiguity is analogous to the difference between the seals and trademarks as discussed by Morris [Morris 73]. Sealing authenticates the contents of the sealed object (the exception name), while trademarks authenticate the source of the object (the signaller).

Either the centralized or the structured exception naming scheme will provide adequate identification and validation while avoiding ambiguities. The centralized naming scheme permits a subsystem to signal any exception for which it can obtain authorization. In a capability based system, authorization to signal an exception can be validated thru the use of capabilities. In such a system, authorization to signal an exception might be passed from one subsystem to another. This generality makes it difficult to verify the source of an exception signal.

The structured naming scheme automatically associates an exception with the subsystem which reports it. This means that it is not possible to signal an arbitrary exception. Occasionally it is necessary to force

the occurrence of a particular exception in order to test new handler procedures. If the desired exception is difficult to produce (e.g. I/O transfer errors), it becomes difficult to test the handler. The handler must be recoded (temporarily) to respond to a different exception used only for testing the handler. When the handler is ready to be installed, it can be modified to catch the correct exception. it handles.

2.2.7 Exception Parameters

The eventual handler of an exception should be given some information about the error. The exception name, as discussed in the preceding section, classifies the exception and participates in the selection of a handler. The exception name is certainly the starting point for information to be passed to the exception handler. Identifying the exception to the handler allows a single handler to respond accurately to several different exceptions.

Besides supplying an exception code, the signaller will usually be able to supply details about the particular exception being signalled. For example, the signaller of an attempted reference to a nonexistent file can pass along the name of the file. This extra information, called the exception message, is useful to the handler of the exception. Different signallers will be able to provide different amounts of information about the exceptions they signal. Therefore, the exception message should be a variable format parameter to the signal operation. At the language level, the type of the exception message can be associated with the exception name to which it corresponds. The signal operation, like other subsystem transfer operations, must be able to take

parameters of arbitrary type. The exception message, along with the exception name should be passed, as parameters, to the eventual handler of the exception. Note that the exception name is used by to the exception facility to implement exception specific handler selection, while the exception message is solely for the benefit of the exception handler.

In addition to information identifying and describing the exception, the handler may need to know the circumstances under which the exception occurred. In particular, the handler may need to know the identities of the signaller and the invoker. If the signaller-id is incorporated into the exception name, only the name of the invoker needs to be added to the parameters passed to the handler. The invoker-id tells the handler on whose behalf it is operating. If subsystems could inquire about the contents of the subsystem call stack, the invoker-id would not have to be passed to the handler. Other information about the environment, such as episode termination restrictions (discussed later) and authorization to access the invoker's state, could also be passed as parameters to the handler. Access to the invoker's state would be passed only if the handler was authorized, by the handler specification, to access the invoker.

Most discussions of and proposals for exception processing facilities do not provide for parameters to the exception handler. PL/I, Multics [Organick 72], CLU [CLU 75], and Goodenough's proposals [Goodenough 75] do not support parameters from the signaller to the handler. Lampson's MPL proposal [Lampson 74b] does provide for an exception message of arbitrary length from the signaller to the handler. The lack of an information passing facility from the signaller to the handler can

lead to unstructured ad hoc protocols for passing this information. For example, in order to make information about the exception available to the handler, PL/I provides built-in functions which return information about the current condition [Nobel 68, PL/I 74]. The ONCODE built-in function returns a condition code which further classifies the condition represented by the condition name. Other special built-in functions, such as ONFILE which returns the name of the file involved in an input/output or conversion condition, provide information about particular conditions.

2.2.8 Handler Environment

Once a handler for a given exception has been selected, control is passed to the entry point of the handler code. The choice of an environment in which to execute the handler code raises several issues. In order to facilitate restarting the invoker after recovering from the exception, the state of the invoker should be preserved. If the failed operation is to be retried, the original parameters of the operation should be preserved.

Some exception handlers need to exercise privileges not accorded to the other subsystems involved in the exception episode. For example, exception driven virtual memory systems need to access the page and segment tables in order to perform their functions. When an exception handler extends a failed lower level, when it must report or log the exception, or when heavy handed recovery procedures must be initiated, the handler may need to make use of privileges logically associated with the handler and not available in the environment of either the signaller or

the invoker.

One privilege which may be needed in the handler environment is the right to access the invoker's environment. For example, the Multics linker must read and update the invoker's linkage section.* . Exception triggered debugging facilities and supervisor services also need to access the invoker's environment. Although it must be possible to confer the privileges of the invoker on the handler, it is not always appropriate to do so. If the handler does not need to make use of the invoker's privileges, it should not be able to do so. Restricting the privileges of the handler conforms to the principle of least privilege [Dennis 66, Saltzer 75, Denning 76], which states that a program should not be able to exercise privileges that it does not actually need.

While most systems provide some mechanism for giving some handlers extra privileges, few systems protect the main line computation from the actions of the handler. In most programming languages and operating systems, the handler execution environment is the same as or includes the environment in which the association is made between the handler and the exception which it handles. It is not possible in these systems to divorce the handler execution environment from the environment is which it is specified (enabled) as the handler of a particular exception. In CLU [CLU 75] and Goodenough [Goodenough 75], the handler's scope contains the invoker's scope, while in PL/I the handler executes as an inner block of the scope which enables it. User defined exit routines in OS/VS2 execute in the environment of the task which declares them

---

* Actually the linker runs in the same environment or ring as the faulted procedure, but conceptually the linker is only interested in the linkage section of the faulted procedure [Daley 68, Janson 74].

[OS/VS2 75]. Supervisor service traps and privileged instruction simulations in CP/CMS [VM 74] run in environments containing the invoker's environment. In CAL, the exception handler ran either in an independent environment or in an environment enclosing the invoker's environment * [CAL 69]. In Multics, some handlers run in the hardcore (e.g. page fault, segment fault), while others run in whichever ring is selected by the procedure call executed to initiate the handler [MPM 75].

It should be possible to separate the definition of the handler execution environment from the environment in which it is nominated (specified) as a handler. The requirements for special handler privileges and handler privilege restrictions suggest that the exception handler should execute in its own environment. As privileges are normally associated with subsystems and made available in the execution environment of subsystem activations, the existing subsystem transfer mechanisms can be used to control the privileges conferred upon exception handlers. By creating a new environment for the exception handler through the use of an ordinary subsystem call, the invoker's state is preserved and the privileges of the exception handler are controlled by the subsystem transfer mechanisms. If subsystem activation environments are system objects, access by the handler to the invoker's environment can be conferred by authorizing handler access to the environment object corresponding to the invoker's activation.

---

* The distinction was conditioned by the form of the signal: "f-return" or "return-with-error". Actually, the only case in which the independent environment or "f-return" form was used was one in which the handler had access to the invoker anyway.

2.3 Handler Specifications

In order to control the response to an exception, there must be some way to indicate which handler should be called under various circumstances. Handler selection is based on the handler specifications which the user and the system make to control the response to an exception. A handler specification associates a handler entry point or gate with a particular exception. By associating handler entry points with particular exceptions, the response to different exceptions can be specified independently. This permits independently developed subsystems to be designated as handlers for different exceptions without risk of interference between exception handlers. Programming generality considerations [Dennis 68] suggest that it should be possible to process unrelated exceptions with unrelated handlers. Several existing systems send control to a single handler for all exceptions. Among the exception processing facilities which do not support exception specific handlers is the IBM SPIE facility [OS/VS2 75] and the fault procedure in CAP [CAP 76b].

To specify the handler, a subsystem gate must be designated. The handler gate may be from any subsystem. The form of the handler reference depends on the external reference mechanism is the system. The handler reference does not need to be bound until after the exception causes the handler to be selected. Dynamic binding of handler gate references allows unused handlers to remain unbound until needed.

The exception name in the exception specification must be bound before the exception occurs. Handler selection must be able to determine which specifications correspond to the current exception. If the

exception name includes the signaller-id, it would seem difficult to postpone binding to the signaller. However, since the exception cannot occur before the signaller is called, we can let the name portion of the specification reference the linkage variable used by the call. This means that the signaller-id will be bound by the call and before the exception specification is needed. When searching for a handler, specifications which reference unbound signallers can be skipped since an exception from that signaller cannot have occured.

Besides containing an exception name and handler gate reference, the handler specification may be tied to one or more activation points. The activation point of an operation is the place in a program where the invocation of the operation is specified. The execution of an activation point invokes the associated operation. A single operation can be invoked from several different activation points or a single activation point can give rise to multiple invocations of the same operation. In figure 2-1 , the operation 'bat' has two different activation points and 'bat' will be invoked several times from the second activation point. Also, the multiplication operation has two activation points, but it will be invoked twenty times whenever 'frog' is called.

Handler specifications can be statically or dynamically associated with the activation points which may give rise to the indicated exceptions. Static handler specifications associate handlers for particular exceptions with particular activation points which might cause the exception. Dynamic handler specifications associate handlers with particular exceptions regardless of the activation point involved. Static association allows different handlers to be associated with different ac-

```
procedure frog;
   begin
      var I : integer;

      call bat( 5 );

      for I := 1 to 10 do call bat(I*I*I);

   end frog;
```

figure 2-1: Invocations and Activation Points

tivation points while dynamic policies allow different handlers to be used on different occasions at the same activation point.

2.3.1 Dynamic Handler Specifications

Dynamic handler specifications are communicated to the exception facility by executing handler enabling operations. The ON statements in PL/I [Nobel 68] are dynamic handler enabling operations. Dynamic association of handlers with exceptions means that the handler for an exception at a particular activation point will depend upon the enable operations executed on the path leading to the activation point. The exception processing environment reflects the most recently executed handler declarations. Dynamic control over the current exception-to-handler associations facilitates precise control over the exception processing environment as a different handler can be supplied on different invocations from a single activation point.

Figure 2-2 introduces notation to represent the execution of dynamic handler specification statements. The 'enable' operation associates a handler with an exception. The 'enable' operation takes two parameters. The first parameter is an exception name, represented here as a string literal containing the signaller-id and the exception code. The second parameter specifies a handler. The handler can be a procedure identifier or a block of code. If a code block is specified, it is treated as the body of a nameless, parameterless procedure. If exception parameters are referenced by the handler, the procedure identifier form of the handler declaration must be used. The parameters to exception handlers are assumed to be in a standard format for all exceptions.

```
procedure toad;
   begin
      var flag boolean;
          .
          .
      { local handler procedure declaration }
      procedure hand1(invId, sigId, exCode, exMess);
         begin
            < body of local handler >
         end hand1;
         .
         .
      call enable("gnat:gone", hand1);
         .
         .
   A: call gnat;
         .
         .
      if (flag = true)
         then call enable("fly:missed",
               begin
                  <handler body A>
               end handler;);

         else call enable("fly:missed",
               begin
                  <handler body B>
               end handler;);
         .
         .
   B: call fly;
         .
         .
   end toad;
```

figure 2-2: Dynamic Handler Specifications

The handler procedure can be either a local or non-local procedure. This notation is presented only to facilitate the presentation of examples.

In the figure, the 'gone' failure of the call to 'gnat' will cause the locally declared procedure 'hand1' to be entered. The 'missed' failure of the invocation of 'fly' on line 'B' will cause either handler body A or handler body B to be executed depending on the value of 'flag' which selects one or the other of the 'enable' statements. Note that different handlers may apply on different invocations from the single activation point of 'fly'.

Maintaining dynamic handler specifications causes exception processing overhead in the absence of signalled exceptions. The maintenance of the association between handlers and the exceptions they service generates main line exception processing overhead to set up for exceptions which have not, and may never, occur. If changes to the exception processing environment are frequent, the overhead may become significant. The continuing main line overhead associated with dynamic handler specifications legislates against their use if the exception processing environment must be updated often.

Executable handler specifications lead to another problem: it is not possible to simultaneously change the exception processing environment and transfer control from one subsystem to another. The simultaneous updating of the exception processing environment and transfer of control to a different subsystem is important when we consider asynchronous exceptions or interrupts. Also, if the exception processing facility is used to control the definition of the virtual machine interface,

the transfer of control from one subsystem to another should cause the virtual machine interface to immediately reflect the new execution environment.

2.3.2 Static Handler Specifications

Static handler specifications always associate the same handler with a given activation point. Since the handler choice strategy in force is always the same at any particular point in the program, the representation of the handler specifications can be constructed once and for all when the program or subsystem is constructed. Because static handler specifications are associated with sections of the program, the exception processing environment can be defined as part of the program representation. The flow of control at run time does not need to be analysed to determine which handler specifications are in force at a particular point in the program. When using static handler specifications, there is no continuing exception processing overhead because the information representing the handler specifications does not need to be updated at execution time. Of course, the elimination of main line exception processing overhead may result in increased exception time processing to decode the static mapping from exception names and activation points to handler gates.

Static handler specifications, by associating handlers with activation points, reflect the point of control at the moment of the exception. This means that the exception processing environment automatically reflects the passage of control from one subsystem to another or from one statement to the next. As mentioned above, static specifications

allow the exception facility to be used to control some aspects of the virtual machine interface. Static handler specifications are necessary when some subsystems are responsible for implementing the virtual machine facilities used by other subsystems. The implementing subsystem will execute in a different virtual machine environment from the one which it supports. Exceptions caused by the implementing subsystem must, in general, be handled differently at the lower level. For example, a page fault in the program responsible for handling page faults is usually an altogether different problem than a normal page fault.

The automatic and immediate updating of the exception processing environment also permits exception processing facilities to be used to deal with asynchronous exceptions such as the console attention key. When asynchronous events are involved, there is no time to execute 'enable' statements on entry to, or on exit from, the routines which service or are affected by such events. The special purpose return-from-interrupt instruction found on many machines illustrates the importance of combining a transfer of control with a change in the exception processing environment.

Static handler specifications can be represented at the programming language level by appending the handler declarations to the syntactic unit to which they apply. Notations for static handler specifications have been suggested by Goodenough [Goodenough 75], Lampson [Lampson 74b], and Liskov [Liskov 76]. Figure 2-3 specifies the same exception handling strategy as figure 2-2. In this case, however, the handlers are statically associated with the statements which might lead to the indicated exception. The bracketed static handler specification con-

```
procedure toad;
   begin
      var flag boolean;
          .
          .
      { local handler procedure declaration }
      procedure hand1(invId, sigId, exCode, exMess);
         begin
            < body of local handler >
         end hand1;
          .
          .
   A: call gnat ["gnat:gone" : hand1 ];
          .
          .
   B: call fly ["fly:missed" :
            begin
               if (flag = true)
                  then begin
                     <handler body A>
                  end;

                  else begin
                     <handler body B>
                  end;
            end handler;  ];
          .
          .
   end toad;
```

figure 2-3: Static Handler Declarations

sists of the exception name, as before, followed by the handler body or the name of the handler procedure. Again, the notation used here is not intended to be taken as a proposal for the syntax to be used to represent exception processing operations at the programming language level. Note that the effect of dynamic handler specifications for the invocation of 'fly' is achieved by testing 'flag' in the body of the handler.

2.3.3 Local, Default, and Imposed Handlers

Given the idea of a static or dynamic handler specifications, we can consider how the handler specifications might become associated with a subsystem. The implementor (or programmer) of a subsystem should be allowed to supply handler specifications to control the response to exceptions encountered at run time. Implementor supplied handler specifications are called local specifications. Local handler specifications are supplied by the implementor as part of the language level representation of the program. Using local handler specifications, the implementor can provide handlers for exceptions of interest. The bulk of the effort in exception handling research has been directed towards designing mechanisms by which the implementer can control exception handling at the level of the program representation.

Local handler specifications can be either static or dynamic. For the reasons suggested above, static handler specifications are preferred. When handlers are statically specified, the specifications can apply to a single activation point, or a single handler specification may apply to several activation points. At the programming language

level, it is convenient to associate handlers with the major syntactic units of the language. Thus, handlers might be associated with operators, statements, blocks, procedures/functions, classes/clusters, or compilation units.

Nested handler specifications can lead to overlapping specifications which specify different handlers for the same activation point. Only one handler at a time can be called to respond to the exception. When local handler specifications overlap the innermost handler specification usually overrides the handler specifications associated with enclosing syntactic units. In figure 2-4 , the failure of the invocation of 'cat' at statement 'A' causes 'handler1' to be executed, while the failure of the call at statement 'B' causes 'handler2', the handler associated with the block, to be called.

Local handler specifications allow the implementor to specify and supply handlers for exceptions of interest. However, it is too much to expect the implementor to specify handlers for all the possible exceptions which might befall the executing subsystem. It should be possible to supply local handler specifications for exceptions of interest while, at the same time, relying on system supplied default handlers to manage the response to other exceptions.

Default exception handlers can provide the response to exceptions not caught by local handler specifications. Implementor supplied local handler specifications, which override default handler specifications, permit the implementor to choose the handler whenever the default is inappropriate. By relying on default handler specifications, the implementor is relieved from the burden of specifying local handlers for

```
begin
      .
      .
  A:  call cat ["cat:fail1": handler1];
      .
      .
  B:  call cat;
      .
      .
end ["cat:fail1": handler2];
```

figure 2-4: Nested Static Handlers

exceptions which are not expected and/or for which the default handler actions are acceptable. However, the subsystem implementor should be able to find out which default handlers are supplied by the system. Without the ability to determine which default specifications are in force at run time, the subsystem implementor cannot give a precise description of what will happen when the subsystem is faced with an exception for which there is no local handler specification.

The system can supply default specifications in several ways. The most direct way is for the system programs which prepare the user's subsystem for execution (e.g. compiler, linker, loader) to add static handler specifications to the local specifications provided by the implementor. The default specifications are like local specifications which apply to all the activation points in the subsystem. Since the defaults apply to the outermost block of the subsystem, local handlers will naturally override the default handler specifications.

It is not necessary to make the assumption that the subsystem implementor always has the privilege of overriding the system supplied handler specifications. If one or more levels of supervisory interface are imposed on a subsystem before it is allowed to execute in a user process, the required supervisors may need to have the first chance to supply the response to certain exceptions. For example, the virtual memory manager normally needs to intercept all exceptions caused by page and segment faults. The system command interface should be activated to orchestrate the response to 'time limit' and the console 'kill' button in order to terminate things in an orderly manner [Needham 71]. Overall process and system control may depend on the proper, system level,

response to selected conditions such as resource or accounting exceptions.

In order to reflect supervisory prerogatives and privileges, it must be possible to impose handlers for selected exceptions on user subsystems. Imposed handlers cannot be overridden by implementor supplied local handler specifications. The imposition of supervisory handler specifications reflects the hierarchic relationship between a subsystem and its supervisor(s). Imposed handlers can be thought of as enforcing parts of the virtual machine interface supplied to the user subsystem. The role of exception processing in the definition and maintenance of the virtual machine environment has traditionally been separated from the exception processing facilities available to the user. Imposed supervisory exception handlers can be integrated into the handler specification and selection facilities of the system. As in the case of default handlers, the user should be able to find out what handlers are imposed in order to understand the computation evoked by the exceptions caused by the user's subsystem.

2.4 Handler Choice Policies

Once the exception processor has been activated by an exception signal, it must select and activate a handler for the reported exception. The criteria used by the exception processor to select a handler define the handler choice policy. The handler choice policy extends the subsystem interface by controlling the flow of control following a reported exception. In the environment of a computation composed of interacting, independently developed, mutually suspicious subsystems, the

handler choice policy must reflect and protect the interests of the subsystems affected by the exception.

Selecting the proper handler for a given exception is somewhat analogous to the choice involved in evaluating generic procedure calls in PL/I [PL/I 74] or generic forms in EL1 [ECL 72, Wegbreit 74]. In these languages the choice of the procedure body to be executed can be conditioned by the number and attributes of the actual parameters. In our case, the exception processor has the exception name and must choose an exception handler. Besides taking into account the exception name, the handler choice policy should also be sensitive to the requirements of the subsystem which caused the exception (the invoker).

In this section several handler choice policies are discussed. The shortcomings of object oriented, global, and inherited handler policies are exposed and then a policy which reflects and protects the interests of the invoker of the failed operation is presented. The invoker controlled handler choice policy overcomes the problems associated with the other handler choice policies without restricting the exception processing protocols available to the user. Invoker controlled handler selection localizes the response to an exception by considering only handler specifications associated with the the invoker of the failed operation.

2.4.1 Object Oriented Handler Choice Policies

Instead of associating exception handlers with the activation points of operations which might lead to the indicated exception, the exception handler can be associated with the operand of the operation.

By associating the exception handler with the object being operated upon, the response to a failure can be controlled on the basis of which object is involved. The handler invoked following the failure of an operation reflects, not the static or dynamic association of handlers with the activation point, but the identity of the object on which the operation was attempted.

An example of an object oriented handler choice policy is the policy of the AED Free Storage Package [Ross 67]. The AED system associates handlers with "zones" of storage. Whenever an attempt to allocate space from a zone fails, the handler associated with that zone is activated. PL/I also provides object oriented handler association for some of the exceptions which it recognizes [PL/I 74]. All input/output conditions can be enabled for particular files. When the condition occurs, the on-unit associated with the file will be entered. Object oriented exception processing has also been proposed recently by Levin [Levin 77]. Levin suggests that program units be allowed to associate handlers with any object instance which they can reference.

Associating exception handlers with individual objects permits exception processing actions to be associated with the particular objects to which they are intended to apply. Because the handler is associated with the object, any such object passed as a parameter to a subsystem will carry its handler associations with it. A subsystem which operates on parameter objects will inherit the handler associations of the object. The behavior associated with operations on the object is affected by the handlers which run in response to reported exceptions. This means that the subsystem which operates on an object received as a

parameter cannot be sure of the effect of the operation unless it controls the handler associations of the object. A similar problem occurs with the inherited handler policy discussed in section 2.4.3.

Not all exceptions can be attributed to a particular object. Errors of usage, for example, cannot always be associated with a particular object. Exceptions caused by failures-of-mechanism or by attempts to operate on nonexistent objects cannot be dealt with under an object oriented handler choice policy. The association of handlers with objects implies that the exception processor must have some way to deduce the current handler given the object involved. If the handler choice information is embedded in the object representation, the exception processor must be able to access that information. User defined, extended objects must also carry handler choice information if object oriented handler selection is to apply uniformly to all objects. This requires either a standardized object representation, as in HYDRA, or the addition of operations for declaring and determining the handler associations for each object type.

An alternative to embedding the handler choice information in the object representation is to maintain process local associations between objects and handlers. The PL/I approach to input/output exceptions implements an inherited, object oriented handler choice policy on a per-process basis. In PL/I, a file may enjoy different handler associations in different processes and is subject to inherited handler associations in each process. Object oriented handler selection can be simulated by other selection policies by using handler specifications associated with the activation points which might cause exceptions associated with the

objects of interest. The selected handler can locate and call the appropriate object associated handler using local tables or information contained in the object. Because object oriented handler choice policies associate handlers with objects, they cannot be applied uniformly to all exceptions. Moreover, they can be locally implemented as an extension to handler choice policies which use handler specifications associated with the appropriate activation points.

2.4.2 Global Handler Choice Policies

The simplest handler choice policy chooses the handler from a process or system wide set of handler specifications. The handler specifications define a global exception-to-handler mapping. The current state of the exception-to-handler map controls the choice of a handler for any signalled exception.

Handler specifications under a global handler choice policy apply to all the activation points which could cause the indicated exception. By maintaining a representation of the exception-to-handler map, the exception processor can determine which handler to activate in response to an exception signal. The handler choice map can be maintained either on a per-process basis or a single system wide map can control the handler choice. Also, the exception-to-handler map can be static and unchanging or the exception facility can support enable operations which dynamically update the handler specifications controlling the handler choice.

A static global handler choice policy provides uniform exception processing responses for all subsystems in a process. The response

depends only on which exception has been signalled. In general, a static global mapping is too inflexible. A static mapping does not permit the response to a given exception to reflect the current state of the computation. Different subsystems may require different responses to the same exception.

Instead of a static exception-to-handler map, the exception facility could allow the map to be updated. This would permit the handler choice policy to reflect the current exception processing strategy of the process. Such a dynamic handler specification facility requires the execution, at run time, of enable operations to maintain the state of the global map. The UNIX system [Ritchie 74, Thompson 74] supports a dynamic global handler choice policy. In that system, a per-process table controls the handling of twelve different exceptions, each of which can be ignored, defaulted, or handled individually by the process. The initial state of the global map of a process is inherited from the creating process. The equivalent of a subsystem call in UNIX resets the handler choice map to the default handlers. There is no equivalent of a subsystem return in UNIX. The JSYS traps in TENEX [Thomas 75] provide a per-process global trap vector which controls the handling of operating system calls. The meaning of the various system calls is controlled by the state of the JSYS trap vector.

Under a dynamic global handler choice policy, each subsystem can enable handler specifications to reflect its own exception handling needs. Unless a subsystem executes its own enable operations, the subsystem will inherit the exception handling strategy of its caller. By updating the global map, a subsystem can enable handlers for the excep-

tions which it wishes to process specially, while existing handler specifications provide default processing for other exceptions. If the right to enable handler specifications for particular exceptions can be controlled, handlers can be imposed on subsystems lacking the necessary privilege. This facilitates high level control over the set of subsystems executing in a process since handlers for exceptions such as console interrupts and resource problems can be imposed on all but the most privileged subsystems.

Figure 2-5 illustrates the effects of handler specifications in different procedures/subsystems under a dynamic global policy. The failure of the 'dog' call at 'A' causes 'hand1' to be selected. At statement 'C', the handler is inherited from the caller of 'cat'. If 'cat' is called from 'ant', 'hand1' will be chosen to handle a "dog:bite". Finally, the response to the failure of the 'dog' call at 'B' in 'ant' is controlled by the handler specification in 'cat'.

Under a dynamic global handler choice policy, not only are the handler specifications in force for a subsystem inherited by the subsystems it calls, but also, handler specifications enabled by a subsystem remain in force when the subsystem returns to its caller. This is a serious problem since the exception processing environment of a subsystem may be modified as the side effect of a call to any other subsystem. This makes it difficult for the subsystem implementor to control what will happen in response to exceptions encountered at run time. The updatable global exception-to-handler map is a global variable and, as such, it increases the complexity of the computation by producing new dependencies between otherwise unrelated programs in different subsys-

```
procedure ant;
   begin
         .
         .
      call enable("dog:bite", hand1);

   A: call dog;      {"dog:bite" enters 'hand1'}

      call cat;
         .
         .
   B: call dog;      {"dog:bite" enters 'hand2'}
        {because of enable in 'cat'}
         .
         .
   end ant;


procedure cat;
   begin
         .
         .
   C: call dog;      {"dog:bite" enters 'hand1' if}
                     {'cat' called from 'ant'     }
         .
         .
      call enable("dog:bite", hand2);

   D: call dog;      {"dog:bite" enters 'hand2'}
         .
         .
   end cat;
```

figure 2-5: Global Dynamic Handlers

tems [Wulf 73].

2.4.3 Inherited Handler Choice Policies

The global handler choice policy leads to side effects in the exception processing environment which make it difficult to maintain control over the response to signalled exceptions. The side effects of a subsystem call on the exception processing environment can be eliminated by (logically) undoing the handler specifications of a subsystem when the subsystem activation is terminated. Under an inherited handler choice policy, the effects of handler specifications supplied by a called subsystem are reversed when the subsystem activation is terminated. Exceptions for which there is no local handler specification will be processed according to the most recent applicable handler specification which was supplied by some active subsystem.

Instead of maintaining a single set of handler specifications, the exception facility implementing an inherited handler policy can associate handler specifications with each subsystem or with each subsystem activation. To select a handler for a given exception, the specifications of the invoker are checked first. If the invoker has not specified a handler for the exception, the handler search traces back thru the dynamic sequence of subsystem activations. The handler specifications associated with each subsystem activation are checked until a handler for the exception is found.

The inherited handler policy is exemplified by the condition facility in PL/I [Nobel 68], the signal mechanism in MPL [Lampson 74b], and

the BLISS [BLISS] enable statement. In PL/I, ON statements are executed to enable handlers for specific exceptions. Once enabled, a handler specification remains in force until either 1) the enabling subsystem is terminated, or 2) the handler specification is temporarily overridden by enabling operations in the same or another subsystem, or 3) the handler specification is explicitly removed (reverted) by the enabling subsystem.

Figure 2-6 , containing the same program as figure 2-5, illustrates the effects of an inherited handler choice policy. At 'A' and 'B', 'hand1' will be chosen, while 'hand2' is selected at 'D'. The handler choice at 'C' depends on the caller of 'cat'. If 'cat' is called from 'ant', 'hand1' will be chosen.

The inherited handler policy allows any subsystem to override the handler specifications inherited from its caller. However, a subsystem can be sure of how a particular exception is handled only if it enables its own handler specification. Exceptions not covered by the invoker's handler specifications may be handled differently on different calls to the invoker due to different handler specifications in force at the moment of the exception. This makes it difficult to specify the effects of a call to the invoker without discussing the dynamic state at the moment of the call.

In a procedure or operation based system, each procedure activation executes on behalf of and in response to the needs of the caller. Every procedure activation returns control to the most recently activated, but not yet terminated, procedure activation. One major difficulty with the inherited handler policy in a procedure based system is that the handler

```
procedure ant;
   begin
         .
         .
      call enable("dog:bite", hand1);

   A: call dog;      {"dog:bite" enters 'hand1'}

      call cat;
         .
         .
   B: call dog;      {"dog:bite" enters 'hand1'}
         .
         .
   end ant;


procedure cat;
   begin
         .
         .
   C: call dog;      {"dog:bite" enters 'hand1' if}
                     {'cat' called from 'ant'     }
         .
         .
      call enable("dog:bite", hand2);

   D: call dog;      {"dog:bite" enters 'hand2'}
         .
         .
   end cat;
```

figure 2-6: Inherited Handlers

is designated by, and presumably operates on behalf of, the subsystem which supplies the selected handler specification. When the selected handler specification does not come from the invoking subsystem, we find that the activated handler must serve two masters: the invoker which caused the exception, and the subsystem which supplied the handler specification.

We believe that a carefully written subsystem could probably protect itself against interference from inherited exception handlers. However, the suspicious subsystem implementor cannot depend on the goodwill of exception handlers inherited from the caller. The inherited handler may adversly affect the invoker by returning incorrect results, by directly manipulating the invoker's state, or by not returning at all. Consider, for example, the "linkage fault" exception in Multics. Permitting a handler supplied by a hostile subsystem to direct the name search and to "patch" the link in the linkage segment of the invoker would violate the security of the invoking subsystem. An improperly "patched" link can cause trouble when the invoker passes sensitive parameters to the masquerading procedure referenced by the improper link.

Under an inherited handler policy it is difficult to establish control over the special privileges which might be granted to the handler. In particular, access to the state of the invoker and control over permitted handler terminations (see section 2.5) should be specifically authorized by the invoker. The natural place for the termination and access authorizations is the handler specification. When the invoker supplies the handler specification, the degree of trust between invoker and

handler can be reflected in the authorizations with the handler specification. When handler specifications are inherited, they cannot be used to control the handler privileges which affect the security of the invoker. On the other hand, inherited handler specifications can be used to control the selection of unwind targets (see section 2.5.5).

The inherited handler policy introduces a channel for interaction and interference among subsystems. Passing exceptions not covered by the invoker's handler specifications to the invoker's caller violates the principles of programming generality [Dennis 68]. The invoker's caller should not need to be aware of exceptions signalled by subsystems called by the invoker. The computation evoked by a subsystem call may depend on which handlers are called in response to exceptions caused by the subsystem. If handler specifications are inherited from the caller, the effects of a subsystem call cannot, in general, be specified without considering the handler specifications of outstanding subsystem activations at the moment of the call.

2.4.4 Invoker Controlled Handler Choice Policies

The global and the inherited handler policies seem to produce unwanted dependencies between independently developed subsystems. Can the individual exception processing needs of different subsystems be met without violating programming generality principles and creating dynamic program dependencies? The context most immediately affected by an exception is the subsystem activation which called for the execution of the failed operation. The environment containing the most information about the circumstances which led to the call of the failed operation is

the invoking subsystem. Because of the intimate relationship between the failed call and the program containing the call, the invoker is an appropriate source for the information to control the response to a signalled exception. Handler choice policies which depend only upon handler specifications associated with the invoker are called invoker policies.

Under an invoker policy, the response to a signalled exception is under the complete control of the handler specifications associated with the invoker of the failed operation. As in the case of the inherited handler policy, the subsystem implementor can specify local handlers for exceptions of interest. Local handler specifications can be either static or dynamic. Language level constructs similar to those proposed by Goodenough [Goodenough 75] or the "except" facility in CLU [CLU 75] can be used to express local handler specifications.

However, as suggested in section 2.3.3, it is burdensome to require the implementor to supply handler specifications for every exception which might be signalled to the subsystem. The inherited handler policy relies on the dynamic execution environment to supply handler specifications whenever the invoking subsystem has not specified a handler. The ability of the inherited policy to provide default handler specifications from the dynamic environment is both a strength and a weakness of that policy.

Instead of relying on handler specifications inherited at run time from the calling subsystem, default handlers can be specified once and for all during program preparation. Default handlers can be supplied by the system facilities which participate in the subsystem definition

process which prepares the subsystem for execution. Before a program can be executed, one or more levels of supervisory interface are usually imposed on the program. Supervisory interfaces define and enforce the virtual machine environment in which the program executes.

The preparation of a program for execution involves a sequence of transformations of the program representation and the binding of program variables and references to system objects and resources. These transformations and bindings are performed by various system facilities. The process begins with the translation of the language level representation of the program to a machine level representation. Link editing operations bind free variables of the program. Finally, the instantiation of the subsystem in a process requires the assistance of kernel level facilities which perform allocation and bind the program and its data structures to virtual memory locations [Daley 68, Jones 73, Sturgis 73].

Subsystem preparation involves a sequence of steps during which the current program representation is submitted to various system supplied operations which modify and transform the representation. Each operation which works on the program/subsystem representation can be thought of as a supervisor because each has access to the current representation of the (not yet executable) program/subsystem. Each supervisor which is given access to the program representation can impose run time interfaces by suitably modifying the representation. For example, the compiler can generate calls to its run time component and the link editor can insert overlay management code.

The order in which supervisors operate on a program can be used to define a hierarchy among the supervisors. When several levels of supervisory interface are required, the less privileged levels are usually applied to the program before the more privileged supervisors are called to do their part of program preparation. Thus, the language translator is usually the first system operation to manipulate the program representation. Then link editors are called to combine independently translated subsystem fragments. Eventually the user virtual machine supervisor must be asked to accept the subsystem text and to make it into an executable (callable) subsystem. If several virtual machine levels are needed to define the user virtual machine, there is a lowest level, kernel interface, which must be invoked regardless of which virtual machine is preparing the subsystem. Because program preparation proceeds from high level supervisors (language processors, link editors) to more privileged virtual machine supervisors, the temporal sequence in which supervisors operate on the program/subsystem reflects the hierarchy of virtual machine interfaces in the system [Sturgis 73, Lampson 71, Lauer 74].

If handler specifications can be statically represented as part of the subsystem text, the sequence of system supervisors which help prepare the subsystem can add handler specifications to the subsystem representation when they are called to work on the not yet executable subsystem. Instead of relying on the dynamic environment to supply default handler specifications, the system facilities which prepare the program for execution can introduce static handler specifications into the representation. The supervisor supplied default handlers can provide standard responses to exceptions not caught by implementor supplied

local handler specifications. For example, link edit for testing can supply the debugger as a default handler; while a production link edit would supply a default handler which initiates recovery and restarts the system after dumping relevant debugging information.

The time sequence in which the supervisors operate on the program representation can be exploited to extend the nesting of local handler specifications and to define a priority for default handler specifications. Just as inner block local handlers override the handlers associated with enclosing blocks, early (less privileged) supervisor defaults can override the default handler specifications supplied by supervisors which participate in subsequent stages of program preparation. A default handler specification applies only if there is no earlier handler specification for the same exception. Unlike local handler specifications which may apply to a subset of the activation points of the program, default specifications apply to all the activation points which might give rise to the indicated exception.

By supplying default handler specifications during the program preparation process, the default exception handler environment of the subsystem will not depend on the dynamic environment at the moment of the exception. Supervisor supplied default handlers, like inherited handlers, will be executed only if there are no overriding local handlers. Unlike inherited handlers, supervisor defaults do not lead to dynamic dependencies between separately developed subsystems because they are defined once and for all during subsystem definition and creation.

Supervisory subsystems should also be able to impose handlers on the subsystems under their control. By imposing handlers on programs as they are transformed at each stage of program preparation, supervisory prerogatives can be stated and enforced. The imposition of exception handlers reflects the hierarchic relationship between a subsystem and its supervisor(s).

The priority of supervisor imposed handler specifications, like default specifications, can be controlled by the order in which supervisory subsystems are called during program preparation. As mentioned earlier, whenever several levels of supervision are required, the order in which the supervisors participate in the preparation of a executable subsystem usually reflects the hierarchic relationships among the supervisors. Like default specifications, imposed handler specifications apply to all the activation points in the subsystem. However, the priority of imposed handlers should be the reverse of the priority for default handlers. Imposed handler specifications override all preceding handler specifications.

A crucial point here is that once a supervisor adds its imposed handler specifications to the subsystem representation, it must not be possible for less privileged supervisors or the user to remove or override those specifications. One way to enforce this requirement is to have each supervisor call the next (more privileged) supervisor after adding its default and imposed handlers but before returning to its caller. The most privileged (kernel) supervisor, after installing its own handler specifications, can freeze the subsystem representation. The frozen subsystem representation cannot be modified without recreat-

ing it from scratch. The representation of an object can be frozen by changing (copying) it to a different type of object which is inaccessible to all, or by providing for a new state of the storage object containing the representation. For example, the HYDRA system [HYDRA 74] allows most objects to be frozen, making it impossible to modify their state.

If a supervisor does not return until it has called the next supervisor, and if the kernel supervisor freezes the representation, the user and other supervisors will be unable to update the representation containing the imposed handler specifications. In practice, the first few stages of subsystem preparation (compile and link) may be unprotected but, once the user virtual machine supervisor is called, it will complete the subsystem preparation by calling the next supervisor before it returns to the user. Once a supervisor calls its supervisor, the representation will be frozen by the kernel before the supervisor regains control.

In summary, the invoker controlled handler choice policy, because it depends only upon handler specifications associated with the invoker, does not lead to run time dependencies between independent subsystems. Because the handler specifications are statically associated with the activation points which might cause the exception, the response at run time to any exception can be determined without considering the dynamic activation environment. The implementor, if (s)he cares to, can examine the subsystem representation and the published specifications of the supervisors to exactly determine the exception processing environment of the subsystem.

Three sets of handler specifications control the choice of an exception handler at run time. The handler choice rule searches first for an imposed handler, then for a local handler, and finally for a default handler. Conflicting handler specifications are resolved by priority rules within each set of handler specifications and between the three sets of specifications. The last imposed handler, corresponding to the more privileged supervisor, overrides earlier imposed handlers. The lexical nesting of statically specified local handlers gives priority to the innermost local handler specification. If no imposed or local handler applies, the earliest default handler specification is chosen.

Invoker policies statically associate handler specifications with activation points in the program of a subsystem. Static handler specifications do not consume main line overhead and by always providing the same handler for a given activation point and exception, they avoid uncertainty about the exception processing environment under which the program will execute. Extending the nesting of local handlers to the supervisory programs which prepare and oversee the execution of the subsystem facilitates supervisory control over exception processing and allows supervisory programs to impose handlers and to supply handlers for exceptions which are not caught by local handler specifications. The exception processor developed in Chapter Four implements an invoker controlled handler choice policy.

2.5 Handler Terminations

Once an exception handler has been called and has completed execution, control must be returned to the interrupted main line computation.

Existing and proposed exception facilities do not provide very much flexibility in the termination of exception episodes. A variety of handler termination modes is needed to reflect the various outcomes of the handler's attempts to recover from the exception. The handler may indicate that the exception episode is over or it may indicate that exception processing should continue. Because the possible handler terminations may affect the invoker in different ways, the handler specifications must be extended to control which termination modes will be allowed for each handler. The terminations permitted a handler reflect the expected outcomes of handler execution and the degree of trust between the handler and the invoker.

In this section, a number of handler termination modes are discussed. Some of the handler terminations lead to the termination of the exception episode while others cause the episode to continue. The handler terminations which end the current exception episode include: 1) continue the invoker following the failed operation, 2) restart the invoker so as to retry the failed operation, 3) exit to a non-standard continuation of the invoker, 4) abort the invoker and signal a new exception to the invoker's invoker, and 5) unwind the computation to an earlier subsystem activation. The handler terminations which do not terminate the exception episode are: 6) reclassify the exception, and 7) reject responsibility for the current exception.

2.5.1 Continue Termination

The handler can sometimes recover from an exception by simulating the effects of the failed operation. Whenever the handler is able to

produce the results and side effects which were expected from the failed operation, the execution of the handler can replace the call of the failed operation. The recovery block architecture [Horning 74] is based on the idea that an alternate computation may be able to produce correct or acceptable results after the primary computation has failed. For example, a handler for arithmetic underflow exceptions can return zero as the result of a failed floating point operation.

If the handler has simulated the failed operation, the exception facility can continue the execution of the invoker by returning from the exception processor to the invoker. Not only can the invoker be continued by an ordinary subsystem return from the exception processor, but also, results supplied by the handler can be returned to the invoker. The results generated by the handler are returned to the invoker in the place of the results which should have been returned by the failed operation. When results are returned from the handler to the invoker using continue termination, the entire episode may be transparent to the invoker. If there are no unusual side effects of the failed operation or the handler execution, continuing the invoker with the handler supplied results can conceal the occurrence of the exception from the invoker. Transparent recovery minimizes the interaction between the invoker and the handler.

Instead of simulating the failed operation, the handler can modify the invoker's state as a side effect of its execution and then order the continuation of the invoker. Handler side effects can be tested for by the invoker's program. One thing the handler can do is to convert an exception signal to a change in the value of a status variable accessi-

ble to the invoker. The invoker can then test the status variable after being continued by the handler. Note that such a mechanism for converting signals to status variables is inefficient in cases where the exception occurs, but is more efficient than status code exception reporting whenever the exception does not occur. The status variable can be initialized once by the invoker and does not need to be passed to or returned from called operations. Whenever the exception does not occur, status variable manipulations do not generate any overhead.

Continue termination can be used to return control to the invoker following successful recovery or when the handler has modified the invoker's state so that the invoker's program can respond to the exception. In the first case, the simulation of the failed operation by the handler makes the exception transparent to the invoking subsystem. Interactions between the handler and the invoker are minimized. Continue termination can also be used to return control to the invoker after the handler has posted side effects on the invoker's state. The side effects of the handler execution can be used to trigger the invoker's programmed response to the reported exception. Communicating the occurrence of an exception by posting side effects on the state of the invoker leads to close interaction between the invoker and the handler.

2.5.2 Retry Termination

Instead of simulating the failed operation, the handler may be able to correct the cause of the exception. If the cause of the exception is removed, the failed operation can be retried. Like continue termination, retry makes the exception episode transparent to the invoker.

When the handler returns to the exception processor requesting retry termination, the exception processor must use a privileged subsystem transfer operation to return control to the invoker without incrementing the invoker's instruction pointer.

Recovery actions which can correct the cause of the exception include repairing the parameters to the failed operation and handler actions which drive the signalling subsystem into a state from which the failed operation can succeed. For example, the failure of an attempt to open a non-existent scratch file for output can be circumvented by creating the file and re-executing the open operation. Linkage faults in Multics are handled by repairing the missing link and then retrying the instruction which caused the fault. Page faults are handled by causing the missing page to be brought into real memory and then retrying the failed memory access.

Retrying a failed operation is also sometimes appropriate when the failed operation was partially completed before the exception was signalled. If the failed operation has updated some kind of progress indicator which can be used to control the continuation of the operation, retrying will caused the failed operation to continue from where it left off. The MVCL (string copy) instruction of the IBM 370 system is an example of an operation which may fail after being partially executed. The MVCL instruction updates its parameters as it executes to reflect the progress made. Re-executing the MVCL after correcting the cause of the failure (e.g. page fault) will cause the remainder of the string to be copied. The allocate operation described in section 3.2.6 can also be retried after failing on a virtual memory fault.

Retry termination should not be used if the original parameters of the failed operation have been lost, or if side effects of the failure make re-execution of the operation inappropriate. Another problem with retry termination is the possibility that the handler did not really correct the cause of the exception. If the same exception occurs when the failed operation is retried, the same handler would normally be called. The handler would call for the failed operation to be retried again, leading to an endless loop of exceptions and retries. The exception processor can make some checks to detect the simpler forms of such loops. In the general case, however, it is difficult to distinguish between legitimate repeated exceptions and unsuccessful retries.

2.5.3 Exit Termination

If the handler is closely associated with the invoker, it may be allowed to force the invoker to continue execution at some point other than the current execution point. Not infrequently, the exception should cause the invoker to exit a loop or to transfer to some execution path which reflects the occurrence of the exception. A return from the handler to the exception processor with results calling for exit termination causes control to be returned to the invoker at the address indicated by the handler. The effect of exit termination is to cause a non-local goto from the handler to the indicated address in the invoker.

As Liskov points out [Liskov 76], the association of handlers with the activation points which cause the exception should be specified separately from the continuation points to which the handlers exit. Syntactic constructs at the programming language level can be employed

to give structure to the exit transfer. When the handler body is locally specified as part of the invoking subsystem, exit termination can be used to return control at an arbitrary point in the invoker. When the handler is not part of the invoking subsystem, exit termination should not normally be allowed. One exception to the above rule is the debugging supervisor. The debugger should be able to return control to the debuggee (invoker) at any point in its program.

2.5.4 Abort Termination

It is not unusual for a reported exception to lead to the failure of the invoker of the failed operation. If the cause of the exception cannot be circumvented and alternative calculations to achieve the specified effects of the call to the invoker are unsuccessful, there is no alternative but to signal the failure of the invoker to the invoker's invoker. If the invoker's handler determines that the exception will prevent the invoker from satisfying its output specifications, the handler should be able to force the failure of the invoker and propagate the exception to the environment of the invoker's invoker.

Signalling the failure of the invoker without returning control to the invoker is called aborting the invoker. Abort terminates the current exception episode and initiates exception processing on behalf of the invoker's invoker after the invoker has been terminated. Abort termination permits exceptions to propagate from callee to caller until a subsystem activation which can handle the exception is found.

At each stage during the propagation of an exception, the current invoker's handlers are free to continue the propagation, using abort, or to end the exception episode by returning control to the current invoker. For example, consider a payroll system composed of interacting subsystems. If the check writing subsystem is asked to prepare a check for negative dollars, it should signal an exception. The exception may cause the employee pay calculation subsystem to be aborted by its handler. When the exception reaches the payday subsystem which controls the computation, the control program can print an error report and proceed to the next employee. Abort is also useful in debugging situations. If the debugging supervisor gains control as a handler of the debuggee, it can abort the subsystem under test so that the system can be returned to a clean state from which the programmer can modify the subsystem and then test the new version.

The handler which requests abort termination should be trusted by or imposed upon the invoker. A handler must be authorized by the invoker supplied handler specification to be allowed to cause an abort. In order to implement abort termination, the exception processor must make use of yet another privileged subsystem transfer operation. The abort operation, described in Chapter Three, terminates both the exception processor activation which executes the abort and the activation of the invoker. Abort then reactivates the exception processor to process the new exception in the environment of the new invoker. The new activation of the exception processor can process the exception as if it had been directly signalled by the old invoker instead of by way of the abort operation.

When an exception is propagated from the invoker to the invoker's invoker, the exception code and message can be supplied by the handler which calls for the abort. It would be inappropriate, however, to use the handler as the signaller of the new exception. The invoker's invoker can not be expected to be aware of the identity of the invoker's handlers. If the signaller-id of the new exception was the handler which ordered the abort, the invoker's invoker would be confronted with an exception from a subsystem which it never called. Similar arguments suggest the new exception should not be the same as the original exception. The original exception was signalled by an operation called by the invoker and not by the invoker's invoker.

In almost all cases, it is appropriate to make the invoker the signaller of the propagated exception. In some cases, however, it is useful to let the exception processor be the signaller of the new exception. Sometimes the exception processor must initiate the abort because of improper usage of the exception facility. One example is when the exception processor is unable to find a handler for the current exception.

When a subsystem activation is to be aborted, it is often necessary to restore the subsystem to a consistent state before the subsystem activation is abandoned. If the handler calling for the abort is closely associated with the invoker, it can perform the actions necessary to bring the invoker to a consistent state. Instead of directly restoring the invoker's state, the handler can call a 'cleanup' routine belonging to the invoker. The 'cleanup' routine can be called directly by the handler or the handler can reclassify (see Section 2.5.6) the exception

to cause a 'cleanup' handler belonging to the invoker to be located and called. The 'cleanup' handler can then issue the abort after restoring the state of the invoker.

2.5.5 Unwind Termination

The exception processor can provide an unwinding facility designed to terminate un-needed subsystem activations in order to return control to an earlier subsystem activation. Unwinding is useful when the results of several subsystem activations are no longer needed because the larger computation in which they are participating is being abandoned for one reason or another. Each of the subsystem activations which is to be terminated should be given a chance to restore its data structures to a consistent state before it is forced to permanently relinquish control.

The handler calling for the unwind must indicate how far the process should be unwound. The handler which initiates the unwinding can also supply an exception message to inform the unwind target of the reason for the unwind. When the indicated subsystem activation is reached, it cannot be continued normally. The exception processor can signal an 'unwound' exception to the target subsystem after the superfluous subsystem activations have been terminated.

The selection of the unwind target raises some protection issues. It should not be possible for an arbitrary handler to order an arbitrary unwind. One way in which the unwind target can be validated is to require that the handler present a non-local reference to the target sub-

system activation. The ability of the handler to produce a validated reference to the unwind target indicates, at least, that the handler is somehow related to the target.

To unwind the process, the exception processor must force the termination of the intervening subsystem activations without compromising their integrity. Before terminating a subsystem activation, the exception processor should search for and activate 'cleanup' handlers belonging to be subsystem about to be terminated. The signaller of the 'cleanup' should be the exception processor. The 'cleanup' handlers can restore the state of the subsystem activation about to be terminated. Subsystems which don't have 'cleanup' handlers will be terminated without having a chance to restore their data structures.

To implement the unwinding facility, the exception processor uses the privileged abort operation to force the termination of subsystem activations. In order to continue the unwinding following each abort, the terminated exception processor activation must communicate to the new exception processor activation the fact that unwinding is in progress. Making the exception processor the signaller of the abort permits the new exception processor activation to detect that unwinding is under way by examining the signaller-id and exception code of the propagated exception.

At each step, the exception processor checks to see whether the target subsystem activation has been reached. If so, an 'unwound' exception from the exception processor communicates to the target the fact that an unwind has occurred. The exception message from the handler which initiated the unwind is also passed to the target. If the unwind

target has not been reached, the exception processor searches for and calls the 'cleanup' handler associated with the current invoker. The only termination permitted the 'cleanup' handler is reject termination (see Section 2.5.7). All other handler termination requests from the 'cleanup' handler are ignored and the exception processor propagates the unwinding to the next level by aborting the current invoker.

A number of thorny implementation problems are raised by unwind termination. One problem is that a 'cleanup' handler may fail to return control to the exception processor. Another problem stems from the fact that a second unwind may be initiated by the handler of an exception caused by the 'cleanup' handler. The first problem requires the implementation of watch dog timers to wrest control from handlers which refuse to terminate. The second problem requires extra code in the exception processor to sort out overlapping unwinds and to choose the most distant target. The implementation of unwind is discussed, along with the other handler terminations, in Chapter Four.

2.5.6 Reclassify Termination

The handler termination modes discussed up to this point end the current exception episode. Instead of ending the exception episode, the current exception name can be changed to reflect the handler's decision that the exception should be reclassified. For example, a page fault caused by referencing beyond the logical end of a segment can be reclassified from 'page-fault' to 'non-existent-page' by the page fault handler. Reclassifying an exception allows the handler to recharacterize the exception based on its own analysis of the situation.

To reclassify the exception, the handler can return the new exception code and message to the exception processor along with the indication that reclassification is desired. Since the handler is presumably not a stranger to the invoker, the signaller of the reclassified exception can be the handler. The handler's decision to reclassify an exception causes the exception processor to search for and call handler(s) for the new exception.

Reclassify termination is equivalent to a signal from the handler. Indeed, there is nothing to prevent the handler from signalling instead of reclassifying. The exception facility should be prepared to convert signals from a handler to reclassifications. Also, the exception processor should make sure that the reclassified exception is different from the current exception. If it is not, the exception processor should convert the reclassify to a reject termination (see next subsection).

As in the case of retry, careless or malicious handlers can cause the exception facility to loop. By reclassifying the exception to be the same as it was earlier in the same exception episode, the handler can cause the exception processor to call a previously called handler. That handler, behaving as before, can again reclassify the exception leading to an endless sequence of reclassifications. The exception processor can detect such loops by keeping track of which exceptions have been encountered during an exception episode.

2.5.7 Reject Termination

Yet another handler termination action permits the handler to reject responsibility for the exception. In this case, the handler has done whatever it could to recover from the exception, but cannot itself handle the exception. Reject termination allows the handler to give up on the exception in the hope that some other handler can deal with the problem. By rejecting responsibility for the exception, the handler directs the exception processor to search for and call another handler for the same exception.

Under the invoker controlled handler choice policy of Section 2.4.4, the handler specifications for a given exception are ranked by their type (imposed, local, or default) and order of specification. The handler specification priority ordering yields a sequence of handler specifications for each exception. If the first handler rejects responsibility for the exception, the next handler specification in the sequence can select the next handler for the exception.

If the handler specification sequence is exausted, reject termination by the last handler cannot cause another handler for the same exception to be called. In this case, the exception processor can reclassify the exception to the 'noHandler' exception to indicate that no handler can be found for the current exception. If the current exception is 'noHandler', more drastic measures must be taken (see Section 4.4.7).

2.6 Summary

This chapter has discussed a number of exception processing issues. The requirements for uniform exception reporting, unambiguous exception naming, low main line overhead, and isolation of the handler environment were analysed at the beginning of the chapter. The issues surrounding handler specifications were discussed in section 2.3. The advantages of static associations between handlers, exceptions, and activation points were pointed out. Also the distinctions between local, default, and imposed handlers were discussed in Section 2.3.3.

Given the requirements for exception episode initiation and the mechanisms for specifying exception handlers, the problem of selecting a handler without compromising the integrity of the invoker was examined. The traditional global and inherited handler choice rules were shown to lead to undesirable interactions between different subsystems. The invoker controlled handler choice policy was then presented. This policy, with it's default, local, and imposed handlers was shown to protect the interests of the invoker while providing flexibility in the selection of a handler. The notion of a sequence of supervisors which participate in the preparation of the subsystem for execution was exploited to control the priorities of default and imposed handlers. The invoker policy, by considering only handler specifications associated with the invoker, provides for the isolation of protected subsystems, reflects system supervisory privileges, and permits flexibility in the control of exception processing.

The final section of this chapter discussed the handler termination modes which should be supported by the exception facility. A variety of

handler terminations is necessary to reflect the possible outcomes of the handler's attempts to recover from the exception. Handler terminations which return to the invoker, terminate the invoker, and which cause additional handlers to be selected were shown to respond to the various ways in which handlers might wish to terminate their attempts at recovery.

Chapter Three

An Implementation Model

3.1 Introduction

The implementation of a system level exception processing facility requires that some exception processing operations be integrated into the base level of the system. In this chapter we develop a process model supporting protected, mutually suspicious subsystems and implementing exception reporting and termination mechanisms. The implementation of protected subsystems requires defining the environment in which programs execute and specifying how that environment changes when control passes to a program in a different subsystem. The term *domain* has been used to denote the protection environment associated with a running or runnable program. Although the term has been employed in various contexts [Lampson 71, Schroeder 72, Spier 73], we use the term "domain" to refer to the protection environment within which a subsystem activation executes. Different activations of a subsystem should execute in different domains to reflect the differing parameter privileges of the several activations.

In order to demonstrate an implementation of the exception processing facility developed in this thesis, one must assume some sort of starting point. The relevant features of a number of systems which support the ability to create arbitrary protected subsystems have been abstracted and combined into what we call the *basic process model*. Sec-

tion 3.2 defines the basic process model with an emphasis on its addressing mechanism and the operations for creating domains and transferring control from one domain to another. Section 3.3 augments the basic processor model to support exception processing. The exception processing primitives for initiating and terminating exception episodes must be added to the basic process model because they require domain switching protocols not available in the basic process model. Finally, Section 3.3 elaborates the basic processor implementation of the subsystem activation stack so as to permit a small portion of the stack to be resident in real memory while the rest of the subsystem activation records which comprise the stack are stored in virtual memory.

3.2 The Basic Process Model

The basic process model implements an execution environment which supports protected, mutually suspicious subsystems. A subsystem executes in a domain which is defined by the contents of its *current address space*. The current address space is a mapping from computable integer addresses to storage locations. The current address space binding is changed by the subsystem transfer operations. The basic process model includes operations for calling and returning from subsystems and operations for allocating and freeing temporary storage. The process model does not include facilities for switching physical processors from one process to another. We ignore the issues involved with multiprocessing because the exception processing facility we wish to develop does not require (or support) concurrent processing.

3.2.1 Basic Addressing

Basic addresses are the integer addresses which a program can present to the execution engine. The mapping from computed addresses to stored data is defined by the basic addressing mechanism which is designed to reflect the allocation and binding strategies needed to implement shared protected subsystems. Basic address spaces are created when a subsystem is activated and destroyed when the activation terminates. An execution point is also associated with each address space. Basic addressing in the process model distinguishes between storage allocated and initialized during subsystem creation, on the first call to the subsystem by a particular process, and on each call to the subsystem.

Basic addresses reference locations in one of three storage segments: 1) the per-subsystem root segment, 2) the per-subsystem, per-process incarnation segment, or 3) the per-call activation frame. The per-subsystem root segment is allocated and initialized during subsystem creation. It contains, among other things, the code for the operations of the subsystem and the variables which are shared by all activations of the subsystem in all processes. A subsystem can be identified with its root segment because the procedures and private data structures of the subsystem are kept in the root segment.

Subsystems typically can be executed concurrently by many independent processes. Subsystem implementors should be able to allow multiple simultaneous activations without having to explicitly handle the multiplexing of their per-user variables. In order to facilitate the use of per-user or per-process variables, the addressing mechanism of the

process model binds references to such variables to the subsystem's incarnation segment. The incarnation segment can be allocated and initialized when a subsystem is first used in a process.

Per-process, per-subsystem incarnation storage is also known as own storage [Spier 73, ALGOL 60], static storage [PL/I 74], and linkage storage [Daley 68]. Incarnation storage is used to maintain per-user subsystem state information between subsystem activations. An incarnation segment is necessary only if the operations of the subsystem need to save per-process information from one activation of the subsystem to the next. A compiler, for example, does not usually need to save any information from one activation to the next. A terminal interface subsystem, on the other hand, might need to remember terminal characteristics and buffer characters on a per-process basis. A dynamic linking facility [Daley 68] makes extensive use of incarnation storage to store linkage variables.

The operation of allocating and initializing the incarnation segment, if it is needed, is called instantiating or making the subsystem known to the process. In order to initialize the incarnation segment when a subsystem is made known, a template of the incarnation segment can be stored in the root segment of the subsystem. The template can be copied into the new incarnation segment when the subsystem is made known. This approach to subsystem instantiation is similar to the management of the linkage section in Multics [Organick 72, Daley 68].

In addition to per-subsystem and per-process storage, we must make some provision for accessing the arguments of a subsystem call, allocating local variables and temporaries, and accessing the results returned

by subsystem calls. Argument binding, local variable allocation, and result passing mechanisms can be implemented using a sectioned stack. The sectioned stack allocates all activation frames from a single segment. The management of such a stack segment can be handled by mechanisms similar to those used in the Burroughs B5500 and B6500 processors [Organick 71] or using the schemes described by Rotenberg in his thesis [Rotenberg 74]. An activation frame is created on each subsystem call and destroyed when the subsystem activation terminates. Parameters are passed and results returned through the activation frame. The activation frame is also used by the subsystem to store its temporary variables. Much of this chapter is concerned with how the activation frame is formed and how it changes in response to the various subsystem transfer operations.

The basic addresses of the process model select locations in one of the three slots of the address space. Local addresses are mapped to locations in the activation frame. Static addresses are mapped to locations in the incarnation segment, while shared addresses are mapped to locations in the root segment. All addresses are checked to see that they lie within appropriate limits in the segments they reference. The root segment is partitioned, at a point which we denote as root RO, into a read/execute region and a read/write region. An instruction pointer which references the current instruction in the root segment is also associated with every subsystem activation. Figure 3-1 depicts the basic address space of a subsystem activation. The utilization of the three address space segments is also represented in the figure.

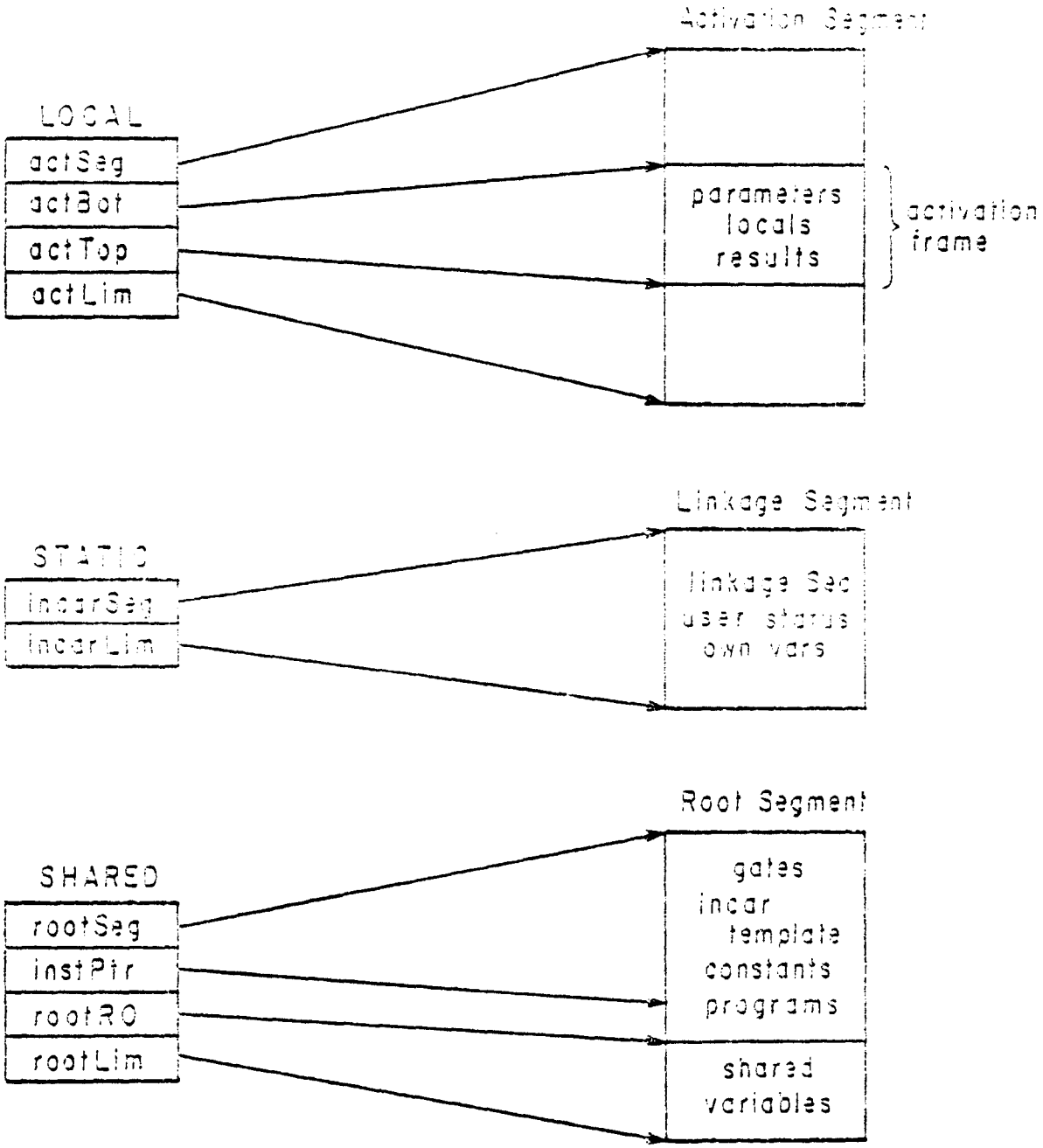

Figure 3-1: The Basic Address Space

Figure 3-2 is a flowchart of the basic addressing algorithm. The basic address consists of a tag and an integer index. The tag portion of the address selects one of the slots of the address space and the index specifies a location relative to the beginning of the indicated segment. The storage for the contents of the address space is assumed to be implemented as virtual storage at a higher level of the system. This means that, in general, any reference to a location in the basic address space may fail because of a virtual memory fault. Besides virtual memory faults, faults caused by addresses larger than the indicated segment will cause the basic addressing mechanism to fail.

Addressing and other faults causing basic processor operations to fail are called basic faults. The basic processor simply halts when basic faults are encountered. In Section 3.3, the basic process model is augmented to include exception reporting facilities which respond to basic faults and signalled exceptions.

3.2.2 External References

The addresses generated by basic machine operations are normally interpreted in the current address space. The basic addressing model does not provide for referencing objects outside of the three segments bound to the current address space. In order to reference one subsystem from another or to share objects between subsystems, some facility for referencing external objects must be defined. The external reference mechanism must protect the interests of subsystems by being able to control accesses to external objects.

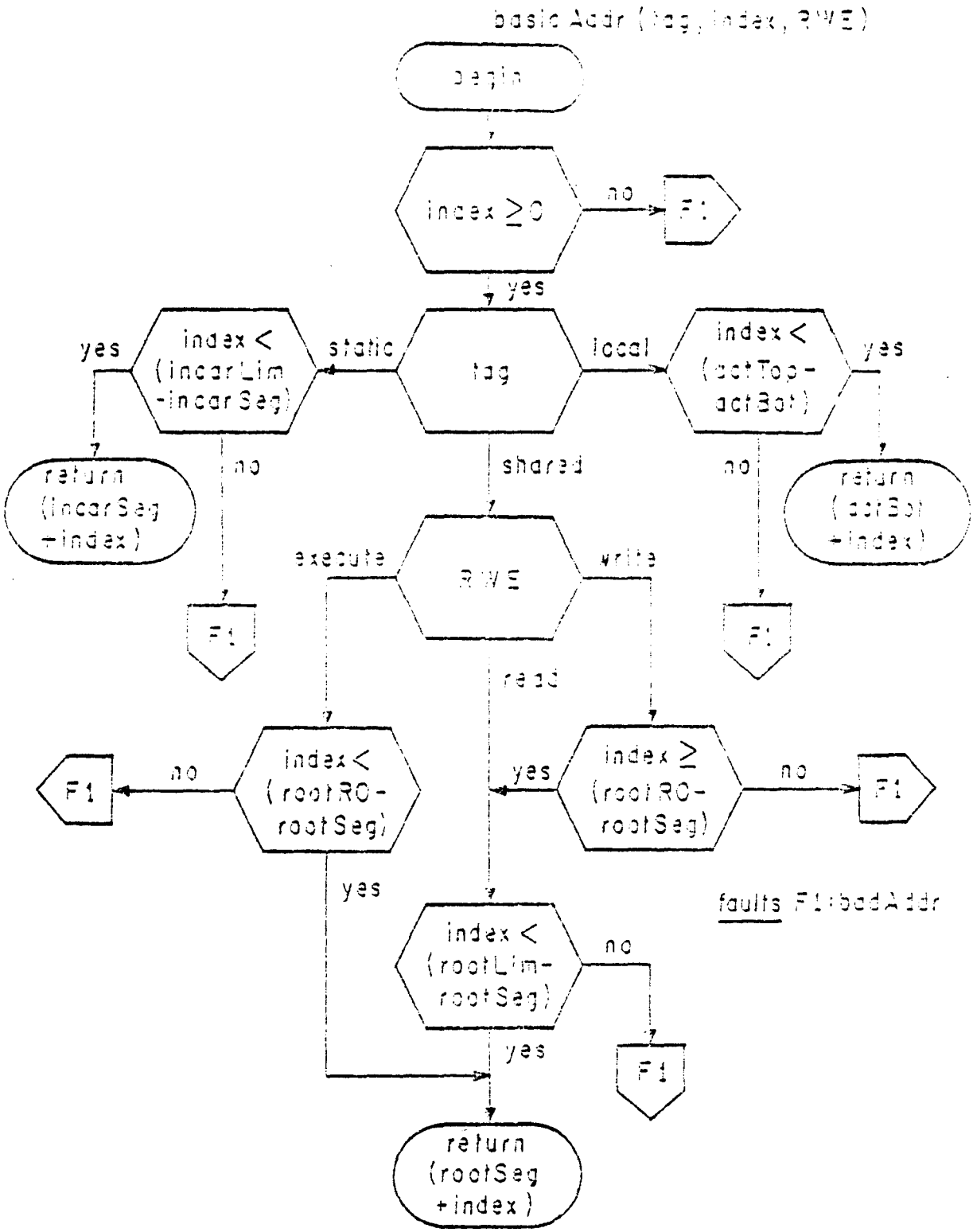

Figure 3-2: Basic Addressing

Capability based addressing, in which protected or sealed pointers are stored in the basic address space, provides a suitable external reference mechanism. Capability systems have been discussed at length by many authors [Dennis 66, Fabry 74, Lampson 71, Redell 74, Jones 73, Walker 73]. Various extension mechanisms, based on capabilities, permit protected subsystems to define and control access to new types of objects [Redell 74, Morris 73, Lindsay 73, Jones 73, Ferrie 74, Gligor 76].

There are other methods for referencing and controlling access to external objects. For example, access control lists can be used to validate unprotected external references [Schroeder 72, Saltzer 75]. Access list schemes check that the accessor is among the authorized users of the external object. The process model assumes the existence of an external reference mechanism, but leaves unspecified how external references are represented and validated. Either a capability or an access list system would be satisfactory. However, the process model is biased towards a capability based external reference system.

Given an external reference mechanism which permits objects outside of the current address space to be referenced and accessed, we can consider references to address spaces other than the current address space. Operations for creating external references to address spaces and for accessing the contents of other address spaces are supplied by the basic processor. The creation and manipulation of address space references must be closely controlled for several reasons. It must not be possible to make use of references to address spaces which no longer exist Because access to the address space of a subsystem activation licenses the

accessor to read and update the private variables of the subsystem, it must not be possible for user programs to create references to arbitrary address spaces. Basic processor operations for creating external references to address spaces are limited to an operation for creating a reference to the current address space. Address space references, like other external references, can be passed as parameters from one subsystem activation to another. The operations available on non-local address spaces include: 1) referencing the storage segments of the address space, 2) reading the instruction pointer, and 3) reading the subsystem name (root segment-id).

3.2.3 The Process Base

When a process is not being executed by a processor, the information defining the process and its current state is stored in the process base. The process base must be present in real memory before a processor can be assigned to the process. The basic process base contains a processor state block and a stack of suspended subsystem activations known as the subsystem activation stack. The activation stack contains the bindings for the address spaces of suspended subsystem activations. The current address space is maintained in a separate set of registers and not on the top of the activation stack. The process state also includes some general purpose accumulators and the registers which define the current address space.

Since we are not concerned in this thesis with processor multiplexing, we assume that changes to the process state are reflected directly in the process base instead of the registers of the processor currently

bound to the process base. Note that since the entire process base is assumed to be in real memory, the basic processor will not encounter virtual memory faults when accessing the process base. Figure 3-3 depicts the contents of the basic process base. The finite size of the activation stack implies that subsystem calls, which suspend the current address space and create a new current address space, may fail because the activation stack is full. The augmented process model introduces mechanisms which allow the top region of the activation stack to be in the process base while the rest of the stack is stored in non-fixed virtual memory segments.

3.2.4 Subsystem Call

The subsystem call operation initiates an operation of the called subsystem. The parameters of the subsystem call operation are 1) the subsystem to be activated, 2) the gate of the called subsystem to receive control, and 3) the index in the current activation frame of the beginning of the actual parameters for the called subsystem. We assume that the invoking subsystem prepares the actual parameter vector for the called subsystem at the high end of its activation frame. The parameters to the called subsystem are passed by value or by means of external references passed by value. The subsystem call operation saves a description of the current address space, including the instruction pointer, on the activation stack and then creates a new current address space for the called subsystem activation. The instruction pointer of the calling address space is left pointing to the call instruction.

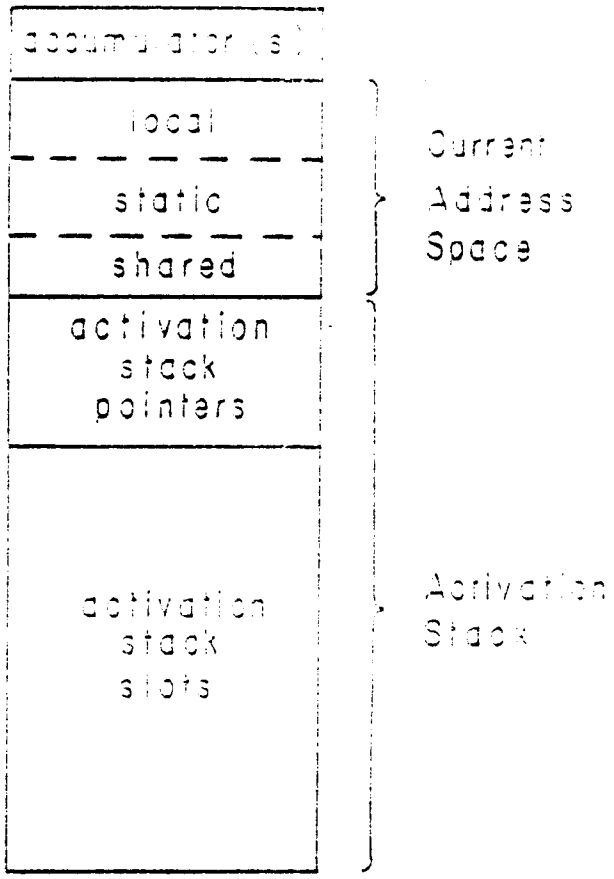

Figure 3-3: The Basic Process Base

The call operation creates a new current address space for the called subsystem. The new root, incarnation, and activation segments, as well as the new instruction pointer, must be determined and bound to the current address space. If subsystem gates are clustered at the beginning of the subsystem's root segment, the instruction pointer for the new current address space can be set to the gate index specified by the second parameter of the call operation. For simplicity, we will assume that all subsystems have the same number of gates. This limitation can be circumvented in a variety of ways which will not concern us here.

The activation frame for the called subsystem is derived from the caller's activation frame. The new activation frame should contain just the parameters of the call. If the parameters are prepared by the caller at the top of its activation frame, that portion of the caller's activation frame can become the initial activation frame of the new current address space. The called subsystem will begin execution with an activation frame containing just the parameters of the call. An initial workspace for the called subsystem could be allocated and initialized by the call operation. However, extra work at call time as well as the extra complexity to determine the workspace size make it unattractive to allocate the callee's workspace during the call. Instead of mapping the parameters into the callee's address space, the subsystem call operation could copy the parameter vector. This solution, attempted in CAL [CAL 69b] and HYDRA [HYDRA 74, Wulf 76] greatly increases the overhead of the subsystem call operation.

The root and incarnation segments for the new address space are derived from the the "subsystem" parameter of the call operation. The

"subsystem" parameter is an external reference which specifies either the root segment or the incarnation segment or both depending of the specific implementation. We explore the various cases in the following paragraphs.

A subsystem can be identified with its root segment where the code for the subsystem, the shared data, and the shared external references are stored. If the subsystem reference provides only a reference to the subsystem's root segment, the process-local incarnation segment must be located by searching a process-local "known-subsystem table". Systems which select the called subsystem by specifying only the root segment include the Chicago Magic Number Computer [Fabry 68], the Plessey System 250 [England 74], the HYDRA system [Wulf 74], and CAL [Lampson 76]. Of these systems, only CAL supports incarnation storage.

Instead of specifying the root segment, the subsystem reference of the call operation can specify the incarnation segment of the target subsystem. The incarnation segment, having been initialized when the subsystem was made known, can contain an external reference to the root segment. In Multics, the call operation specifies the incarnation segment (linkage section) of the callee. The linkage section contains a reference to the root segment of the subsystem [Daley 68]. One consequence of this approach is that every subsystem must have an incarnation segment to store the process-local references to the incarnation segments of the subsystems it calls and through which it can receive control. In the CAP system [CAP 76b, Needham 72], the external references used to identify the called subsystem specify both the root and incarnation segments. Subsystems which don't need incarnation storage do not

have incarnation segments in CAP.

Like the mechanism for implementing external references, the mechanism by which the "subsystem" parameter specifies the root and incarnation segments is left unspecified in the process model. It is assumed however that the mechanism for identifying the root and incarnation segments may fail to produce results if the subsystem is not known to the process.

A subsystem call pushes a description of the current address space onto the activation stack and then constructs a new current address space. The root and incarnation segments, as specified by the "subsystem" parameter of the call operation, are bound to the current address space. The new activation frame is defined to extend from the beginning of the parameters to the top of the caller's activation frame. Figure 3-4 illustrates how the activation frame for the new current address space is derived from the caller's frame and the parameter index of the call. Figure 3-5 depicts the effect of a call on the activation stack. Note that in the figure, the current address space is represented as being at the top of the activation stack instead of in a special set of registers. Figure 3-6 is a flowchart of the algorithm of the call operation. Manipulations of the activation stack are represented by qualified occurrences of the module 'actStk' which implements and maintains the representation of the subsystem activation stack. (Note: the notation used in the flowcharts is drawn partially from the Euclid syntax [Euclid 77].) 'ActStk.full' tests for overflow of the activation stack while 'ActStk.push' adds the description of an address space onto stack. The current address space is referenced by the structured vari-

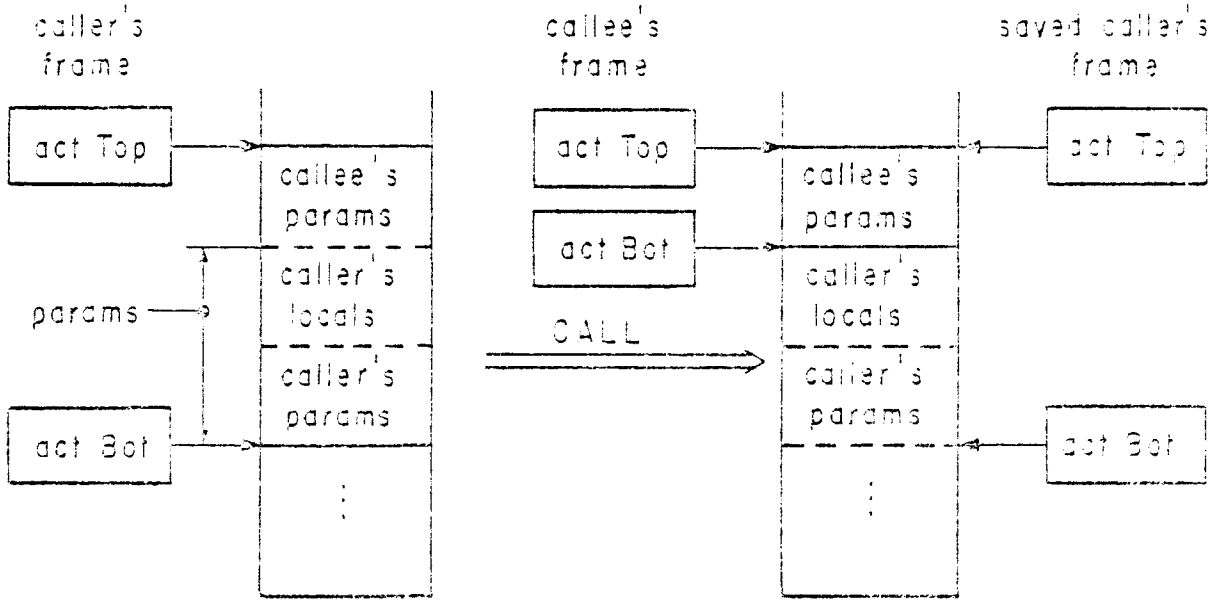

Figure 3-4: Subsystem Call -- Parameter Passing

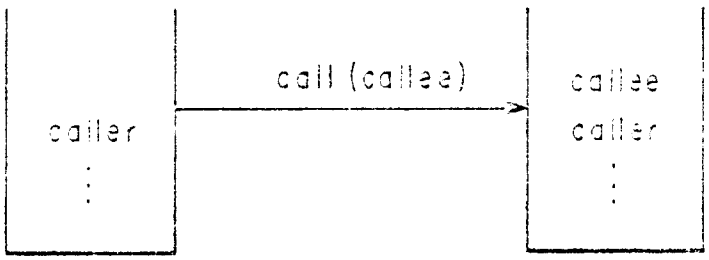

Figure 3-5: Subsystem Call -- Activation Stack

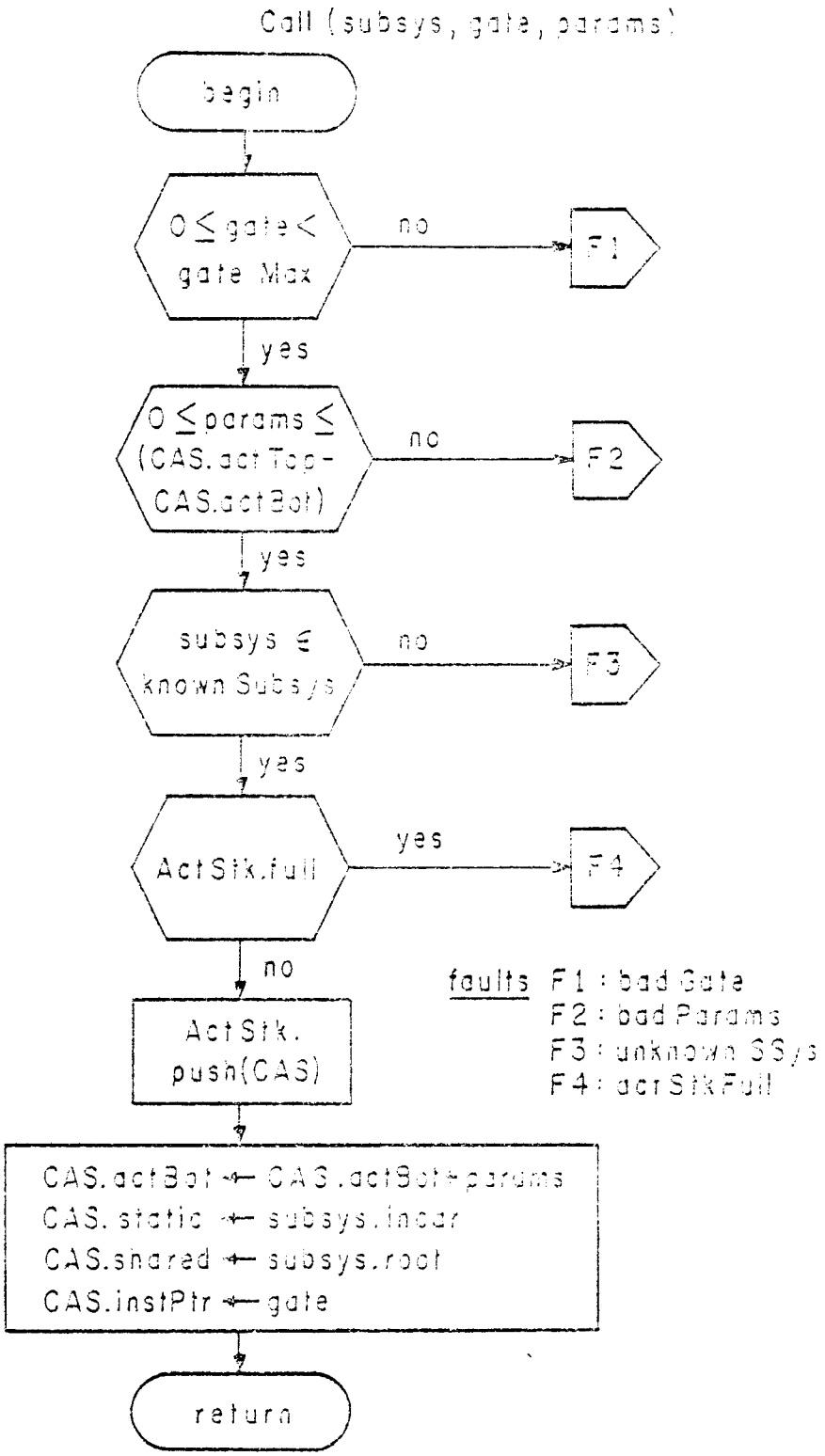

Figure 3-6: Subsystem Call -- Algorithm

able 'CAS'. Note that the return at the end of the flowchart causes control to pass to the called subsystem. It should not be confused with the subsystem return operation which is discussed next.

3.2.5 Subsystem Return

The subsystem return operation terminates the executing subsystem activation and continues the topmost suspended subsystem activation. Result values can be transmitted to the calling subsystem by adjusting the activation frame of the caller to include the results. The return operation does not take any parameters. Before returning, a subsystem shrinks its activation frame to include only the results (plus the original parameters). The results to be returned to the caller are assumed to occupy the entire activation frame. The return thus restores the activation frame of the caller to its state before the call, except for any results which may have pushed into the top of the frame. Figure 3-7 illustrates the adjustments to the saved activation frame which pass results from the callee to the caller.

The subsystem return operation removes the top address space from the activation stack and binds it to the current address space. Figure 3-8 shows how subsystem return affects the activation stack. The instruction pointer in the caller's address space is advanced to reflect the completion of the operation which was under way when the address space was suspended. To keep things simple, we assume that all instructions are the same length. Normally the operation under way will have been a call operation. In the augmented process model, however, basic faults may leave the instruction pointer of a suspended address space at

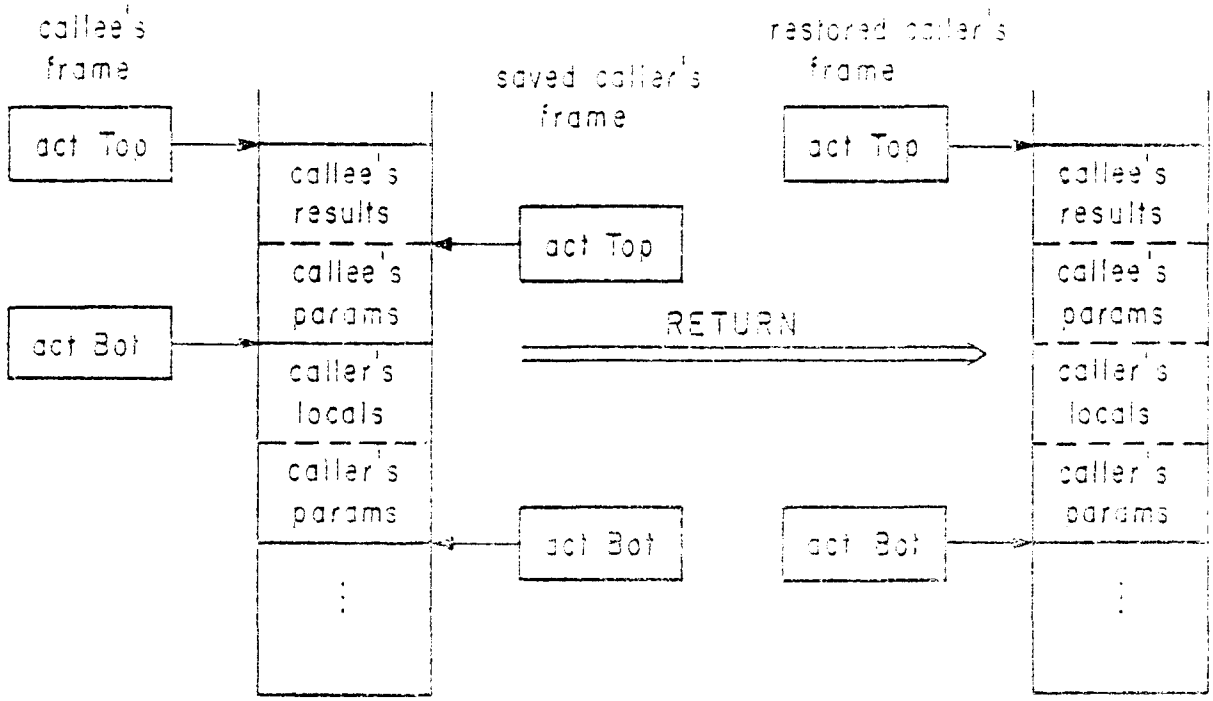

Figure 3-7: Subsystem Return -- Result Passing

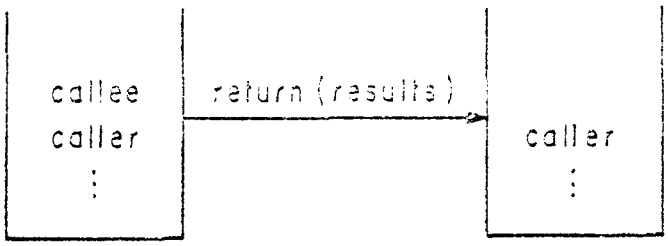

Figure 3-8: Subsystem Return -- Activation Stack

any operation which might cause a fault.

Return adjusts the activation frame of the caller to include the results returned from the terminating callee. The activation frame of the reactivated subsystem has the same base as before the call but the top of the frame is set to include the returned results. If the activation stack is empty, the return operation fails without terminating the subsystem executing the return operation. Figure 3-9 gives the algorithm of the subsystem return operation.

3.2.6 Allocating Activation Storage

The subsystem call operation initializes the new activation frame to contain just the parameters of the call. Most subsystems will need storage for local and temporary variables. The basic processor includes an operation for allocating and initializing activation frame storage and an operation for determining the current size of the activation frame. The operation frame size simply stores the size of the current activation frame into one of the processor's general purpose accumulators. The allocate operation takes one parameter: the new size of the activation frame. If the new frame size is larger than the current frame size, the current activation frame is extended at the top and the new storage is initialized to zeros.

If the new activation frame size is smaller that the current size, the activation frame is truncated from the top. However, the activation frame is not permitted to become smaller than it was when it was created. This restriction avoids problems stemming from the fact that the

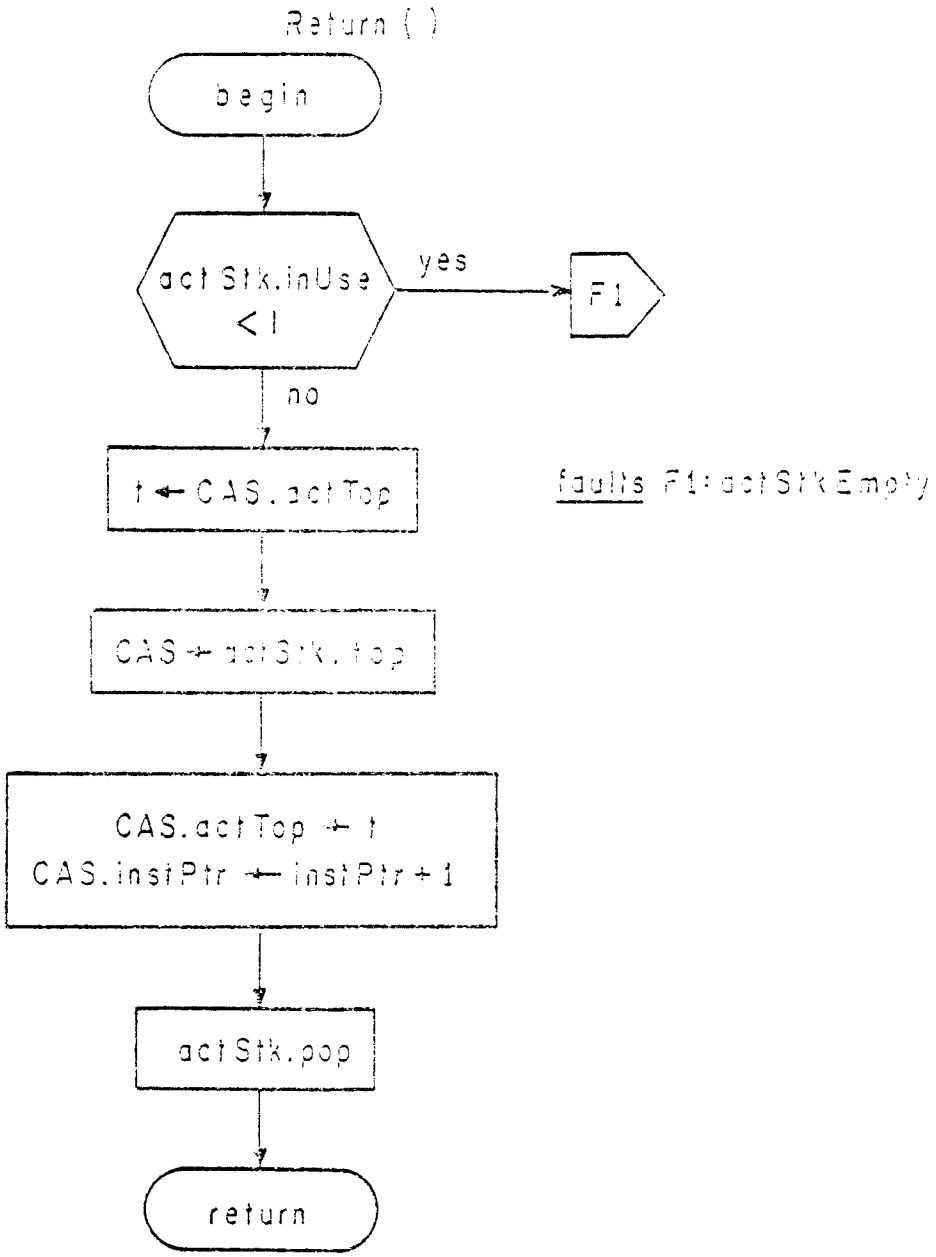

Figure 3-9: Subsystem Return -- Algorithm

caller's saved activation frame includes the parameter area which became the initial activation frame of the callee. Accesses to a suspended address space can be make using non-local address space references which use information saved in the activation stack. If the callee could free and the re-allocate the parameter area of its activation frame, non-local references to the top region of the caller's activation frame would reference storage no longer associated with the caller. The problem is circumvented by preventing the callee from freeing the shared portion of its activation frame.

The allocate operation will fail if the new frame size would exceed the space left in the activation segment as well as if it would reduce the frame to less than its original size. Allocate will also fail if it encounters a virtual memory fault while initializing new activation storage. If the allocate operation fails because of a virtual memory fault, it can be re-executed after the virtual memory fault has been handled. The allocate operation is re-executable because its parameter specifies the the desired size of the activation frame. If the parameter specified the change to the activation frame size, a partially executed allocate operation could not be re-executed without first changing the parameter value to reflect the progress made before the virtual memory fault. Instead, the current activation frame pointers in the process base record the progress made by a failed allocate operation. Note that the allocate operation does not conform to the ideal of no side effects for failed operations and must be carefully designed to be re-executable following a failure.

Figure 3-10 illustrates the changes to the process base caused by a successful allocate operation. Figure 3-11 gives the algorithm of the allocate operation. Note the use of the function 'Seg.write' to initialize newly allocated activation storage and the use of the 'actTop' pointer to record the progress made.

3.2.7 Non-Local Address Space References

The basic process model supports non-local addressing thru the use of external references to suspended address spaces. In the basic process model, any subsystem activation can create an external reference to its own address space. The reference can then be passed as a parameter to other subsystem activations. The representation of the non-local address space reference is left unspecified. However, we can assume that the information contained in the reference is sufficient to locate the address space in the activation stack and that accesses to non-local address spaces are interpreted using the information in the activation stack. No provision for restricting access in the address space is included in this simple process model.

The management of non-local address space references is tricky because, if no precautions are taken, the reference may outlive the address space to which it refers. One way to validate a non-local address space reference is to assign a unique identifier to each address space as it is created and to store this identifier with the address space in the activation stack. If the non-local reference contains the address space identifier and its activation stack index, the use of obsolete address space references can be detected. Note that attempts to access

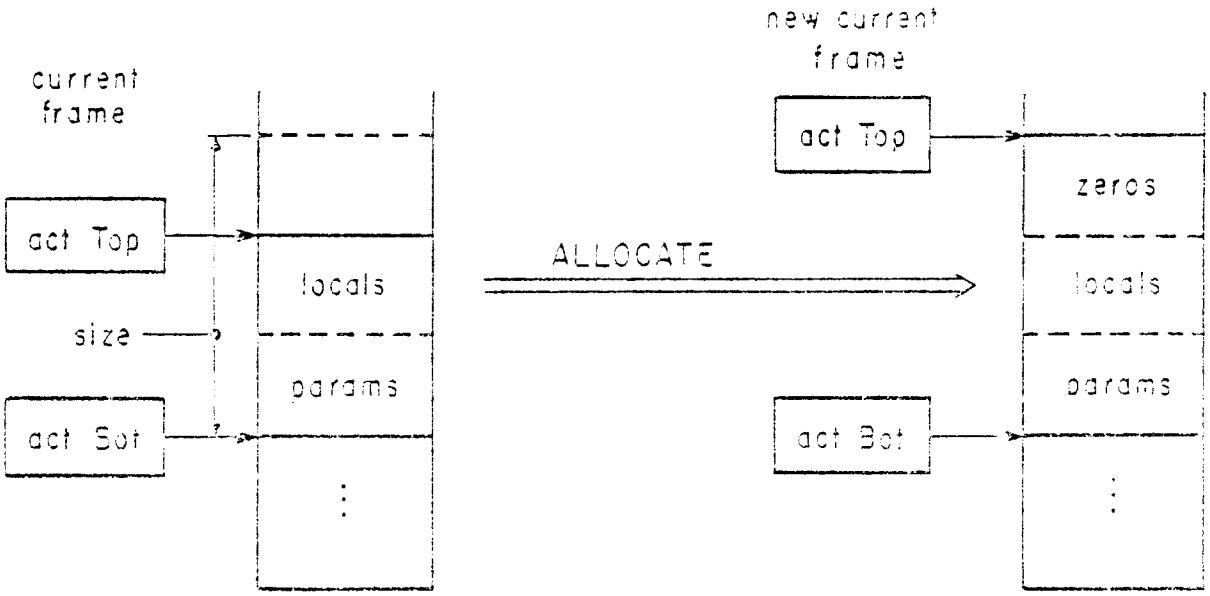

figure 3-10: Allocate Activation Frame

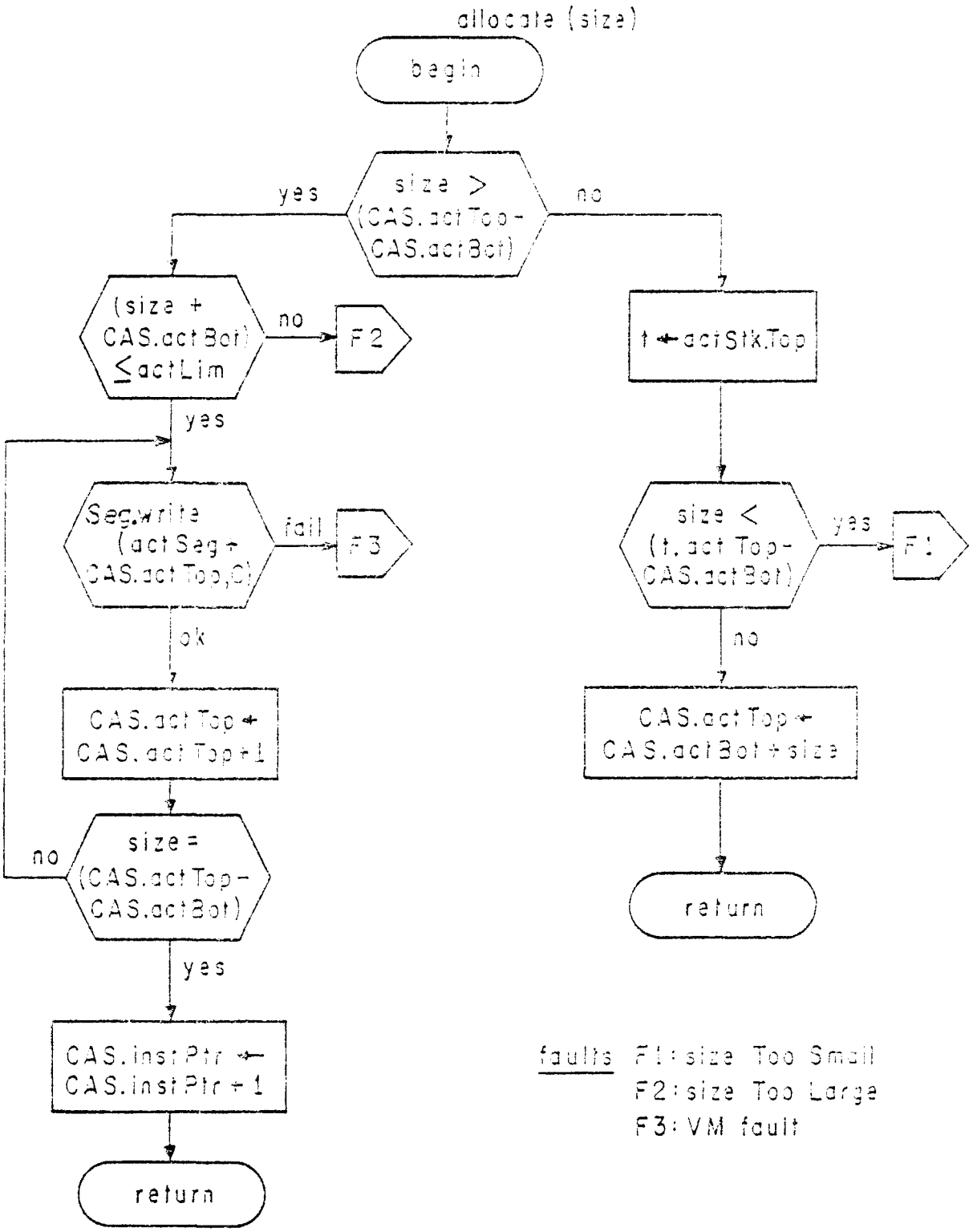

figure 3-11: Allocate Algorithm

the contents of a non-local address space can cause a basic fault if the address is out of bounds, if the address space is not in the activation stack, or if a virtual memory fault is encountered.

3.3 The Augmented Process Model

The basic process model makes no provisions for reporting faults which prevent the completion of basic process operations. In this section, the basic process model is augmented to include a mechanism for dealing with basic faults and an operation for signalling an exception from a subsystem activation. Subsystem transfer operations for terminating an exception episode are also added to the basic process repertoire of instructions.

The basic fault mechanism of the augmented process model activates the distinguished exception processor subsystem and communicates information describing the fault. The basic fault mechanism must succeed in activating the exception processor without encountering any problems which would prevent the exception processor from being successfully activated. If the augmented processor encounters basic faults attempting to process a basic fault, there is no continuation for the process. Note that once control is passed to the exception processor, additional faults can be processed. However, the basic fault mechanism must be designed to avoid virtual memory and subsystem call faults while attempting to activate the exception processor. It is still possible that the processor hardware is broken. In this case, the processor should halt, leaving the process base in a state from which other processors might carry on the process.

In response to basic faults and exception signals, the augmented processor activates the exception processor subsystem whose identity is stored in the process base. The distinguished status of the exception processor is also used to authorize the privileged subsystem transfer operations used to terminate an exception episode. Operations for retrying failed operations, exiting to any location in the invoker, and for aborting the invoker's activation allow the exception processor to terminate its own activation in ways not permitted for other subsystems. These termination operations are too powerful to be used directly by normal subsystems. It is assumed that the exception processor will control the use of the special terminations on behalf of the invoker of the operation which caused the exception processor to be activated. Chapter Four discusses the details of the protocols which can be used by the exception processor to control the use of the special terminations.

The augmented processor model also expands the implementation of the activation stack to allow the top portion of the stack to reside in the in-core process base while the rest of the stack is stored in virtual memory segments. The activation stack management facility interacts with the non-local reference mechanism of the basic process model. Because the address space referred to by a non-local reference may not be in the active portion of the activation stack, non-local references may fail in new ways in the augmented process model. The details of the activation stack management facility are discussed in Section 3.3.4.

3.3.1 The Basic Fault Mechanism

Whenever a basic (or augmented) processor operation is unable to complete normally, the basic fault mechanism is used to terminate the operation and to activate the exception processor. The exception processor must have at least two gates: one for basic faults and one for signalled exceptions. Two gates are necessary because the parameters describing the exception are transmitted differently in the two cases. The parameters describing signalled exceptions can be prepared by the signaller in its (virtual memory) activation frame. The processor, however, cannot risk a virtual memory fault trying to store the information describing another basic fault. For this reason, the fault data for basic faults is stored in the wired down process base. The exception processor can access the fault data in the process base by using an external reference which it obtains at initialization time.

In response to the detected failure of a basic (or extended) processor operation, the information describing the failure must be saved in the process base and the exception processor must be activated. The process base is extended to include room to store the identity of the exception processor and space for the parameters of a basic fault. The identity of the exception processor can be stored as an external reference to the exception processing subsystem. The fault parameters include 1) a non-local reference to the invoker's address space, 2) a signaller-id representing a related set of processor faults, 3) the exception code identifying the particular fault, and 4) the exception message giving additional information about the fault. Figure 3-12 depicts the modified process base as expanded to support the implementation of

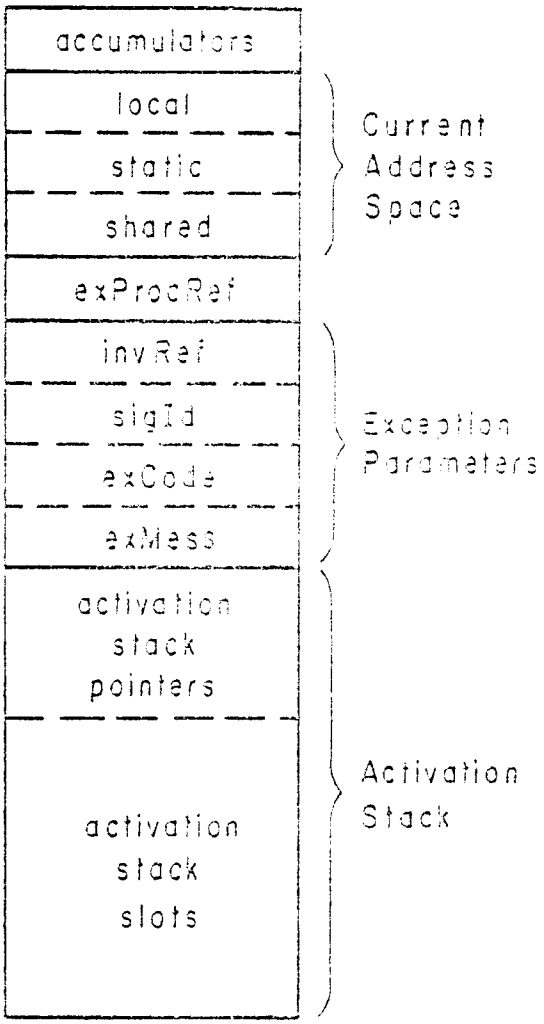

figure 3-12: The Process Base (Augmented)

basic fault processing and signal operations.

Figures 3-13 and 3-14 illustrate how the activation frame for the exception processor activation and the activation stack are manipulated by a basic fault. Note that following a fault the exception processor's activation frame is initially empty. Figure 3-15 is a flowchart of the basic fault mechanism. The augmented processor, in response to a basic fault, pushes the current address space onto the activation stack. The instruction pointer of the invoker's address space is not incremented and therefore it continues to reference the failed operation. We shall assume at this point that pushing the current address space onto the activation stack will not cause the stack to overflow. In Section 3.3.4 the notion of activation stack overflow is defined so as to always leave room to process the worst case basic fault scenario without running out of stack space.

After saving the invoker's address space, the processor stores the signaller-id, exception code, and exception message into the in core process base. A non-local reference to the invoker's address space is also generated and stored in the process base. Once the information in the process base has been updated, the exception processor can be activated. Using the subsystem reference in the process base, an address space for the exception processor is constructed. The subsystem reference in the process base must not fail to produce root and incarnation segments for the exception processor. This means that the exception processor must always be known and ready to run. When the address space for the exception processor has been constructed, the exception processor is entered through its fault gate. Note that the root segment of

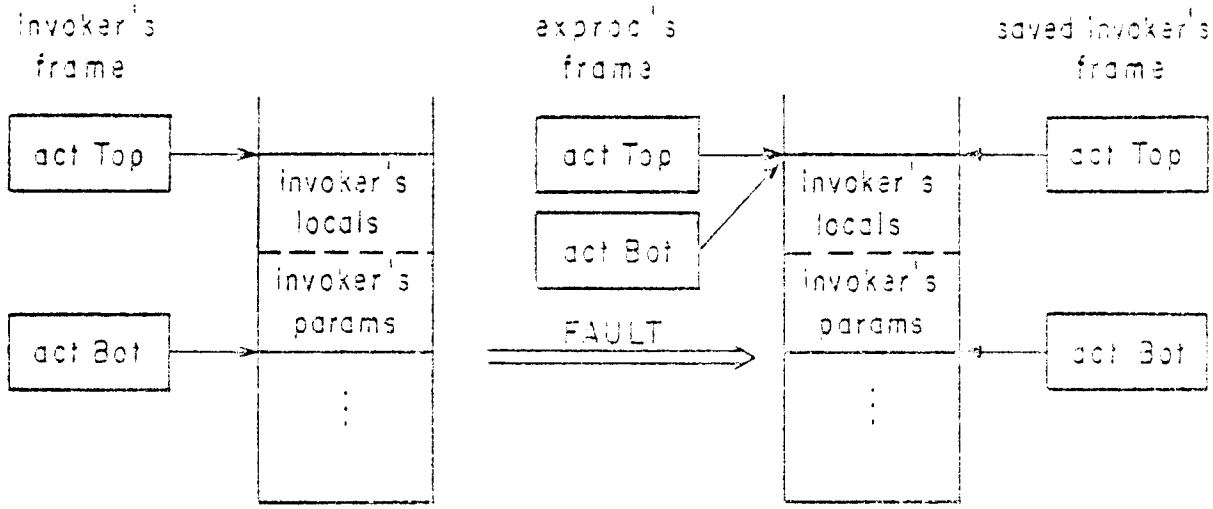

Figure 3-13: Basic Fault -- Activation Frame

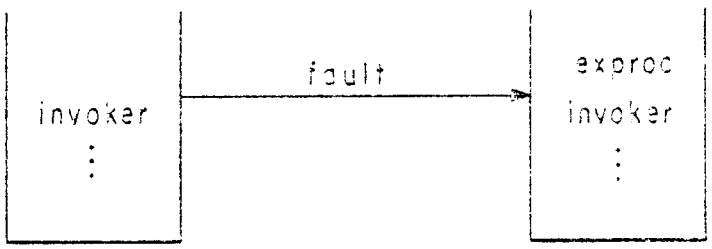

Figure 3-14: Basic Fault -- Activation Stack

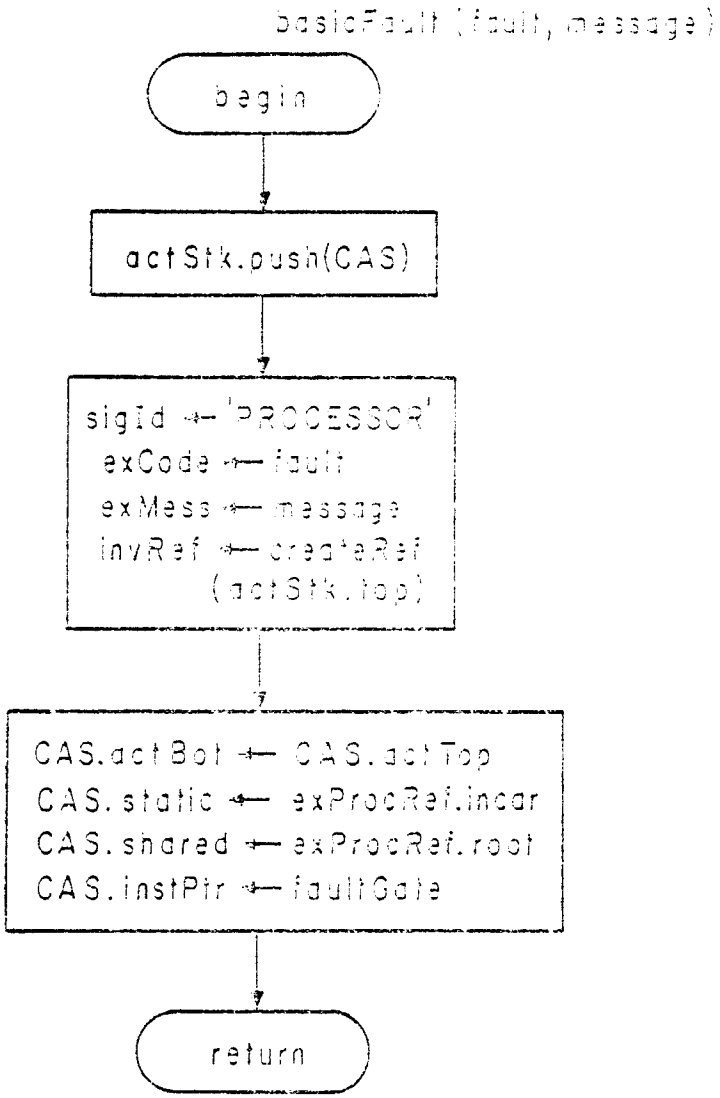

Figure 3-15: Basic Fault -- Algorithm

the exception processor must be fixed in real memory in order to avoid virtual memory faults in the code of the exception processor.

3.3.2 The Signal Operation

Signal is an augmented machine operation in the same class of operations as the subsystem call and return operations. The signal operation terminates the activation of the subsystem executing the signal and then activates the exception processor. The signaller, before executing the signal operation to report its own failure, prepares the exception parameters its own activation frame.

The signaller provided exception parameters, which include the exception code and exception message, should be placed immediately following the parameters of the call to the signaller (i.e. where the results would normally be found). The exception code is chosen by the signaller to distinguish among the different exceptions which it reports. The exception message provides details about the instance of the exception denoted by the exception code. The exception message can be of variable length. Since this is below the language level, the variableness of the length and format must be dealt with explicitly. For simplicity, we assume that the first word of the exception message contains the length of the rest of the message.

After preparing the exception code and message, the signaller should shrink its activation frame so that it contains just the signaller's original parameters and the exception parameters. The signal operation itself does not take any parameters. The exception param-

eters are assumed to occupy the space between the parameters of the call to the signaller and the end of the signaller's activation frame.

The signal operation, like the basic fault mechanism, activates the exception processor. Unlike the basic fault mechanism, the signal operation passes some of the exception parameters to the exception processor through the activation segment. The exception processor's initial activation frame contains the exception code and message generated by the signaller. The rest of the parameters must be generated by the processor and stored in the process base.

Besides activating the exception processor and passing the exception code and message through the activation segment, the signal operation must authorize the exception processor to access the invoker's address space. Also, the signaller-id must be passed to the exception processor. This information can be passed through additional slots in the process base. If the signal operation attempted to store the signaller-id and the invoker reference into the activation frame, a virtual memory fault might occur. Since the exception processor is authorized to access the exception information in the process base in order to deal with basic faults, it can also access the signaller-id and invoker reference in the process base. The exception processor, however, must be very careful to move the information out of the process base before another signal occurs.

Figure 3-16 illustrates how the signal operation forms the activation frame for the exception processor from the signaller's activation frame. Figure 3-17 shows that only the current address space (the pseudo top of the activation stack) is changed by a signal. Signal does not

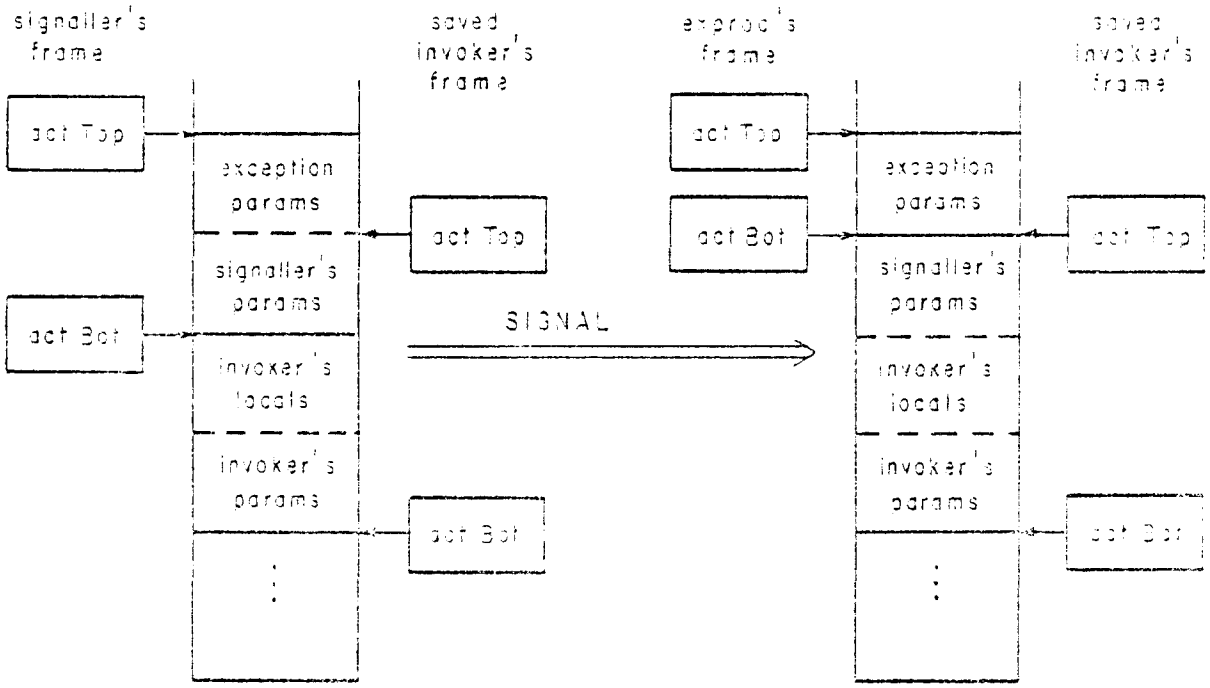

figure 3-16: Signal -- Exception Parameters

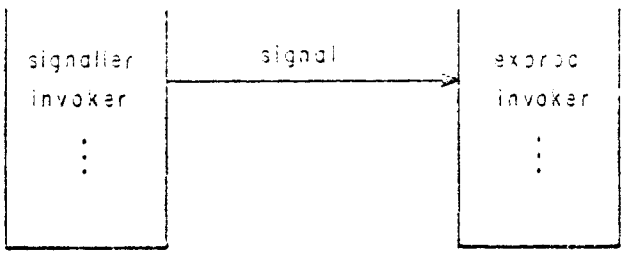

figure 3-17: Signal -- Activation Stack

push an address space onto the activation stack and, therefore, an activation stack overflow cannot occur on a signal. Figure 3-18 gives some of the details of the implementation of the signal operation.

3.3.3 Episode Termination Operations

The augmented processor implements three special subsystem transfer operations designed to support the termination of an exception episode. Only the exception processor is allowed to execute these operations. The retry operation returns control to the invoker at the failed operation, while the abnormal return operation returns control at an arbitrary address in the invoker's root segment. Although retry can be treated as a special case of the abnormal return, we distinguish them at this level because they are used to achieve different sorts of recovery from the exception. The abort operation terminates both the exception processor and the invoker.

Retry is like the return operation except that no results are passed and the instruction pointer of the saved address space is not incremented. The invoker's activation frame will be the same size as it was when the failed operation was executed. If the parameters to the failed operation have been updated by the handler or by the failed operation, the operation will be retried with the new parameters. The retry operation will fail if the activation stack is empty or if the current subsystem activation is not the exception processor. Figure 3-19 depicts the restoration of the invoker's activation frame to its state before the failed operation execution. Figure 3-20 is a flowchart of the action of retry.

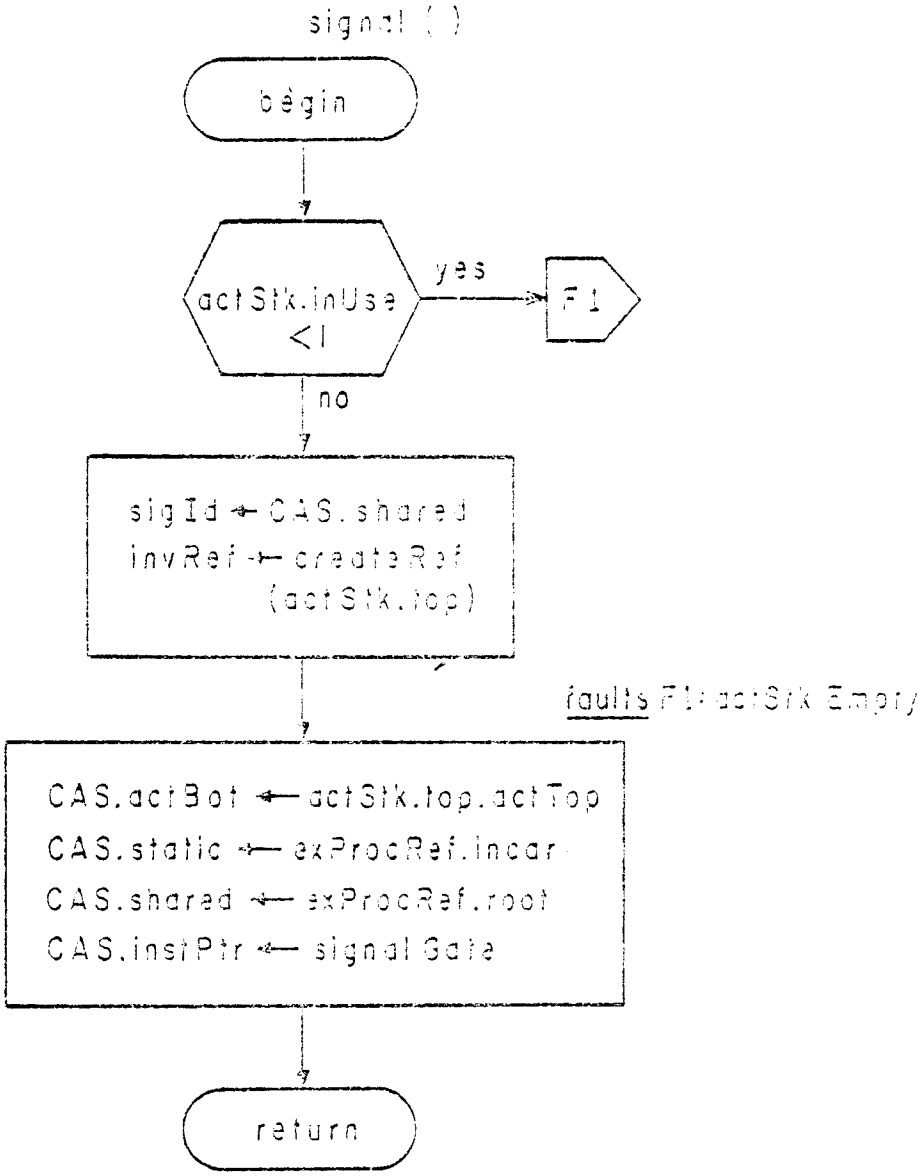

figure 3-13: Signal -- Algorithm

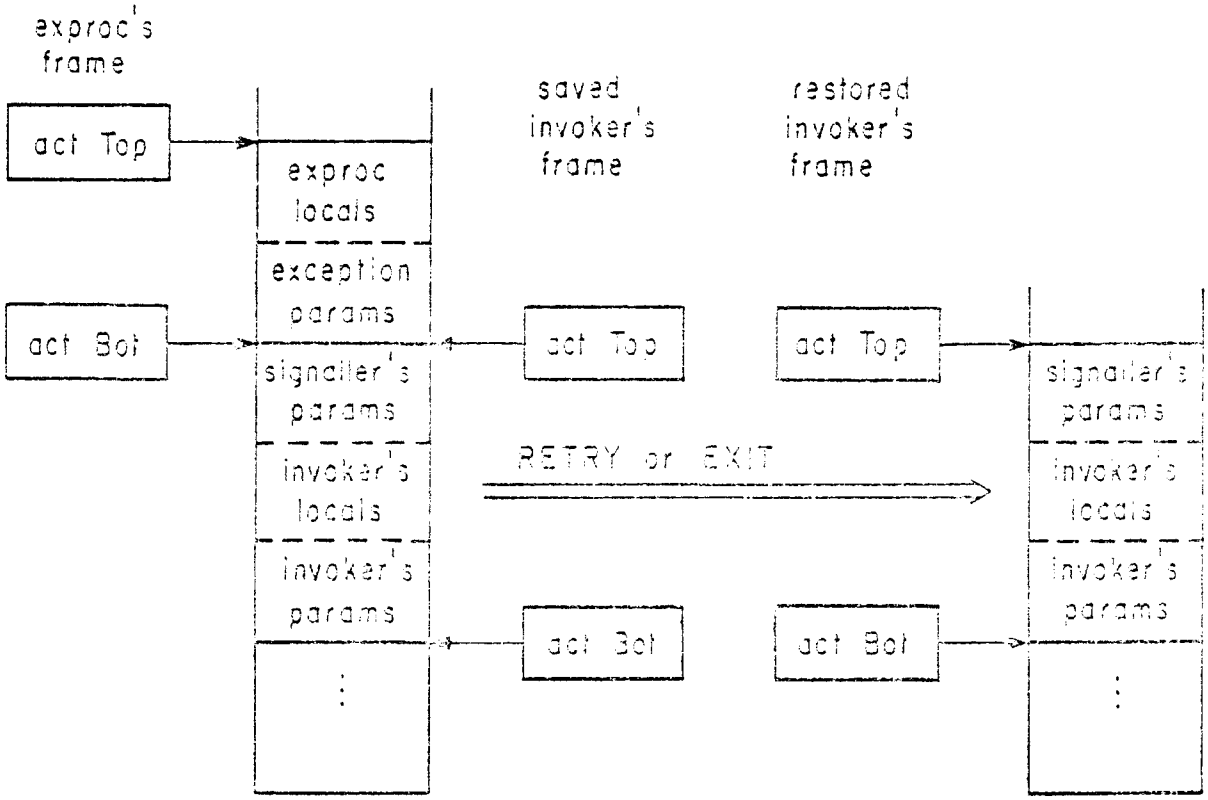

Figure 3-19: Retry -- Restore Invoker's Frame

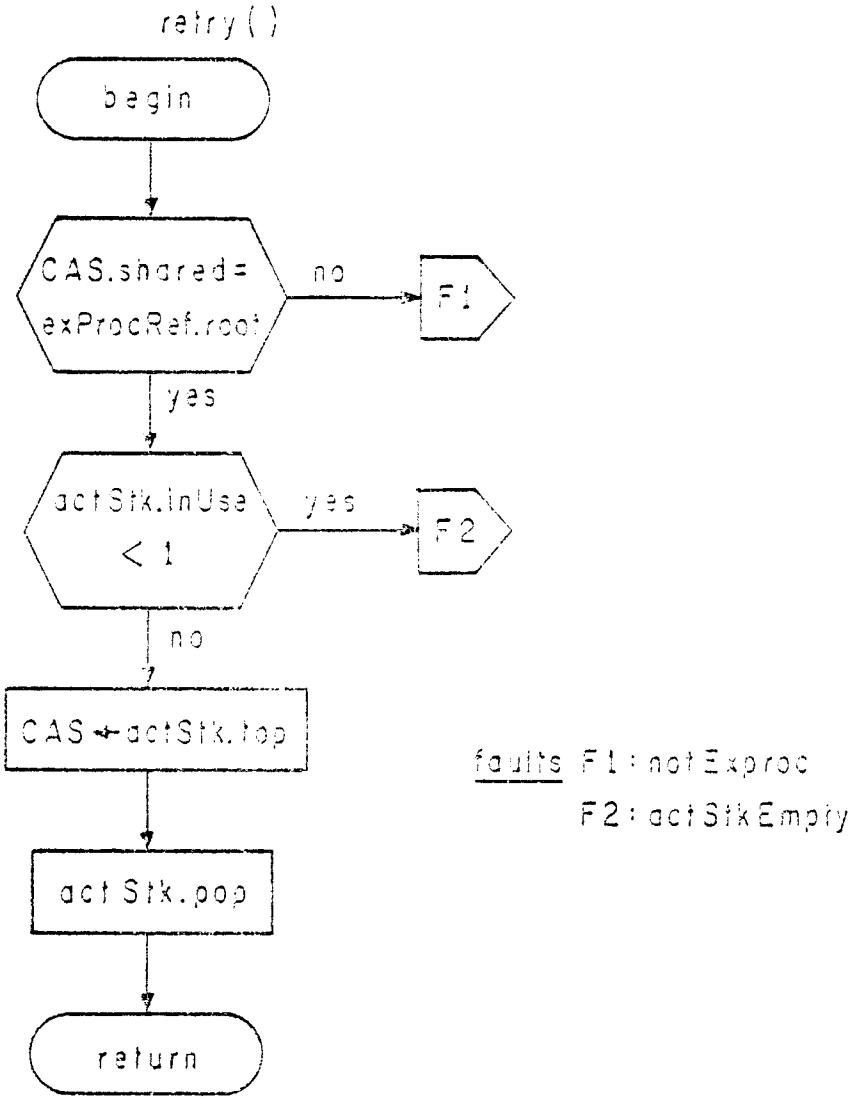

figure 3-20: Retry -- Algorithm

The abnormal return is similar to retry termination except that the instruction pointer of the invoker is arbitrarily repositioned. Abnormal return takes one parameter: the new instruction pointer value for the continuation of the invoker. The new instruction pointer value is checked to see that it lies in the executable region of the invoker's root segment. Like retry, abnormal return will fail if the activation stack is empty or if the current subsystem is not the exception processor. Figure 3-21 illustrates the algorithm of the abnormal return.

Abort termination causes the exception processor to be re-initiated after the invoker has been removed from the activation stack. The abort exception code and exception message are prepared by the exception processor and occupy the entire activation frame of the exception processor both before and after the abort operation. The single parameter of the abort operation is a flag indicating whether the signaller of the abort should be the invoker or the exception processor. Like retry and exit, only the exception processor is permitted to execute an abort. Also, there must be at least two entries on the activation stack to execute an abort.

Because abort terminates the invoker along with the current exception processor, the exception parameters must be moved from one place in the activation segment to another. Moving the exception parameters may cause a virtual memory fault. A progress indicator to indicate how much of the exception parameters has been moved will allow the abort operation to be retried following a virtual memory fault. Since the invoker is not going to be continued, the invoker's activation top pointer can be used as the progress indicator. The invoker's activation top ini-

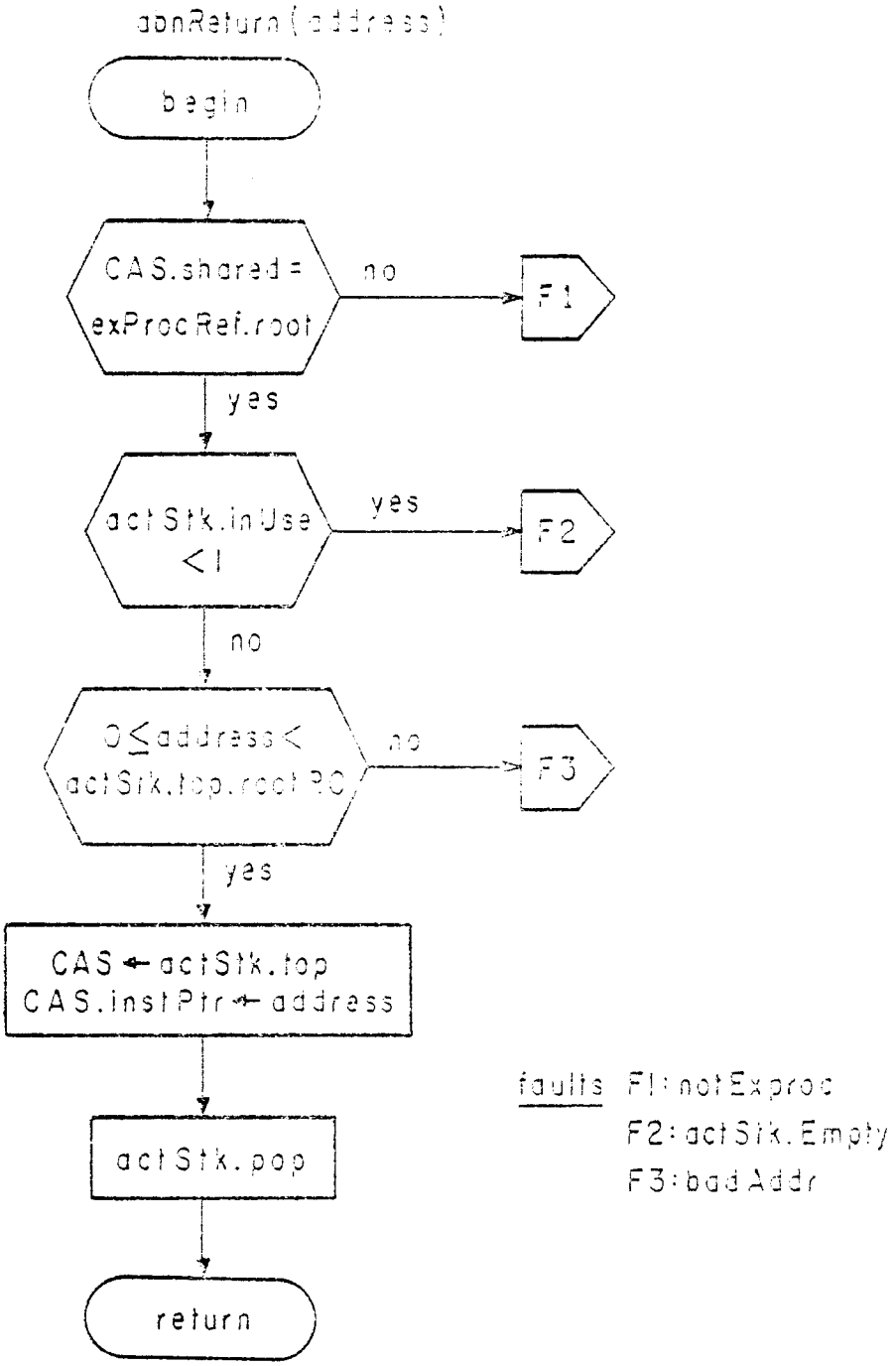

figure 3-21: Abnormal Return -- Algorithm

tially points to the base of the exception processor's activation frame. The abort operation increments the invoker's activation top after moving each word of the exception parameters. The copying is complete when the invoker's activation top equals the exception processor's activation top.

After moving the exception parameters, the abort operation stores a non-local reference to the invoker's invoker into the process base as the new invoker reference. The invoker's name is stored as the signaller-id if the parameter of the abort is False. Otherwise, the exception processor's name is stored as the signaller-id. The activation stack is popped and the exception processor is reactivated. The activation frame for the new exception processor contains the same values as the old exception processor frame except that they are at new locations in the activation segment. Figure 3-22 illustrates the copying of the exception parameters and the activation frame formation protocol while figure 3-23 shows how an abort removes the invoker's activation from the activation stack. Figure 3-24 is a flowchart of the actions of the abort operation.

3.3.4 The Augmented Activation Stack

We have assumed that, following a basic fault, there is always room to push the current address space onto the activation stack. But, we have also assumed that the activation stack is stored in a finite real memory buffer in the process base. It is important to limit the amount of fixed memory necessary to run a process. In this section, the management of the activation stack is extended to permit entries to be

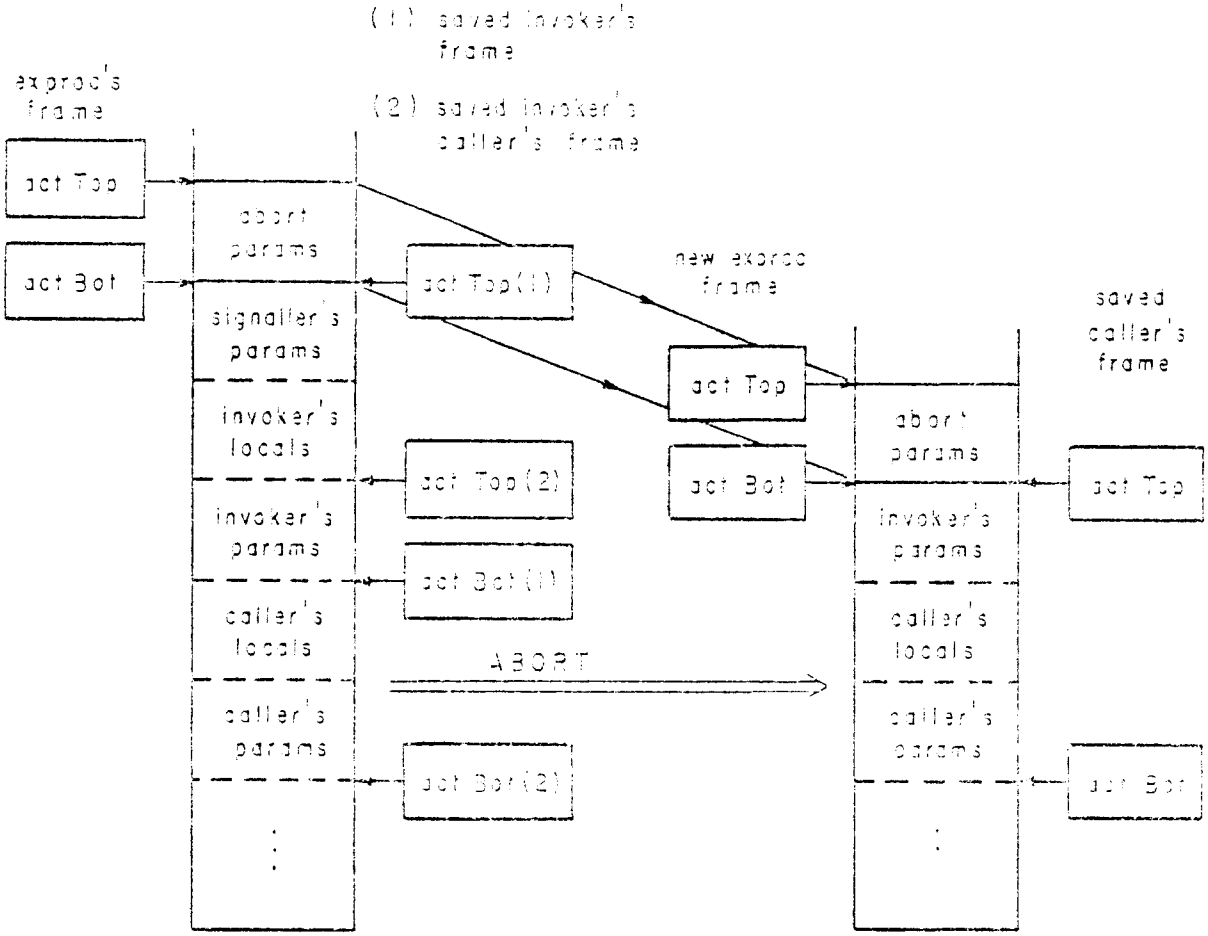

Figure 3-22: Abort -- Parameter Copy

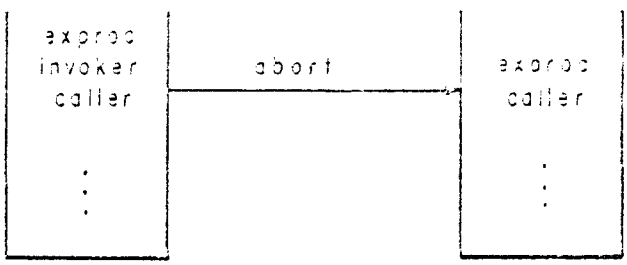

Figure 3-23: Abort -- Activation Stack

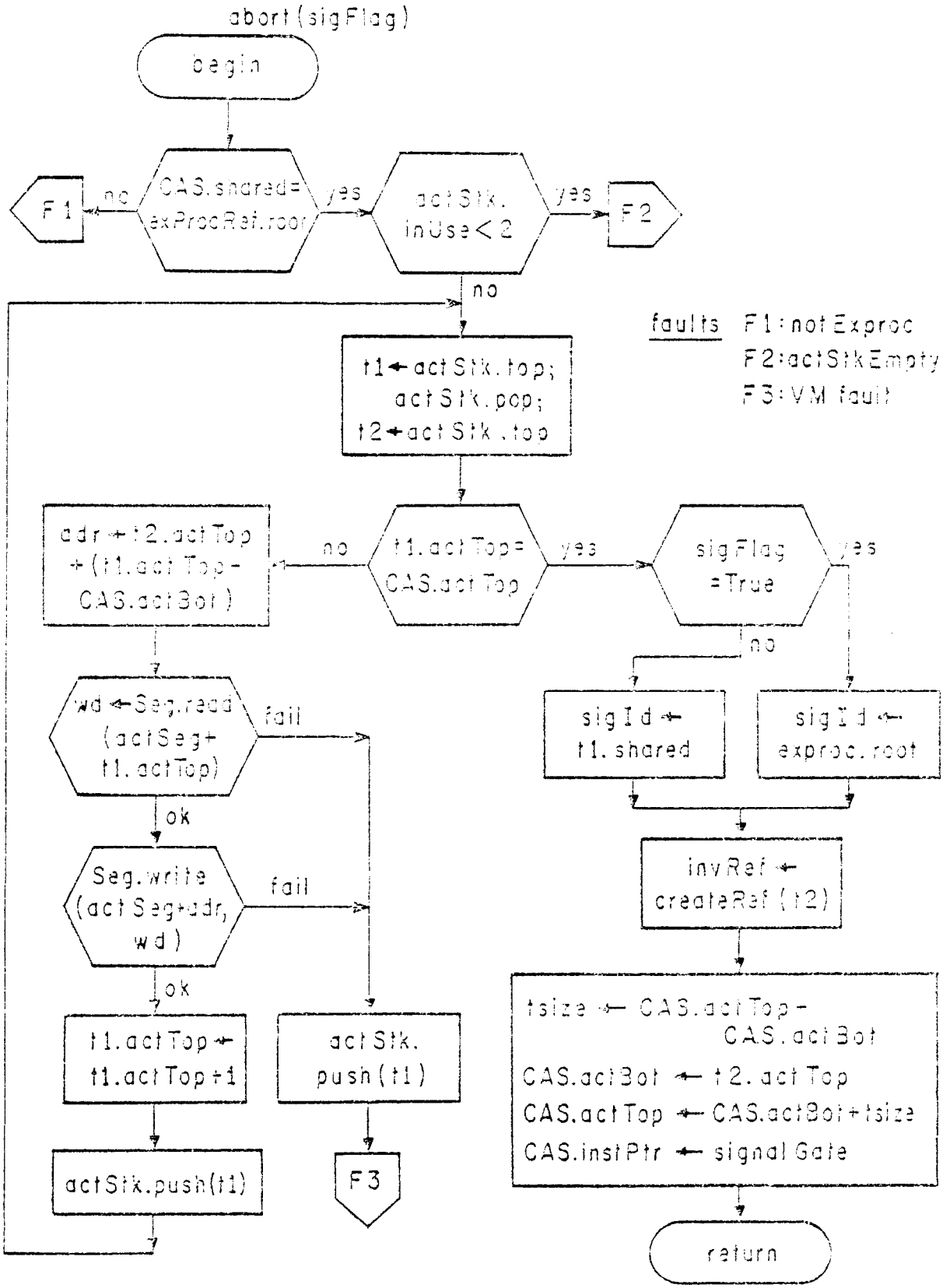

Figure 3-24: Abort -- Algorithm

removed and added at the bottom of the stack so as to limit the amount of fixed memory dedicated to the activation stack. Also, the notion of when the stack buffer is full is modified to permit basic faults to be processed without overflowing the buffer.

The activation stack in the process base can be implemented as a double-ended queue or circular buffer [Knuth 68, p.240]. The subsystem transfer operations will push and pop entries from the top of the queue, while special, privileged operations will be provided for unloading and loading entries at the bottom of the queue. Figure 3-25 illustrates the data structure used to implement the activation stack buffer. An activation stack full fault will be reported before the stack buffer is fully occupied and while there is still enough room to process the worst case scenario of basic faults.

The eventual handler of the stack full fault can use the privileged stack operations to unload entries from the bottom of the stack and save them in virtual memory segments. When the fixed stack buffer becomes empty, a related handler can load entries back into the stack buffer. The privilege of loading and unloading the stack buffer can be controlled by requiring the handler to present a special external reference (capability) or by recording the handler's identity in the process base.

Detecting and reporting that the activation stack buffer is full requires some care. The full fault cannot be allowed to occur during a basic fault sequence because information about the current fault would be lost. To accommodate faults when the stack buffer is logically full, some spare slots are reserved. The full fault only occurs on subsystem calls which occur after the activation stack buffer begins to use its

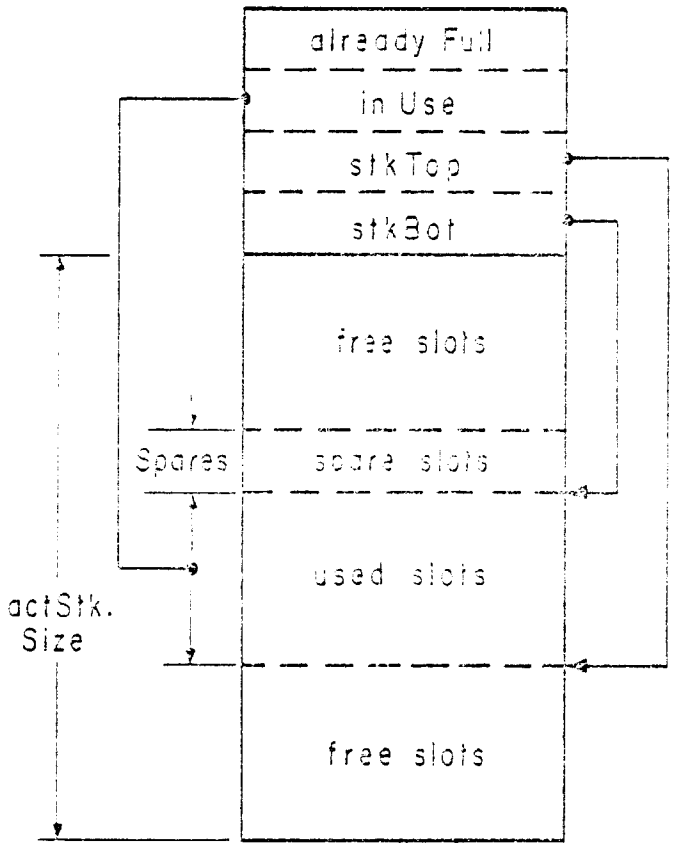

figure 3-25: Activation Stack Buffer

spare slots. Figure 3-26 gives the algorithm for testing whether the activation stack buffer should be considered full. Note that a flag is used so that the full condition is only reported once each time the buffer becomes almost full. We must be able to activate the exception processor and make the handler calls necessary to process the full fault without getting more full faults. Figure 3-27 shows how the full flag is reset when the stack retreats out of the full region. A worst case scenario of basic faults in which seven extra slots in the stack buffer are needed is described in Chapter Four.

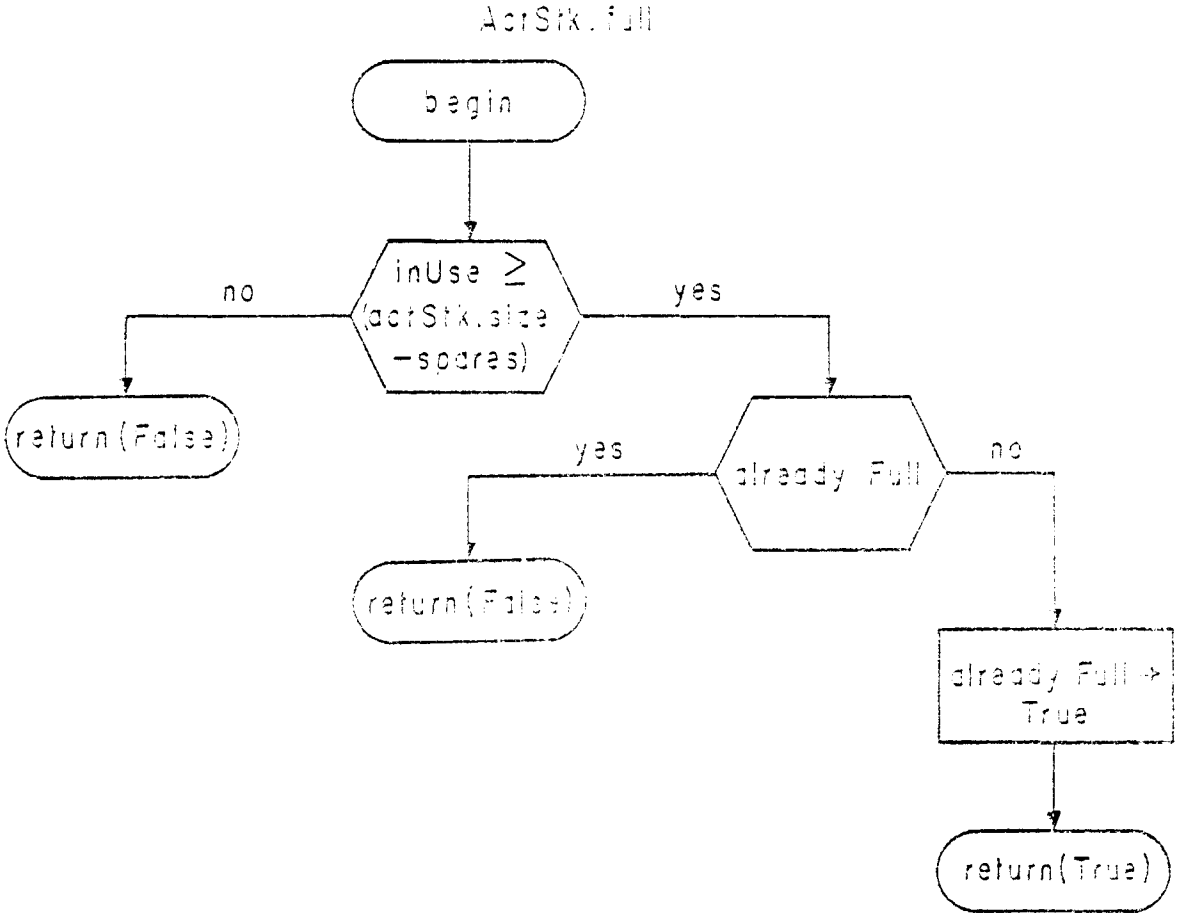

Figure 3-25: Activation Stack Full Test

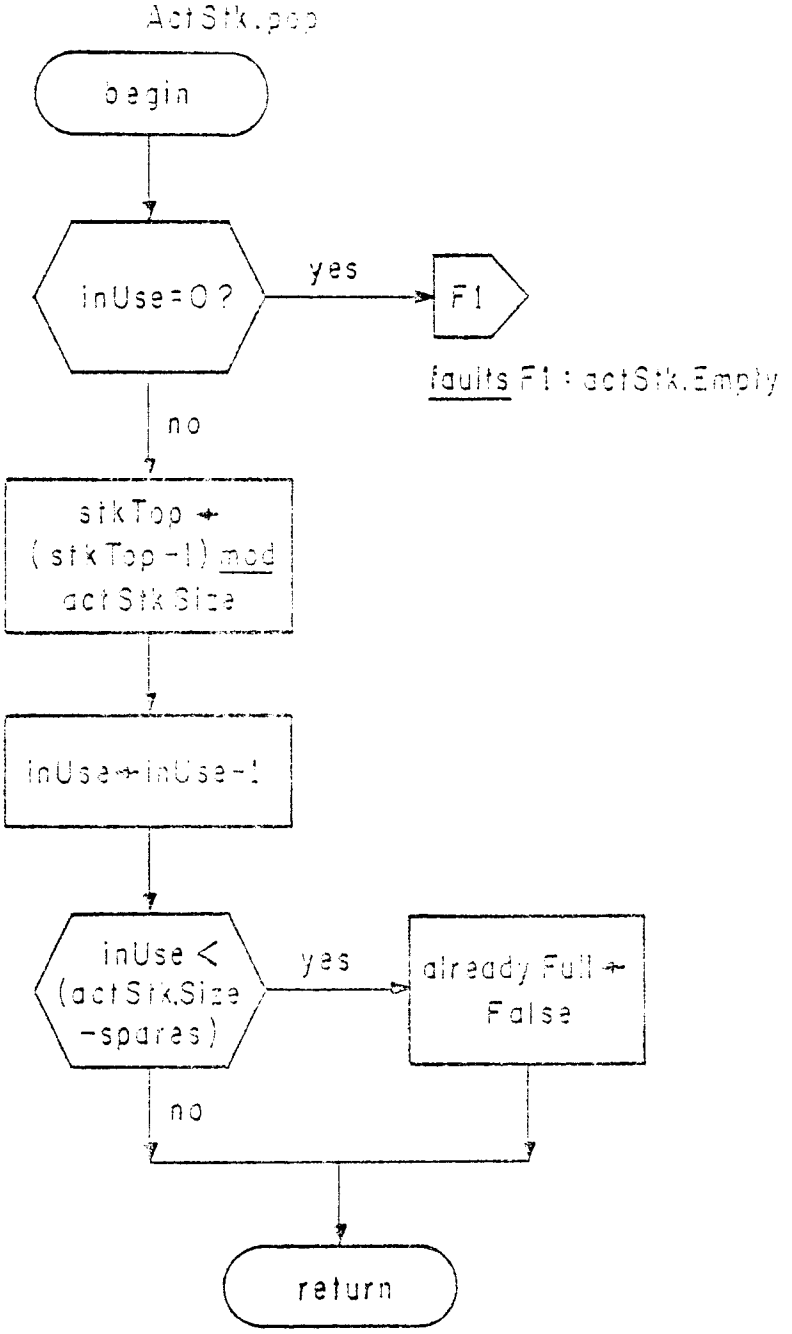

figure 3-27: Pop Activation Stack

Chapter Four

Exception Processor Implementation

4.1 Introduction

In this chapter we develop an implementation of the distinguished subsystem which provides exception processing services for the subsystems executing in a process. These services correspond to the exception facility described in Chapter 2. In particular, the exception processor will implement an invoker controlled handler choice policy and will support the several handler termination modes discussed in Section 2.5.

The exception processor is itself a subsystem, albeit a very special one. Building upon the subsystem isolation and protection facilities described in Chapter 3, the exception processor extends the processor level fault and signal mechanisms. The processor level isolation of subsystems, along with the subsystem transfer operations for signalling exceptions and terminating the exception processor, make the task of implementing the exception processor fairly simple.

The management of the response to exceptions caused by the exception processor itself is of particular interest and importance. The implementation of the exception processor will depend heavily on the exception processor's ability to select handlers to deal with its own exceptions. Exceptions caused by the exception processor are processed, for the most part, by the same algorithms which are used to process normal exceptions. The exception processor interprets its own handler

specifications to determine the response to its own exceptions. The ability to manage exception processor exceptions without inventing new exception handling mechanisms or algorithms supports our feeling that the exception facility being implemented is powerful and flexible enough to handle user and system exceptions without compromising the integrity of individual subsystems.

In order to implement the exception processor, we must make some assumptions. We assume that parts of the exception processor are fixed in real memory so that no virtual memory faults will be caused by referencing the program or the handler specifications of the exception processor. On the other hand, we also assume that the exception processor can cause virtual memory faults on references to its activation frame or to the handler specifications stored in the invoker's root segment. In addition, activation stack full and empty faults may be caused by exception processor actions. These assumptions allow the exception processor to cause faults which are correctable, but rule out faults which might cause an endless sequence of faults. We assume, for convenience, that handlers are available for virtual memory and activation stack faults and that the exception processor can call them directly whenever they are needed.

We must also assume that the exception processor and the necessary handlers are installed in the process when the process is created and that the per-process incarnation segment of the exception processor is initialized and fixed in real memory. This assumption is necessary because the processor model requires an exception processor for fault and signal operations and the exception processor requires virtual memory

and activation stack fault handlers in order to function. Also, the exception processor is assumed to have an external reference to the process base stored in its incarnation segment so that it can reference the fault and signal parameters stored there by the processor.

The implementation of the exception processor can be divided into four parts. Figure 4-1 is a flowchart of the organization of the exception processor. Exception processor execution begins with the entry sequence. There are three entry points to the exception processor. They correspond to 1) basic processor faults, 2) subsystem signals, and 3) exception processor direct calls. The task of the entry sequence is to collect the information describing the exception into the exception processor's activation frame. Once the exception parameters have been safely stored in the activation frame, the handler choice rule is applied to find a handler. Next, the handler is activated. The fourth stage of exception processor execution is the processing of the handler results. When the handler returns to the exception processor, the requested handler termination action is undertaken.

The rest of this chapter is divided into three sections which describe the details of the exception processor implementation. Section 4.2 discusses the entry sequences. Handler selection and activation are discussed in Section 4.3. Finally, Section 4.4 presents the implementation of the several handler termination modes.

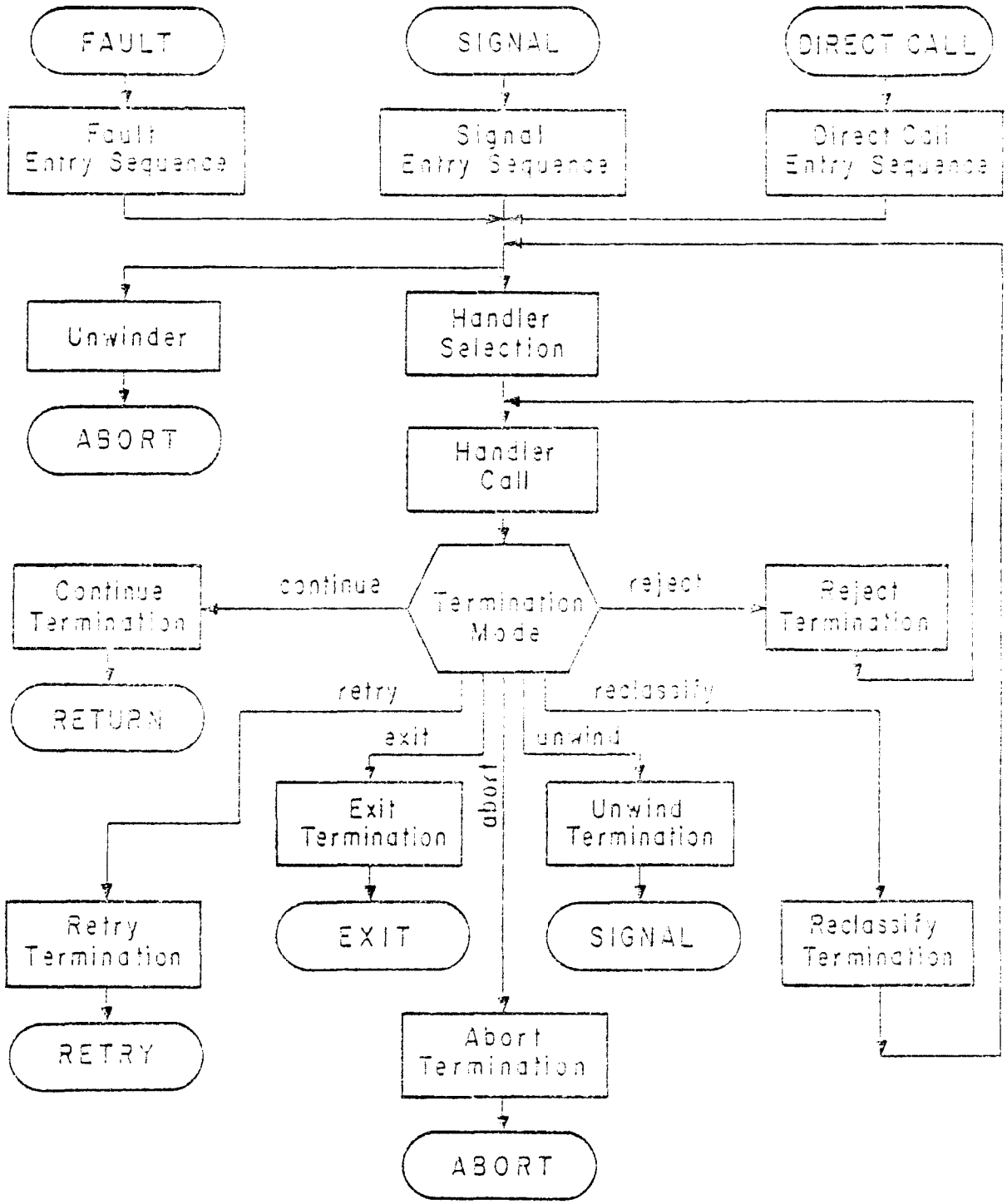

Figure 4-1: Exception Processor Organization

4.2 Exception Processor Entry Sequences

Before the exception processor can proceed to handler selection, the information describing the exception must be stored in a safe place where it will not be destroyed by subsequent processor faults. The fault data is stored by the augmented processor into a buffer in the process base. Should the exception processor cause a processor fault before the current fault data has been moved out of the process base, the original fault information would be overwritten by the new fault data. One main requirement for the entry sequences is that they not cause such premature faults.

The objective of the exception processor entry sequences is to leave the exception information in the activation frame of the exception processor. Once the fault information is arranged in the activation frame, additional exceptions can be processed without losing information about the current fault. Handler selection and activation can proceed using the standard exception processing facilities to manage recovery from exceptions encountered along the way.

There are three exception processor entries to consider. First of all, processor faults cause the exception processor to be activated with an empty activation frame. All the fault information is stored in the process base. Signal operations activate the exception processor with part of the signal data (exception code and message) in the activation frame and the rest (signaller-id and invoker reference) in the process base. Finally, the exception processor direct call supplies all of the exception information, except for the signaller-id, as parameters from the caller.

The exception information which must be gathered together in the exception processor's activation frame includes: 1) a reference to the invoker's address space, 2) the signaller-id, 3) the exception code, and 4) the exception message. Let us assume that the information is to be arranged at the base of the exception processor's activation frame in the order given above. Besides collecting the exception information, the entry sequences perform per episode initialization for the exception processor. The only initialization necessary is to make the set of encountered exceptions empty (see section 4.4.6).

4.2.1 The Fault Entry Sequence

The most difficult entry sequence is the processor fault entry. Because all of the information about the fault is in the process base where it will be overwritten by the next processor fault, the exception processor must move it to somewhere else without causing a fault. Moving it directly to the activation frame is out of the question because accesses to the activation frame can always cause virtual memory faults. Therefore, the exception processor must have a buffer in wired down real memory into which it can store the fault information before risking virtual memory faults. The fault data buffer can be in the exception processor's incarnation segment. The CAP FAULTPROC uses a similar buffer to safeguard its fault data [CAP 76b].

A virtual memory fault may occur in the fault entry sequence when the exception processor allocates activation frame space or while copying the fault data from its buffer to the activation frame. If a virtual memory fault occurs during the fault entry sequence, a new exception

processor activation will be initiated by the processor. Our first attempt to deal with virtual memory faults during the fault entry sequence is to detect, during the fault entry sequence, that a virtual memory fault occurred in the fault entry sequence of the preceding exception processor activation. We shall see that things are more complicated when the finite size of the activation stack buffer is taken into account. The condition which must be detected is characterized by 1) the invoker is the exception processor, 2) the signaller is the processor, 3) the fault is a virtual memory fault, and 4) the instruction pointer of the invoker (old exception processor) is in the fault entry sequence. Condition two is implied by the fact that the fault entry sequence is chosen. The other conditions can be detected by checking the fault data and the activation stack.

If the above condition is detected, the entry sequence of the new exception processor activation can directly call the virtual memory fault handler. The virtual memory fault handler is assumed to take its parameters from the process base since they cannot be passed through virtual memory. We also assume that once the virtual memory fault handler has been activated, it will retrieve the missing activation frame page and return with retry termination without causing any additional exceptions. If the virtual memory fault handler cannot retrieve the missing activation frame page, the process must halt.

Figure 4-2 is a flowchart of our first attempt at an exception processor entry sequence. Before allocating activation frame space and copying the fault data to the activation frame, the current fault data is copied to a wired down buffer in the exception processor's incarna-

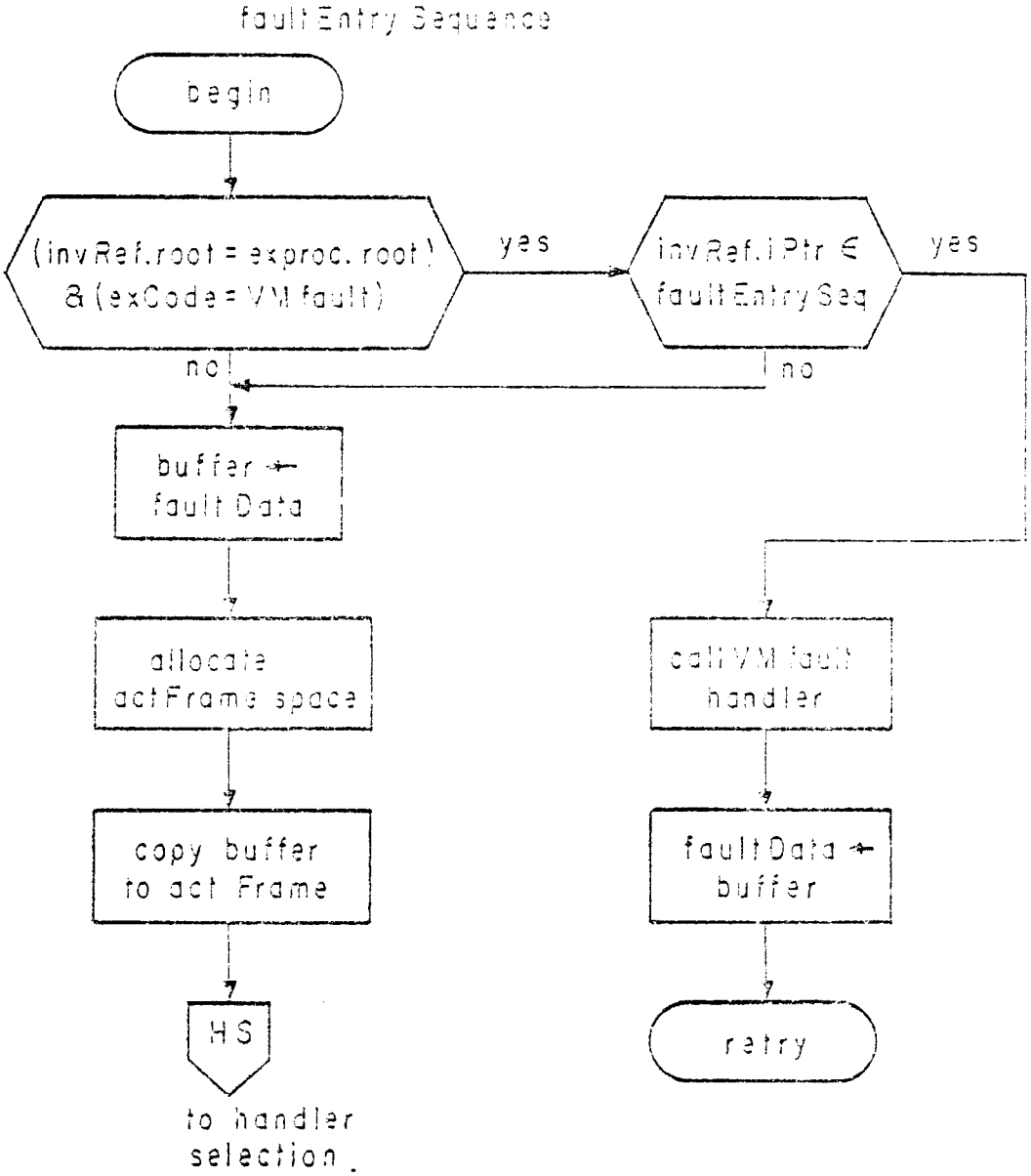

Figure 4-2: Fault Entry Sequence (first attempt)

tion segment. The check for a previous virtual memory fault by the exception processor entry sequence is performed before moving the fault data to the buffer. If a previous virtual memory fault in the exception processor entry sequence is detected, the virtual memory fault handler is activated. When the handler returns, the fault data in the process base is restored from the buffer and the exception processor returns to the earlier exception processor activation using the retry operation. The retry causes the failed allocation or copy operation in the original exception processor activation to be attempted again. Figure 4-3 depicts the sequence of activation stack states for a virtual memory fault during the fault entry sequence.

We must consider carefully the possibility that additional faults may occur while processing the virtual memory fault. In particular, the activation stack may enter its full region on the call to the virtual memory fault handler. This complicates the situation. Data describing more than one fault must be saved in wired down buffers. A stack of buffered fault data in the exception processor's incarnation segment can be used to save fault data while other faults are processed. Figure 4-4 is a modified version of the fault entry sequence. Both virtual memory faults and activation stack full faults are detected during the entry sequence. Fault data is stacked in the fault buffer to safeguard it from being overwritten by subsequent faults. The response to a virtual memory fault is the same as before. If an activation stack full fault is caused by attempting to call the virtual memory fault handler, a third exception processor activation detects the condition and calls a handler to empty the activation stack buffer. When the activation stack handler returns, the fault data in the process base must be restored

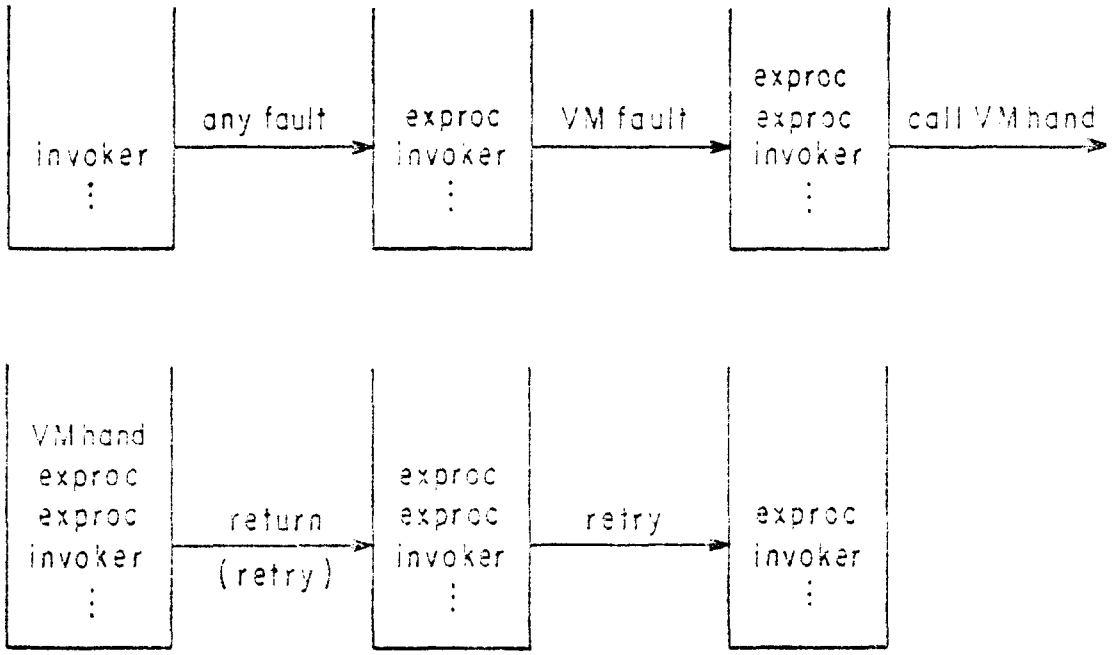

figure 4-3: Fault Entry -- Virtual Memory Fault

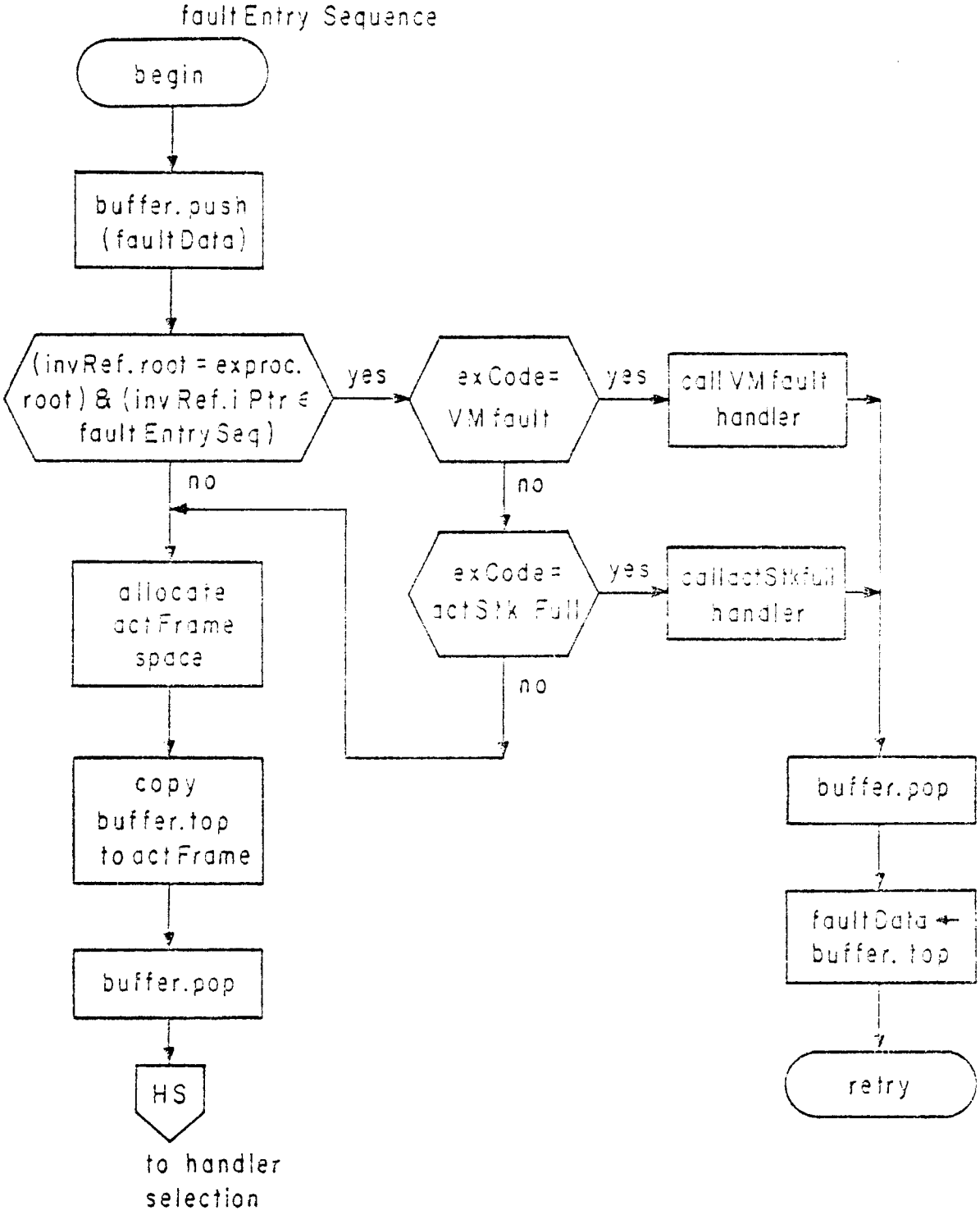

Figure 4-4: Fault Entry (modified)

from the stack of saved fault information because the virtual memory fault handler will expect to find its parameters in the process base.

How many fault buffer must be reserved? Assume that the virtual memory fault handler does not cause any faults, that the activation stack full handler causes only virtual memory faults, and that the activation stack full fault will not occur while the full fault is being processed (see section 3.3.4). Under these assumptions, we can find the worst case scenario of faults during the exception processor entry sequence.

Five faults may occur in the exception processor entry sequence before any exception processor activation completes the entry sequence. Figure 4-5 is the worst case sequence of activation stack states. The circled numbers indicate the number of buffers in use. First, there is the original fault. Next a virtual memory fault moving data to the activation frame interrupts the entry sequence. The third fault is an activation stack full fault caused by attempting to call the virtual memory fault handler. Next, after the activation stack full fault handler has been activated, a virtual memory fault moving data out of the stack buffer may occur. Finally, the entry sequence for the activation stack full handler's virtual memory fault can cause another virtual memory fault. This time the call to the virtual memory fault handler will not fail because the suspended full fault is only signalled once when the stack enters the full region. It appears that a buffer with room for five faults will handle the worst case of faults during the fault entry sequence. Note also that seven spare slots in the activation stack are needed to accommodate suspended subsystem activations if

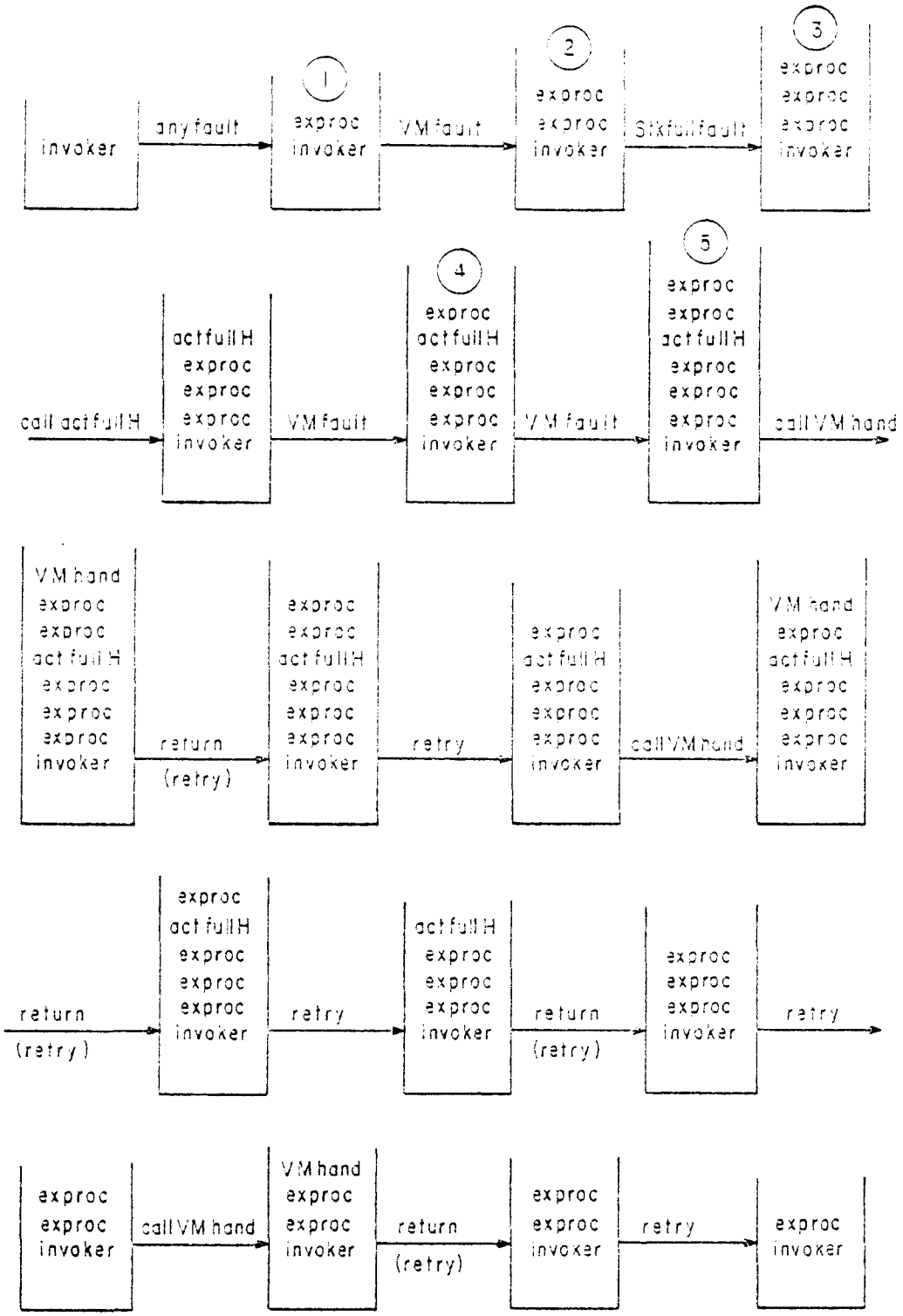

Figure 4-6: Fault Entry -- Normal Case Fault Sequence

the original invoker happens to be the exception processor (see section 3.3.4).

4.2.2 Signal and Direct Call Entry Sequences

Having discovered the fault entry sequence, we can dispose of the signal and direct call entry sequences easily. The signal operation is similar to a basic fault. Instead of placing all the fault data into the process base, only the invoker reference and the signaller-id are stored in the permanently resident process base. The exception code and message are supplied by the signaller in the activation frame. When the exception processor is entered through its signal gate, the exception code and message are already in the activation frame.

The invoker reference and the signaller-id must be copied from the process base to the exception processor's activation frame. The exception code and message are shifted upward in the exception processor's activation frame to make room for the other information. The invoker reference and signaller-id must be staged through a wired down buffer because virtual memory faults may occur moving them to the activation frame. Alternately, the signal operation could have stored the signaller-id and invoker reference in a different place in the process base.

The algorithm for safely moving the invoker reference and signaller-id to the activation frame is similar to the algorithm developed for the fault entry sequence. The difference is that no signal can occur before the signal entry sequence is complete. The only

fault that the signal entry sequence can cause is a virtual memory fault. The virtual memory fault caused by the signal entry sequence can be processed by the exception processor's normal handler selection and activation algorithms. If we know that the exception processor's handler for the virtual memory fault does not signal an exception, and that the fault entry sequence does not lead to signals, the signal entry sequence does not need to maintain a stack of buffers. A single buffer for the invoker reference and the signaller-id will suffice. Figure 4-6 is a flowchart of the signal entry sequence.

The exception processor direct call permits the signaller to initiate exception processing without giving up control. Also, the signaller can select the environment from which the exception processor will select a handler. The parameters of the direct call are the exception code and message plus an invoker reference. The basic processor allows any subsystem activation to generate an address space reference to its own address space. If the signaller passes a reference to its own address space, it will play the roles of both the signaller and the invoker. If the signaller has somehow obtained a reference to some other existing address space, the signaller can cause an exception to be processed as if the passed address space was the invoker. By passing a non-local address space reference, the signaller can implement some non-standard exception handling protocols. For example, a handler can signal exceptions to the invoker without relinquishing control by using a direct call and passing the invoker reference which it received as a parameter. Direct call exception processing can be used to implement notification to the caller that help is needed (c.f. Goodenough) and can also be used to implement an inherited handler policy.

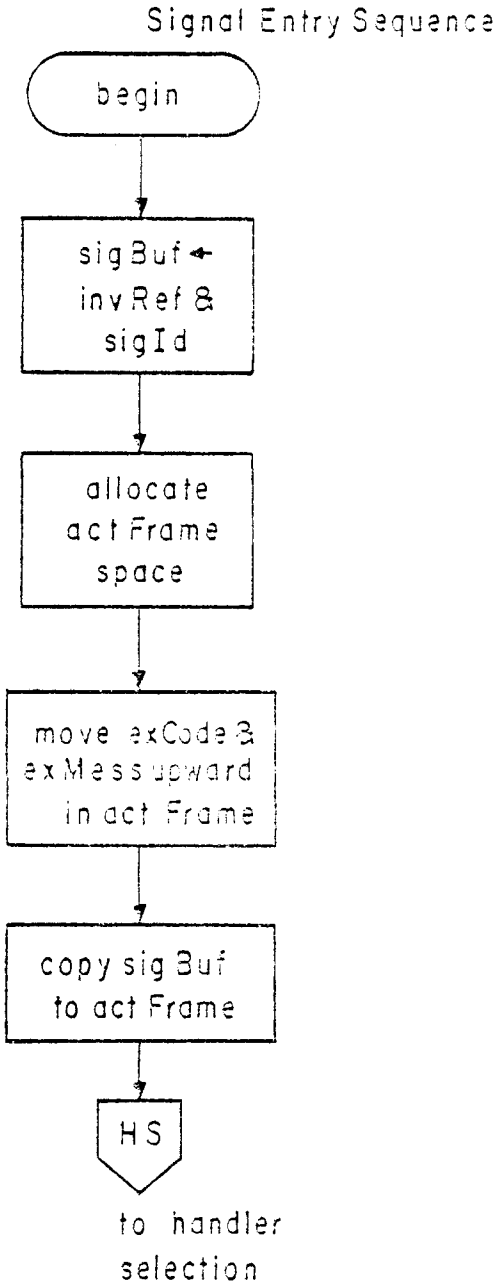

figure 4-6: Signal Entry Sequence

The direct call entry sequence does not need to safeguard any information from the process base fault area. The only information missing from the exception processor's activation frame is the signaller-id. The signaller-id is the same as the identity of the caller of the exception processor and can always be found at the top of the activation stack. The exception processor can examine the activation stack in the process base to deduce the identity of the caller (signaller). There is no danger, in this case, of losing information because of faults during the entry sequence. The algorithm of the direct call entry sequence is given in figure 4-7 .

When the exception processor entry sequence is complete, the exception processor's activation frame contains 1) a non-local reference to the invoker's address space, 2) the signaller-id, 3) the exception code, and 4) the exception message. The information necessary to proceed with handler selection has been assembled in the exception processor's activation frame and the type of the entry to the exception processor is no longer of any importance. Handler selection can proceed without taking into account whether the exception processor activation is due to a processor fault, a subsystem signal, or a direct call.

4.3 Handler Selection and Activation

Once the exception processor has succeeded in collecting the exception data into its activation frame, it can proceed to the selection of a handler for the exception. The exception processor selects a handler using the invoker controlled handler choice policy, which was described in Section 2.4.4. Handler selection is accomplished by examining the

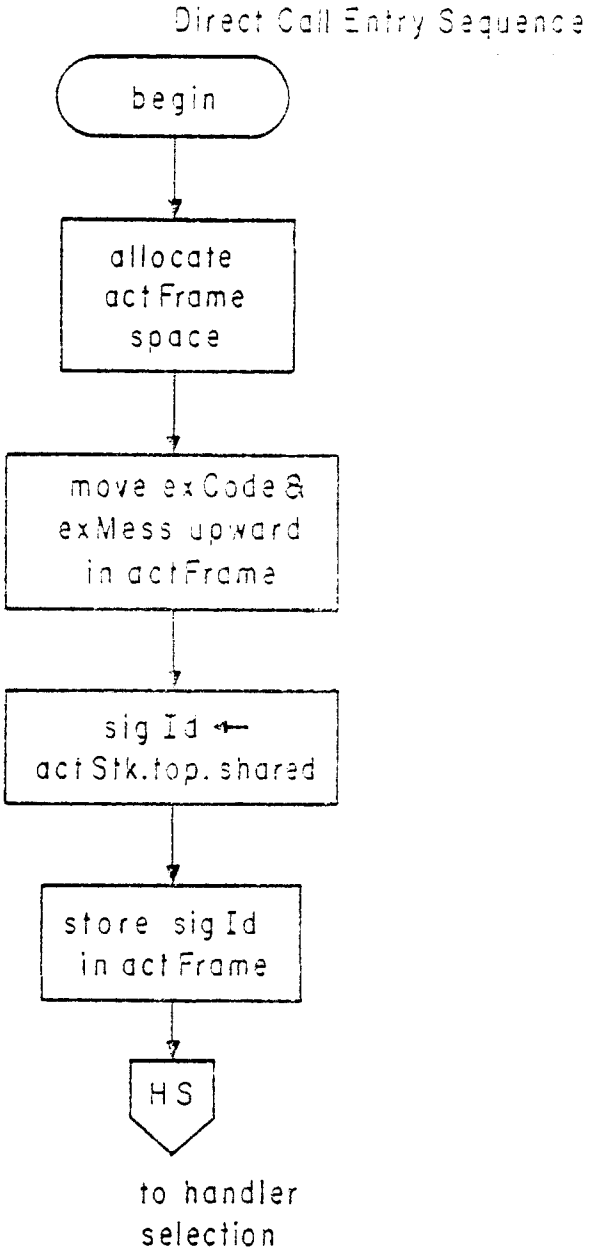

figure 4-7: Direct Call Entry Sequence

invoker's handler specifications using the address space reference which is stored at the beginning of the exception processor's activation frame.

The invoker controlled handler selection rule selects a handler from the imposed, local, and default handler specifications stored in the invoker's root segment. The exception processor handler selection rule is implemented by a procedure, 'findHandler', which searches the invoker's handler specifications for one which corresponds to the current exception. The procedure looks first for an imposed handler. If there is no imposed handler for the exception, 'findHandler' examines the local handler specifications. Each local specification is associated with a range of instruction pointer values in the invoker's root segment. The handler search procedure selects the local specification with the smallest range which contains the current execution point of the invoker. If no local handler can be found, the default specifications are examined.

After a handler specification has been located, the handler must be called. The exception name and message must be passed to the handler. If the handler specification authorizes it, the invoker reference is also passed to the handler. Many exceptions may occur during handler selection and activation. The exception processor's response to its own exceptions is controlled by the exception processor's handler specifications.

4.3.1 Representation of Handler Specifications

The handler selection procedure must examine the data structures representing the handler specifications of the invoker. Because the handler specifications are prepared by subsystems other than the exception processor, care must be taken to prevent malformed data structures from compromising the exception processor. The data structures which represent the handler specifications should be designed with two principles in mind: robustness and safety.

The handler specifications should be robust in the sense that a single mistake in their structure should not ruin the entire set of handler specifications. Also, robustness implies that the well formedness of the data structures can be checked. Safety for the handler specifications means that errors in the data structures should not lead to malfunctions in the exception processor. Particularly, the exception processor should be able to avoid endless looping caused by malformed handler specifications.

Besides locating the highest priority handler for a given exception, the exception processor must be able to locate the next highest priority handler for a given exception when the first handler rejects responsibility for the exception. If 'findHandler' returns the highest priority handler, another procedure, 'nextHandler' is needed to step through the handlers for a given exception. Given the requirements for locating handlers, for robustness, and for safety, we can proceed to a design of the data structures for representing handler specifications. The following design should be considered an example of one way in which the handler specifications might be implemented. The reader will no

doubt think of other, possibly better, implementations.

Simplicity and robustness considerations suggest that the three sets of handler specifications be represented independently. Each set of specifications can be accessed through a hash table keyed on the exception name (signaller-id plus exception code). A chained hash table implementation avoids searching the entire table when no specification with the desired name is present. The elements of the chain from the hash table entry should contain the exception name, a pointer to the next entry with the same hash code, and a pointer to the first handler specification with the indicated exception name. The hash chain elements should also contain some redundant information to protect against looping in the exception processor caused by loops in the hash chains. A check field containing the length of the rest of the chain will do the job. Figure 4-8 illustrates the hash table data structures used for all three types of handler specifications.

The format of the handler specification will depend upon the type of the specification. Imposed and default specifications can be linked in a simple list from their hash table element. The order of the specifications in the list should correspond to their priority. For imposed handlers, new specifications would be inserted at the head of the list during subsystem definition and creation. Default specifications should be added at the end of their list. Besides chaining information, the handler specifications should contain 1) the handler gate, 2) flags to authorize handler access to the invoker's address space and to authorize the various handler termination modes, and 3) a check field similar to the check field in the hash node. Figure 4-9 depicts the

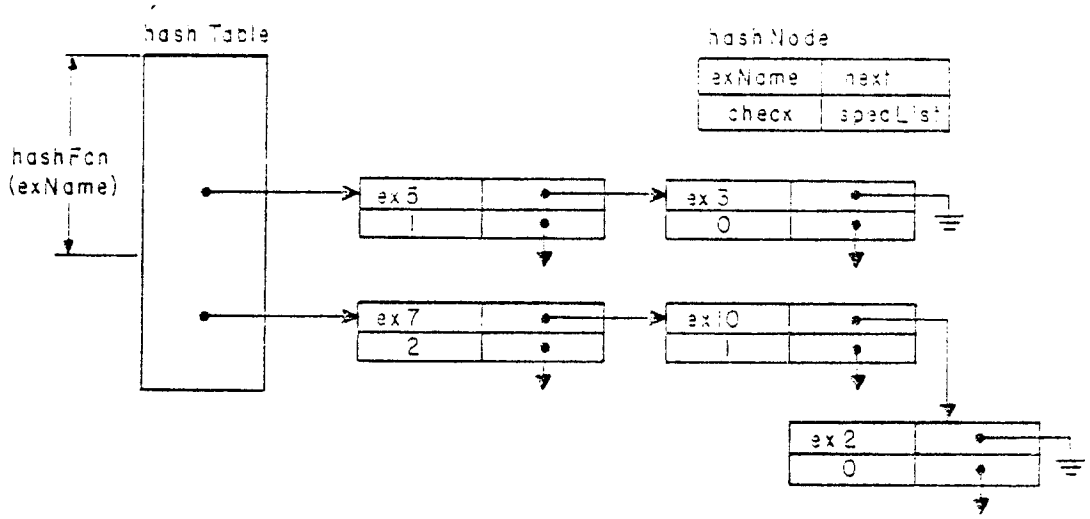

Figure 4-8: Handler Hash Tables

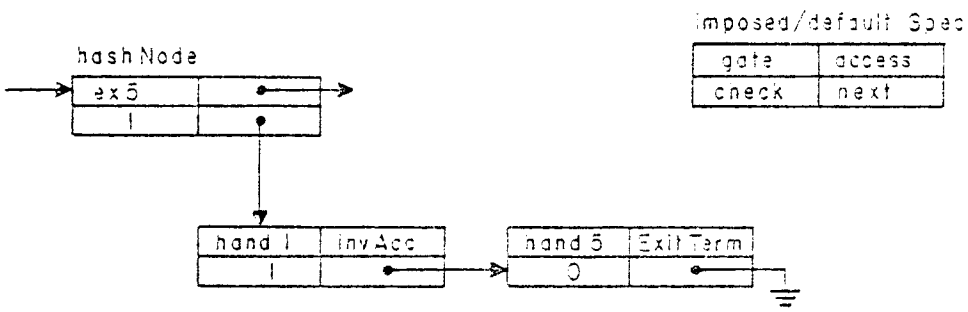

figure 4-9: Imposed and Default Handler Lists

handler lists for imposed or default handler specifications. For imposed and default handler specifications, the exception processor should encounter only monotonically decreasing check fields.

Local handler specifications are more complicated than imposed or default specifications. Local specifications may apply to only part of the invoker's code space, and they can be nested. An appropriate data structure for representing nested structures is a tree. The local handler specifications associated with a single exception can be represented in a triply linked tree [Knuth 68, p.352].

Each node of the tree contains, in addition to the handler gate and authorizations, an upper and a lower instruction pointer value delimiting the range of activation points covered by the local specification. The ranges of the descendents of a node should all be contained in the range of the parent. The ranges of siblings are not allowed to overlap. Three links are used to traverse the local handler tree. Each node contains a link to its parent, to its elder sibling, and to its latest offspring. The sibling and offspring links are used to locate the specification with the smallest range which contains the invoker's locus of control. The parent link is used by 'nextHandler' to find the next enclosing local handler specification.

Figure 4-10 presents a data structure which might be used to implement local handler specifications. As in the case of the other handler specifications, a check field is used to detect loops in the pointer structure. If the local handler tree is numbered in endorder, every parent will have a higher number than any of its offspring, and younger siblings will have higher numbers then older siblings. When searching

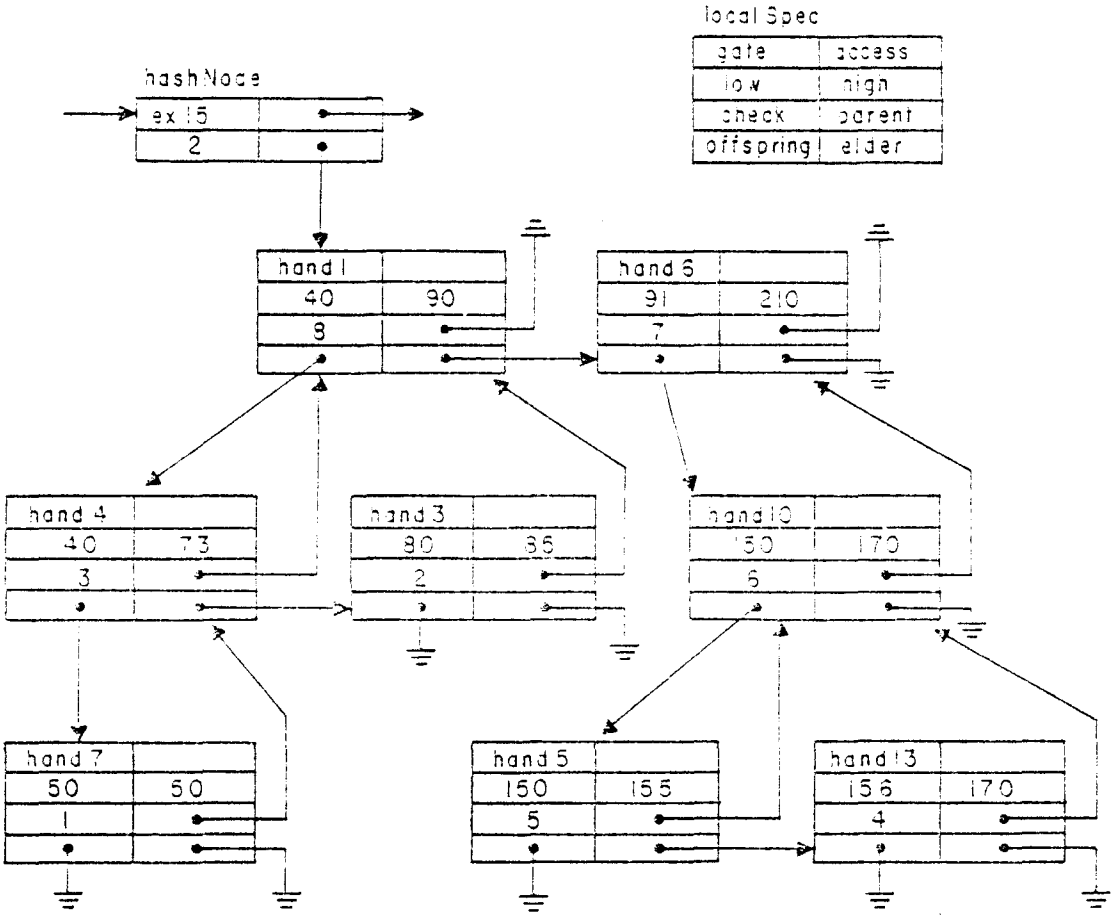

figure 4-10: Local Handler Tree

for a local handler, the exception processor should encounter ever decreasing values of the check field. Because the local handler tree is completed during subsystem creation, the numbering of the nodes must be completed before the subsystem is executed. It is not the exception processor's responsibility to build or number the handler specification trees or lists. Of course, subroutines for building handler specifications can be made available to compiler and supervisor implementors.

4.3.2 Handler Selection

Handler selection begins with a check to see if unwinding is in progress. If the exception processor has been activated to propagate an unwind request, the unwinder is invoked instead of proceeding to handler selection. We defer the discussion of unwinding to section 4.4.5. If unwinding is not under way, the exception processor calls 'findHandler' to select a handler for the current exception. Given the data structures for representing the handler specifications in the address space of the invoker, the 'findHandler' procedure is easily implemented.

The 'findHandler' procedure searches the imposed, local, and default specifications, in that order. If 'findHandler' locates an imposed or default hash table entry with the correct exception name, it returns a pointer to the first handler specification on the list. If a local hash table entry is located, 'findHandler' returns a pointer to the specification with the smallest range which contains the invoker's instruction pointer. If no containing range is found, then the search proceeds to the default table. When 'findHandler' is unable to locate a handler specification, it returns a nil pointer.

Figure 4-11 is a flowchart of the handler selection algorithm. When 'findHandler' fails to produce a handler specification, the exception processor should change the exception to reflect the failure of the invoker to provide a handler specification. Changing the exception is discussed in section 4.4.6. The new exception code can be 'noHandler' and the new signaller should be the exception processor. The old exception name should be prefixed to the old exception message to form the new exception message. After the exception name is changed, handler selection is re-initiated.

The exception processor may encounter a variety of exceptions in the process of selecting a handler, including 1) virtual memory faults in the exception processor activation frame or in the invoker's address space, 2) illegal addresses in the invoker's address space caused by bad pointers in the handler specifications, 3) the use of an obsolete invoker reference passed on a direct call to the exception processor, 4) out of stack non-local references because the invoker is no longer in the active portion of the activation stack, and 5) malformed handler specifications detected by 'findHandler'. The exception processor's handler specifications can designate handlers for each of these anticipated exceptions. The range of these handler specifications should include 'findHandler' and 'nextHandler'. Figure 4-12 illustrates the flows of control and the exception processor actions in response to exceptions occurring during handler selection. Dashed lines represent transfers of control via the exception processor. In this case, the exception processor controlling the transfers is operating on behalf of another exception processor activation.

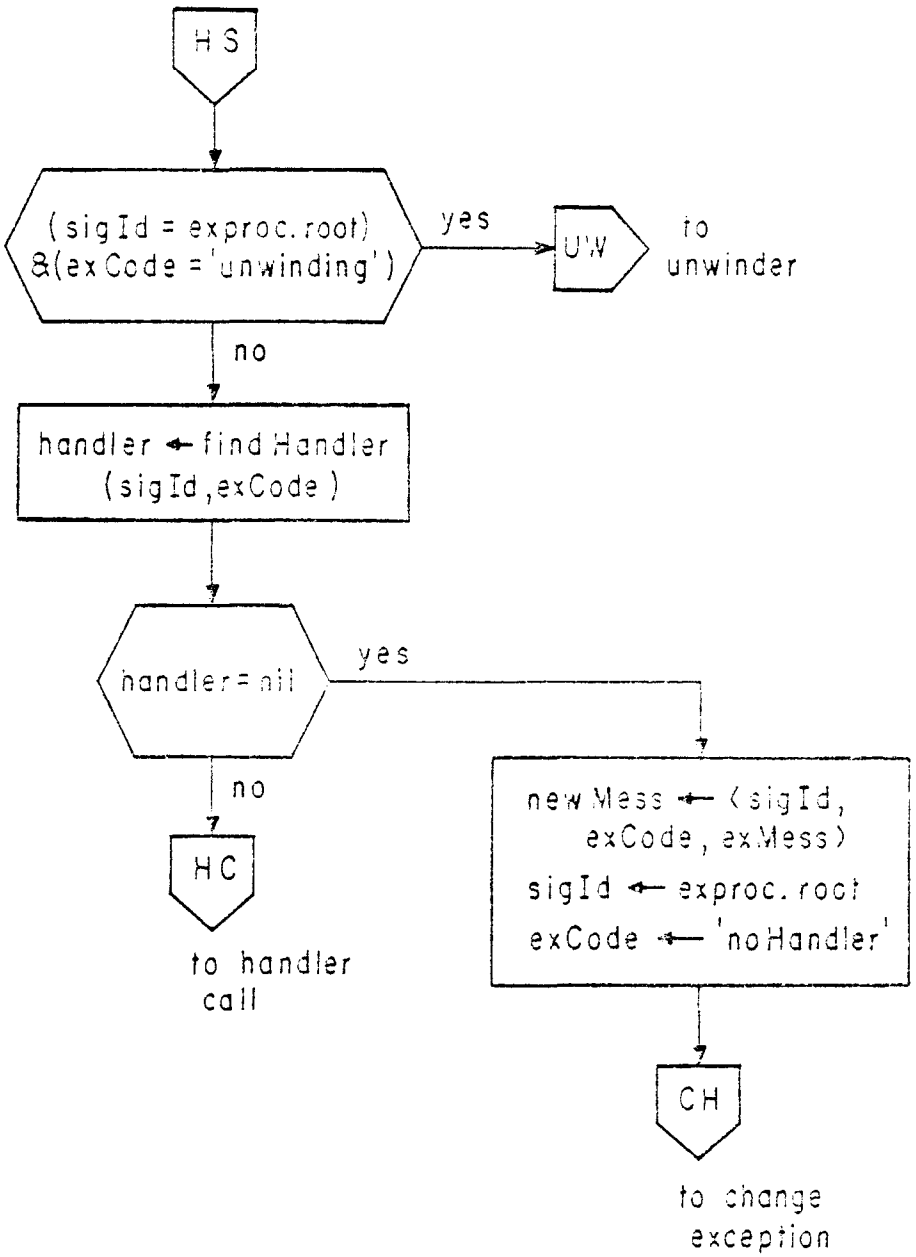

figure 4-11: Handler Selection

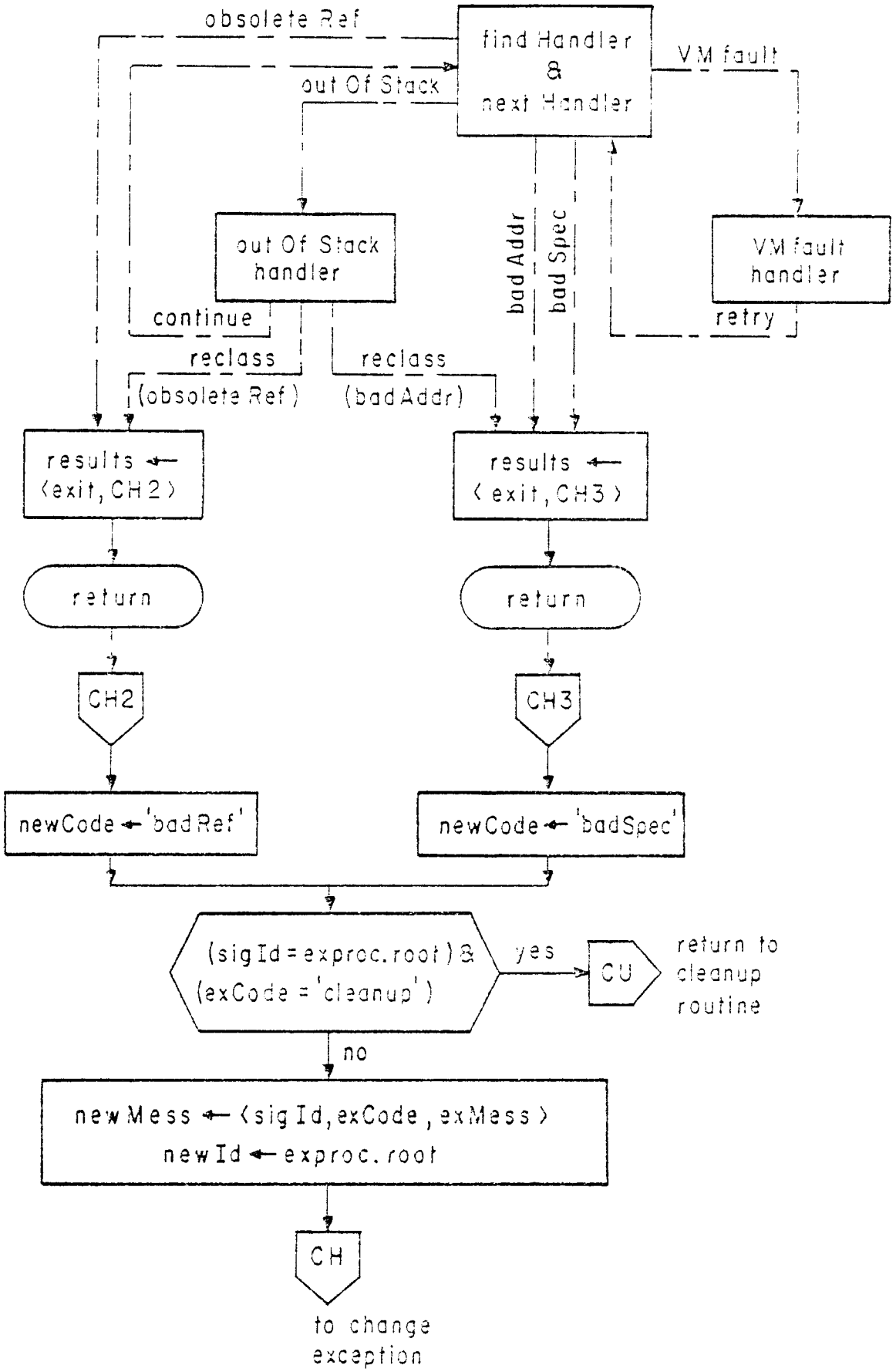

Figure 4-12: Handler Selection Exceptions

Four different handlers are needed to manage the response to the five exceptions which may occur during handler selection. The virtual memory and the 'outOfStack' faults can be directed to system handlers. Exception processor local handlers for obsolete address space references and malformed handler specifications respond to unrecoverable exceptions during handler selection. If handler selection exceptions occur during abort or unwinding termination, the exception is ignored. The cleanup routine, which attempts to locate 'cleanup' handlers for the subsystem activations which are being forced to terminate, is discussed in section 4.4.4.

Virtual memory faults can occur in the exception processor activation frame or on a reference to the handler specifications in the invoker's address space. The response to virtual memory faults can be controlled by an imposed handler specification and a local specification. The imposed handler is the system virtual memory fault handler. After retrieving the missing page, it terminates with a retry causing the original exception processor activation to repeat the failed access to virtual memory.

The 'outOfStack' fault occurs when the invoker is not in the active region of the activation stack. The handler for 'outOfStack' faults will attempt to simulate the non-local reference. This handler is assumed to be part of the subsystem which is responsible for maintaining the virtual memory part of the activation stack. The handlers for 'actStkFull' and 'actStkEmpty' are also in this subsystem. The 'outOfStack' handler can simulate the failed reference using the address space description which has been saved in the virtual memory stack. If

the 'outOfStack' handler succeeds in making the reference, it can continue the original exception processor activation with the requested information. If the non-local reference cannot be simulated, the 'outOfStack' handler reclassifies the exception. The reclassified exception may be either 'badAddr' or 'obsoleteRef'. The exception will be reclassified to 'obsoleteRef' if the address space cannot be located. The exception is reclassified to 'badAddr' if the requested address is not with the ranges allowed for the target address space.

If the original fault is 'obsoleteRef' or the exception is reclassified to 'obsoleteRef', the exception processor's handler specifications select a local handler. This handler exits to a code fragment in the original exception processor activation. The code fragment checks to see if handler selection is in progress for the cleanup routine. If so, a transfer back to the cleanup routine ends the processing of the exception. If normal handler selection was in progre, the exception is changed to 'badRef' by transfering to the change exception sequence (see section 4.4.6). Note that 'obsoleteRef' exceptions from either the processor or the 'outOfStack' handler are directed to the same handler. This requires two handler specifications in the exception processor.

The fourth handler involved in handler selection exceptions is the 'badSpec' handler. This handler is selected by three handler specifications in response to 1) a 'badAddr' fault from the processor, 2) a 'badAddr' exception from the 'outOfStack' handler, and 3) a 'badSpec' exception reported by 'findHandler' or 'nextHandler'. Note that 'findHandler' and 'nextHandler' report exceptions by calling the exception processor thru its direct call gate. The action of the 'badSpec'

handler is similar to the 'obsoleteRef' handler. The 'badSpec' handler uses exit termination to transfer to a code fragment which changes the exception to 'badSpec'. If cleanup processing was in progress, control is returned to the cleanup routine instead of changing the exception.

Seven handler specifications are needed to control the exception processor's response to exceptions during handler selection. The first handler specification directs virtual memory faults to the system virtual memory fault handler. The 'outOfStack' fault from the processor is directed to the system 'outOfStack' handler. Two more handler specifications direct 'obsoleteRef' exceptions from the processor or the 'outOfStack' handler to the exception processor's 'obsoleteRef' handler. A 'badAddr' exception from the processor or from the 'outOfStack' handler is directed to the 'bad'Spec' handler. Finally, 'findHandler' and 'nextHandler' report a 'badSpec' exception by calling the exception processor when loops are detected in the handler specifications. The result of handler execution during handler selection is either to return to handler selection following recovery by the virtual memory fault handler or the 'outOfStack' handler, to change the current exception to 'obsoleteRef' or 'badSpec', or to return to the cleanup routine.

This section has illustrated the use of the exception processing facility to recover from exceptions during handler selection. The ability of the exception processor to make use of its own exception processing facility has been demonstrated. The interactions between the exception processor and its local handlers show how a subsystem can supply the response to its own exceptions. The use of non-local handler for virtual memory and activation stack faults demonstrates how a subsystem

might interact with system supplied exception handlers. The following sections will provide further examples in which the exception processor exploits its own exception processing facilities to control the response to the exceptions it causes.

4.3.3 The Handler Call

Once a handler specification has been located, the exception processor can call the handler through the gate designated in the handler specification. The exception processor must prepare the handler's actual parameter list at the top of its activation frame. The parameters to the handler include: 1) an indication of which handler termination modes are allowed, 2) a reference to the invoker's address space (if authorized), 3) the exception name (signaller-id and exception code), and 4) the exception message.

The handler specification includes handler termination authorizations which are passed on to the handler. The handler should not attempt to use terminations which are not authorized. The invoker reference is a non-local address space reference. The handler can access the contents of the invoker's address space using the invoker reference. If the handler specification does not authorize invoker access, a nil invoker reference will be passed to the handler. The exception name and message are passed to the handler so that it can check to see which exception caused it to be activated.

Once the parameters have been prepared at the top of the exception processor's activation frame, the handler can be activated by a subsys-

tem call. If the exception is a processor fault, the exception processor should restore the fault data in the process base before calling the handler. The only reason for restoring the fault data is that some handlers may not execute in virtual memory. These handlers (e.g. virtual memory fault handler) will take their parameters from the process base. Figure 4-13 illustrates the handler call and the processing of the exceptions which can be caused by the handler call operation.

The invocation of the handler may succeed, or an exception may be encountered either on the call or signalled from the handler. If the handler invocation succeeds and the handler returns, the exception processor can continue to the processing of the termination actions. Otherwise, the handler specifications of the exception processor will be used to select a handler to deal with the handler call exception. Possible exceptions caused by the handler call include: 1) the handler is not known in the process, 2) the activation stack becomes full, or 3) the handler signals or is aborted.

If the handler call fails because the handler is not known to the process or the activation stack is full, the exception processor's handlers for these exceptions can attempt to correct the situation and then return to the suspended exception processor activation to retry the operation. Unknown handlers can be incarnated or made know and a full stack buffer can be emptied by privileged system handlers. The exception processor's handler specifications for the 'unknownSSys' and the 'actStkFull' exceptions should designate a local handler. This local handler calls a system handler for the particular exception in hand and then process the results from the system handler. The system handlers

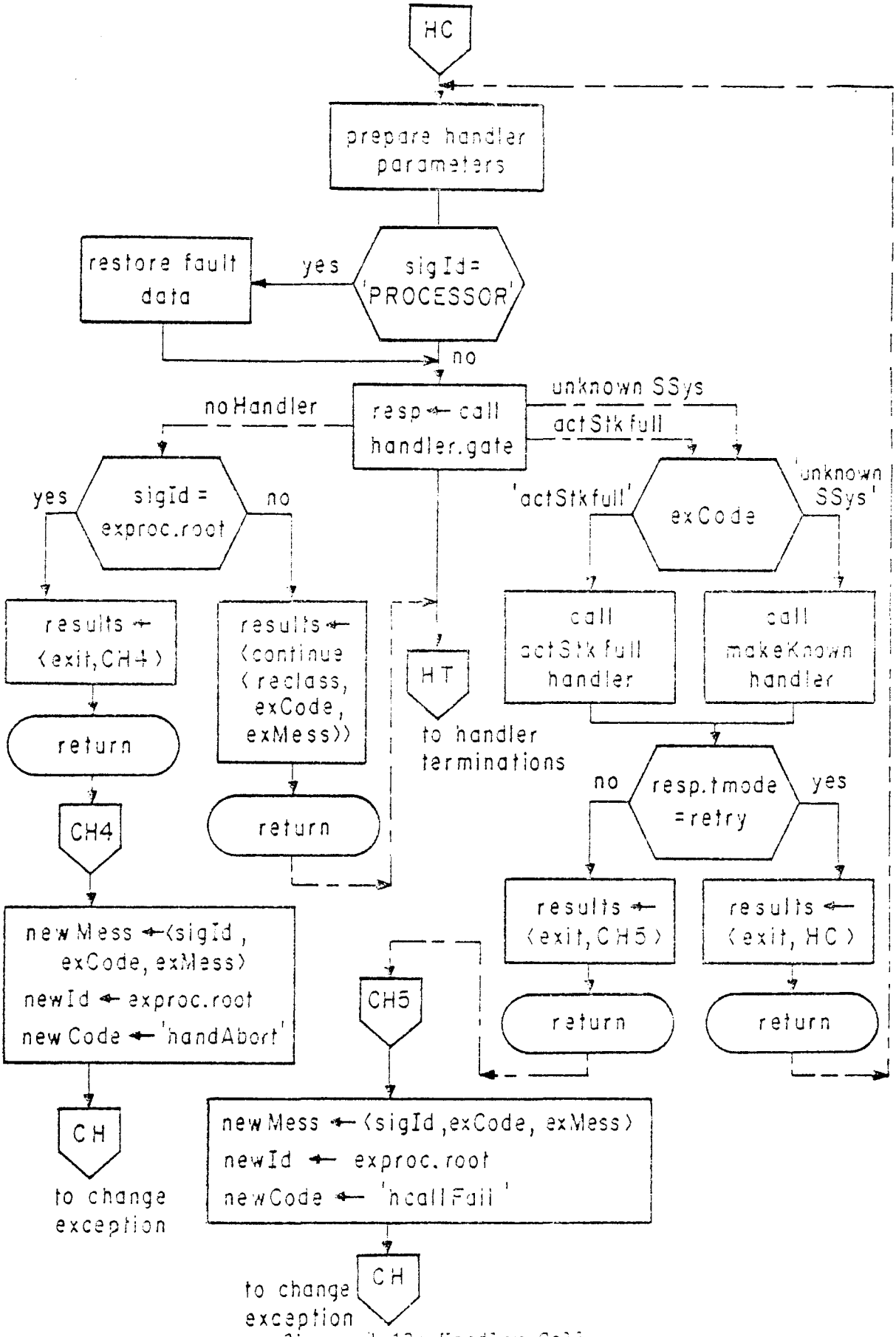

figure 4-13: Handler Call

for 'unknownSSys' and 'actStkFull" should return with retry or reject termination depending on whether they succeeded in correcting the the problem. If the system handler succeeds, the exception processor's handler can exit to the beginning of the handler call sequence. The handler call sequence must be started over because the original exception may be a virtual memory fault. The virtual memory fault handler expects to find the fault data in the process base.

If, instead of correcting the 'unknownSSys' or 'actStkFull' fault, the handler rejects the exception, the exception processor is unable to activate the handler for the original fault. The exception processor's response to this should be to change the original exception. The exception is changed by exiting to a code fragment in the original exception processor activation. The code fragment transfers to the reclassify exception sequence (see section 4.4.6). The new exception will be 'hCallFail' from the exception processor.

If the handler call succeeds, but the handler terminates by signalling an exception, or if the handler is aborted for any reason, a new exception processor will be activated to select a handler. Normally the exception processor will not have handler specifications for exceptions from the handlers it calls since these handlers are selected by the invoker. However, if the exception processor has a local handler for the 'noHandler' exception, the new exception processor activation will call that handler because the failure of 'findHandler' to locate a handler specification for the original exception causes the exception to be reclassified to 'noHandler'. If the signalled exception is from the handler, it can be converted, by the exception processor's 'noHandler'

handler, to a reclassification of the original exception. To reclassify the signal, the 'noHandler' handler continues the original exception processor activation with results calling for reclassification of the original exception. The original exception processor activation cannot distinguish between a return from the original handler and a return from the exception processor's 'noHandler' handler and therefore will proceed with normal termination processing (see section 4.4).

If the original handler was aborted by the exception processor, the 'noHandler' handler must exit to a code fragment which changes the exception to 'handAbort'. A reclassify continuation will not work when the handler was aborted by the exception processor because the signaller of the 'handAbort' should be the exception processor. Normal reclassify termination chooses the handler as the new signaller.

In summary, handler activation involves preparing the handler parameters, calling the handler, and processing exceptions caused by the call or signalled by the handler. Exceptions on the call are processed by calling the appropriate system handler for the exception. If the system handler succeeds in correcting the cause of the call failure, the handler call is re-executed after restoring the fault data in the process base. If the handler call succeeds but the handler signals an exception or is aborted, the exception processor's handler for a 'noHandler' exception will catch the signal or abort. The 'noHandler' handler will change the current exception either by continuing the original exception processor with a reclassify request or by exiting to a code fragment which changes the exception to 'handAbort'. If the handler invocation succeeds and the handler returns normally to the ex-

ception processor, the exception processor can proceed to the handler termination actions.

4.4 Handler Terminations

The exception processor supports a variety of handler termination actions. The various terminations permit the handler to exercise control over the continuation of the computation. The handler termination modes were discussed in section 2.5. In this section, the implementation of the exception processor termination actions is described.

There are seven handler termination modes. The first three, continue, retry, and exit, return control to the invoker. Abort and unwind terminate the invoker and propagate exception processing to the environment of the invoker's invoker. The last two handler terminations, reclassify and reject, continue the exception episode by selecting and activating additional handlers on behalf of the original invoker.

As suggested earlier in this chapter, information embedded in the handler specifications can be used to control which terminations are allowed for a given handler activation. Any attempt to use an unauthorized termination mode will be reported to the invoker as a new exception.

Handlers select a termination mode by returning results to the exception processor. The results are passed through the activation segment from the handler's frame to the exception processor's frame. The handler should prepare its termination request in its activation frame immediately following the exception parameters. The first word of the

result selects the termination mode. Passing the termination request through the activation segment presents no problems except for handlers which do not execute in virtual memory. These handlers must pass their termination requests through the accumulators of the processor. A flag in one of the accumulators can be used to inform the exception processor where to expect the handler termination request.

Figure 4-14 is a flowchart of the beginning of the handler termination processing. If the handler passes the results through the accumulators, the exception processor copies the information to its activation frame. Of course, virtual memory faults may impede the copy, but the exception processor's handlers for those faults can take care of the problem. After checking for non-virtual memory handlers, the handler's termination request is compared with the termination authorizations in the handler specification. If the requested termination is not compatible with the handler specification, the exception processor changes the exception to 'badTerm' by transfering to the change exception sequence. Finally, if the requested termination is compatible with the handler specification, the exception processor branches to the requested termination action. The rest of this chapter is concerned with the implementation of the various termination actions.

The exception processor can sustain the usual virtual memory faults while performing a handler termination request. Also, the exception processor may find the suspended stack is empty when it attempts to return to the invoker. An exception processor handler specification for the empty stack exception can select a gate in the subsystem responsible for managing the virtual memory suspended stack. That handler will load

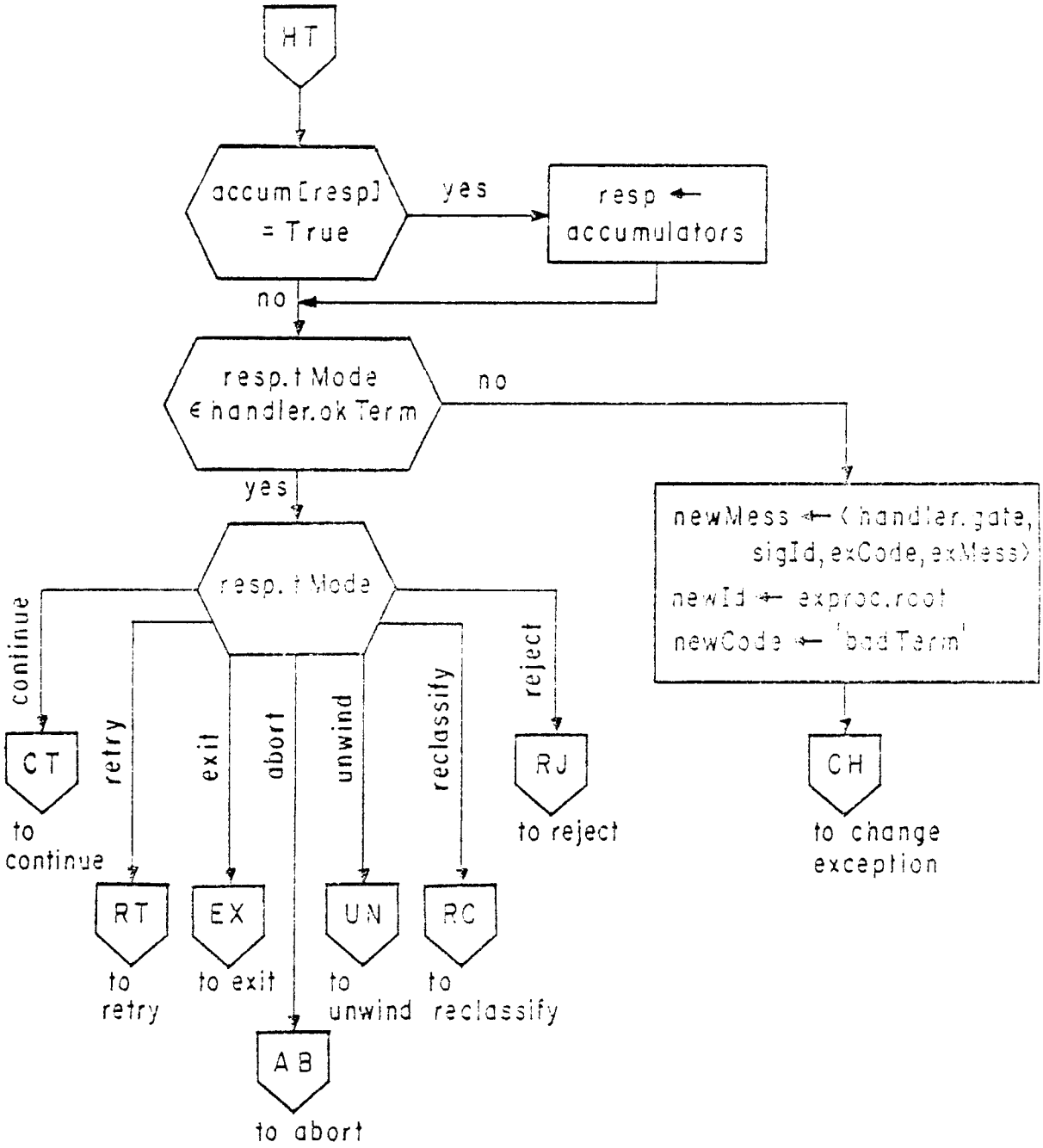

figure 4-14: Handler Terminations

saved entries into the suspended stack and then terminate with a retry request. The return to the invoker should succeed on the second attempt.

4.4.1 Continue Invoker

The handler will request continue termination when it has simulated the failed operation. The substitute results of the failed operation are returned to the exception processor immediately following the termination indicator in the handler results. Continue termination is implemented by copying the substitute results to the base of the exception processor's activation frame, reducing the activation frame, and then executing a subsystem return. The effect of these actions is to return the handler supplied results to the invoker immediately following the parameters to the failed operation. The size of the substitute result vector can be deduced by inquiring about the size of the exception processor activation frame and then subtracting the distance to the end of the handler parameters. Continue termination assumes that the invoker expects its results following the parameters of the failed call. The exception processor also expects the handler results (the termination request) to follow the parameters it passes to the handler. Figure 4-15 illustrates the actions of the continue termination sequence.

4.4.2 Retry Failed Operation

The second handler termination mode is used to retry a failed operation. The result from the handler consists only of the termination

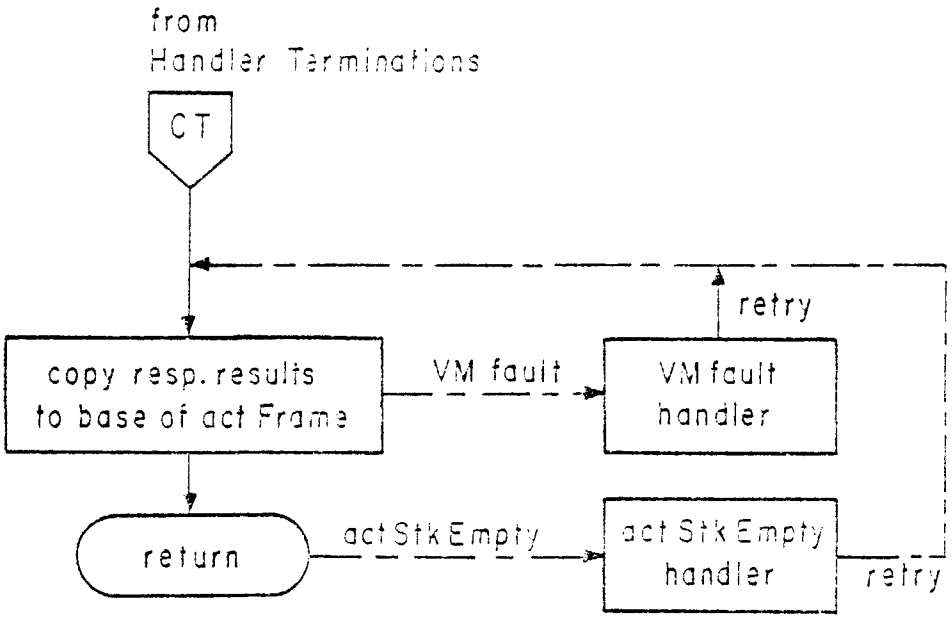

figure 4-15: Continue Termination

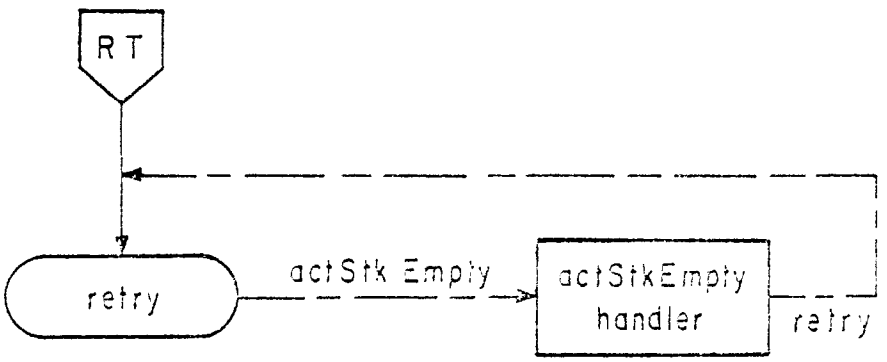

figure 4-16: Retry Termination

indicator. The are no results to return to the invoker. To implement retry termination, the exception processor simply executes the privileged retry instruction. The retry operation was presented in Section 3.3.3. Retry is a privileged processor operation which returns to the invoker without incrementing its instruction pointer.

The possibility exists that the retried operation will fail in the same way as before. The exception processor should probably record some information in its incarnation segment to permit it to detect repeated exceptions on a retry. The test for a repeated retry exception should be made before handler selection. It is difficult to distinguish between legitimate repeated exceptions on a single operation and failed retries. The exception processor would have to somehow determine that the retried operation failed immediately and not on some subsequent invocation. An execution timer associated with the address space is one mechanism which would be helpful for detecting repeated retries [Graham 71, Walker 73].

Figure 4-16 gives the algorithm for retry termination. The test for repeated retries is not included in this implementation. The retry operation restores the invoker's activation frame to its size before the failed operation invocation. If the parameters of the failed call have not been disturbed by the signaller or the handler, the retried operation will be called with the same parameters as before.

4.4.3 Exit To Invoker

Exit termination directs the exception processor to force a transfer of control in the invoker. The handler passes the instruction address for the continuation of the invoker with the results it returns to the exception processor. The exception processor uses the privileged abnormal return operation to return control to the invoker at the indicated address.

The abnormal return operation can fail because the handler selected continuation address is not a legal instruction pointer value. In this case, an exception processor local handler for the 'badAddr' exception reported by the abnormal return operation can change the exception to 'badExit' by exiting to a code fragment as usual. Figure 4-17 shows how exit is implemented.

4.4.4 Abort the Invoker

The handler may request that the invoker be terminated and that exception processing be re-initiated in the environment of the invoker's invoker. The handler will pass with its termination request the exception code and message for the new exception. The signaller of the new exception will be the current invoker. The exception processor will abort the invoker by executing the privileged abort operation (see section 3.3.3). The new exception code and message are passed as the parameters of the abort operation.

Before terminating the invoker, the exception processor should give the invoker a chance to restore itself to a consistent state. The

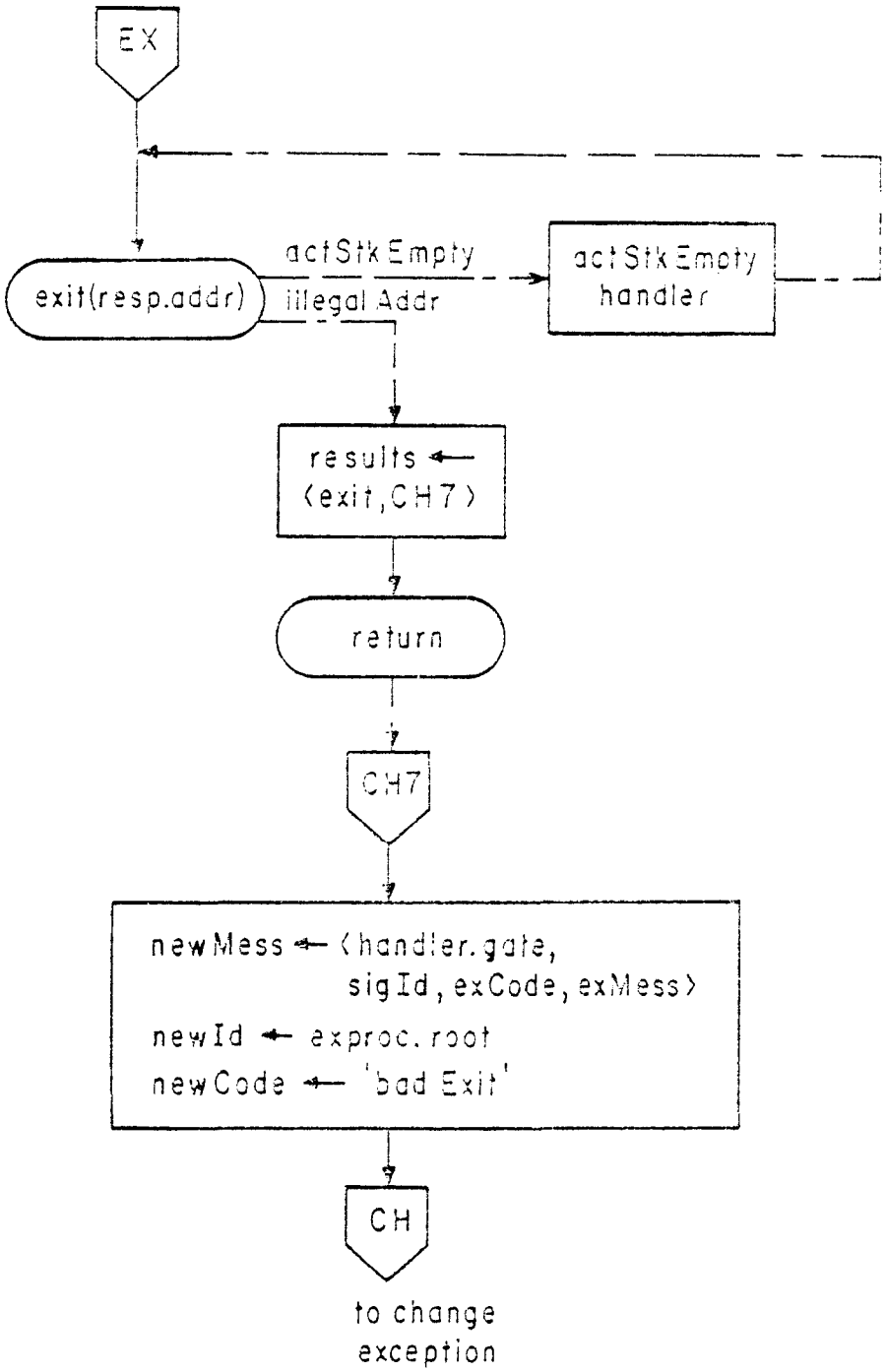

figure 4-17: Exit Termination

invoker can declare its desire to be given control before being terminated by supplying handler specifications for the 'cleanup' exception. Figure 4-18 shows how the exception processor can give the invoker a chance to put its affairs in order before it is forced to terminate. The cleanup routine of the exception processor searches for 'cleanup' handlers using the 'findHandler' and 'nextHandler' procedures. If 'findHandler' locates a 'cleanup' handler, the cleanup routine of the exception processor will call the indicated handler. All of the usual exceptions associated with handler selection and activation can occur in the cleanup routine. However, the handler selection exceptions (except VM faults) are handled a little differently during cleanup. Instead of changing the current exception to 'noHandler' when 'findHandler' fails, control is returned to the cleanup routine because there are bigger fish to fry -- the invoker is being aborted. An exception on the handler call is managed similarly. If the handler cannot be called, or if it signals an exception, the handler is bypassed by the cleanup routine. The exception processing paths and specifications are not represented in the cleanup routine flowchart.

Only reject terminations are allowed from 'cleanup' handlers. All other terminations are converted to reject termination. When a 'cleanup' handler terminates, the cleanup routine calls 'nextHandler' to locate another 'cleanup' handler. This allows a nested set of local handlers to back out from each block of the invoker. If also allows supervisor imposed and default handlers to be activated to close files and release resources. When no more 'cleanup' handlers can be found, the cleanup routine returns to the abort sequence. The cleanup routine is also used during unwinding (see next section).

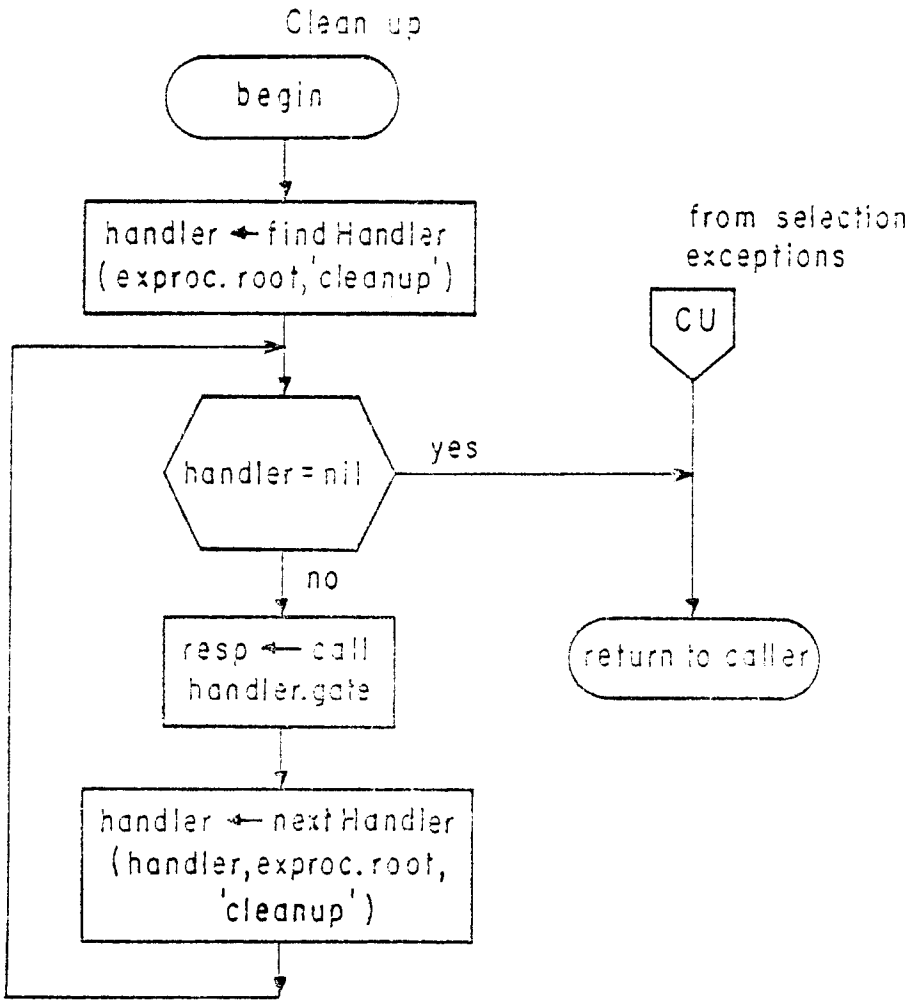

figure 4-18: Cleanup Routine

The possibility exists that the invoker reference used by handler selection during cleanup does not refer to the subsystem about to be aborted. This can occur only on a direct call to the exception processor. The abort sequence skips the cleanup if the invoker reference is not to the subsystem about to be aborted. Note that this can happen only on a direct call in which case the caller has placed itself at the mercy of the invoker which it designated.

Once the invoker has been given a chance to set its affairs in order, the abort sequence prepares to invoke the privileged abort operation. The abort code and message from the handler are copied to the base of the exception processor's activation frame and the activation frame is reduced to contain just the abort code and message. Finally, the exception processor issues the abort. Figure 4-19 shows the algorithm of the abort sequence.

The abort operation will fail if there are not two entries in the activation stack or if a virtual memory fault occurs during the copy of the abort parameters to their new spot in the activation segment (see Section 3.3.3). In either case, a system handler can be selected by the exception processor's handler specifications. The handler, after fetching the missing page or loading entries into the activation stack buffer, can retry the failed abort operation. The implementation of the abort primitive uses the activation frame pointers of the invoker as progress indicators during the copy of the abort parameters. Any attempt to return to the invoker after an abort has been issued will cause trouble because parts of the invoker's activation frame contents will have been overwritten with the abort parameters.

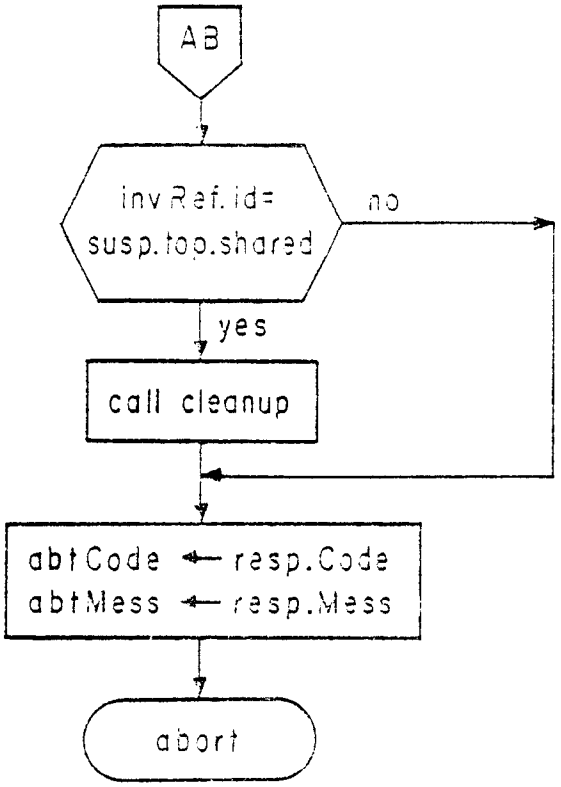

figure 4-19: Abort Termination

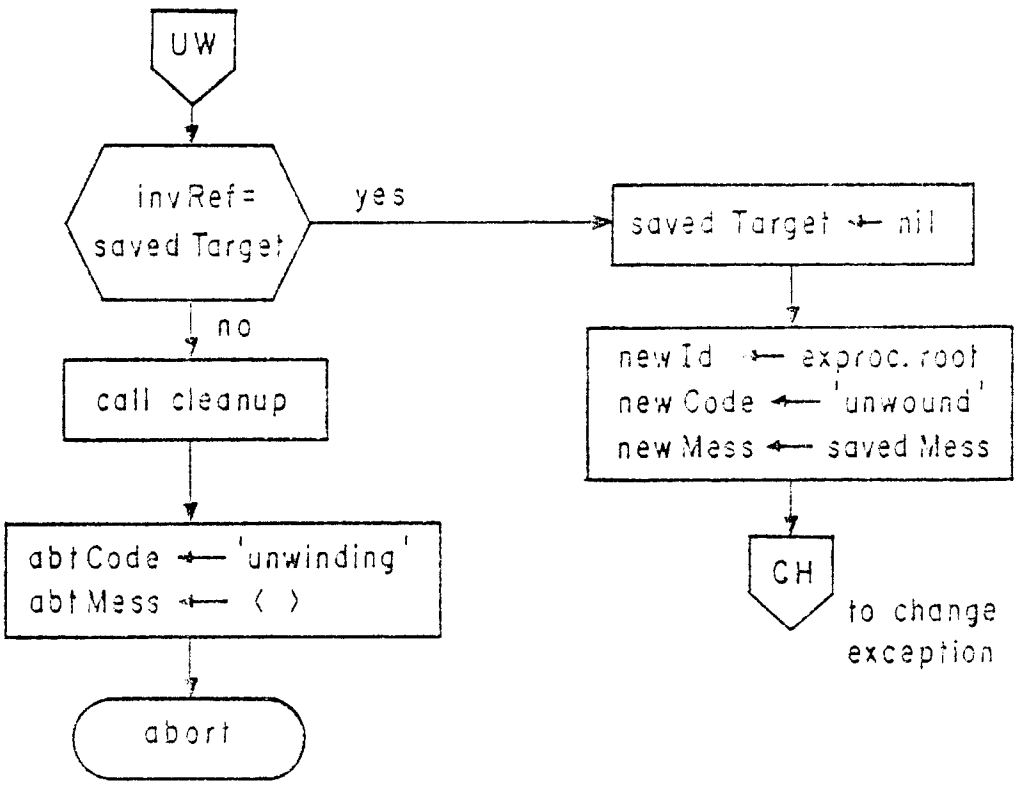

figure 4-20: Unwind Propagation

4.4.5 Unwind Termination

A handler can request that the computation be unwound to an earlier state. This may require the termination of several subsystem activations. The request passed to the exception processor includes the unwind indication, a non-local address space reference, and an exception message. The non-local reference identifies the target of the unwind. The exception message will be passed to the target as the message of an 'unwound' exception. To effect the unwinding of the process, the exception processor may have to abort several subsystem activations. Each activation should be given a chance to cleanup before it is terminated.

The exception processor must propagate the unwinding from subsystem activation to subsystem activation. To propagate the unwinding process, the exception processor will use a distinguished exception code. Every exception processor activation will detect that unwinding is in progress by checking for the 'unwinding' exception before it begins handler selection. The target and the message for the target are saved in the exception processor's incarnation segment during unwinding.

Figure 4-20 illustrates unwind propagation. If unwinding is under way, the test before handler selection (see figure 4-11) will detect the 'unwinding' exception and transfer to the unwinder. The unwinder checks first to see if the unwind target has been reached. If so, the unwinder changes the exception code to 'unwound' and sets the signaller-id to itself. The new exception is eventually reported to the target of the unwind by calling the target's 'unwound' handler.

If the current invoker is not the target of the unwinding, the unwinder calls the cleanup routine which was discussed in the last section. When the cleanup routine has finished giving the invoker a chance to restore its state, the invoker is aborted. The abort code is 'unwinding' and the exception message is empty. The signaller of the abort will be the exception processor. The new exception processor activation responding to the abort will detect the 'unwinding' code and propagate the unwinding another level in the suspended stack.

Having discussed how unwinding is propagated until the target is reached, we must explain how unwinding is initiated by a handler termination request. Figure 4-21 gives the algorithm for initiating unwinding. The handler requesting the unwind passes the target reference and message in the results it returns to the exception processor. The target reference must be validated before the exception processor can begin unwinding. The target reference may be an obsolete address space reference. If so, the exception processor can signal a 'badUnwind' to the invoker by changing the exception.

It is possible for a handler to request unwinding after unwinding is already in progress. The interactions between independent unwind requests must be sorted out by the exception processor. A simple rule is to always aim for the most distant unwind target. In order to sort out multiple unwind requests, the exception processor maintains a record of the current target. If the new unwind request is closer than the current target, it is ignored and unwinding is continued from the current position. If the new unwind target is more distant, the new unwind request is honored and recorded in the incarnation segment. Note

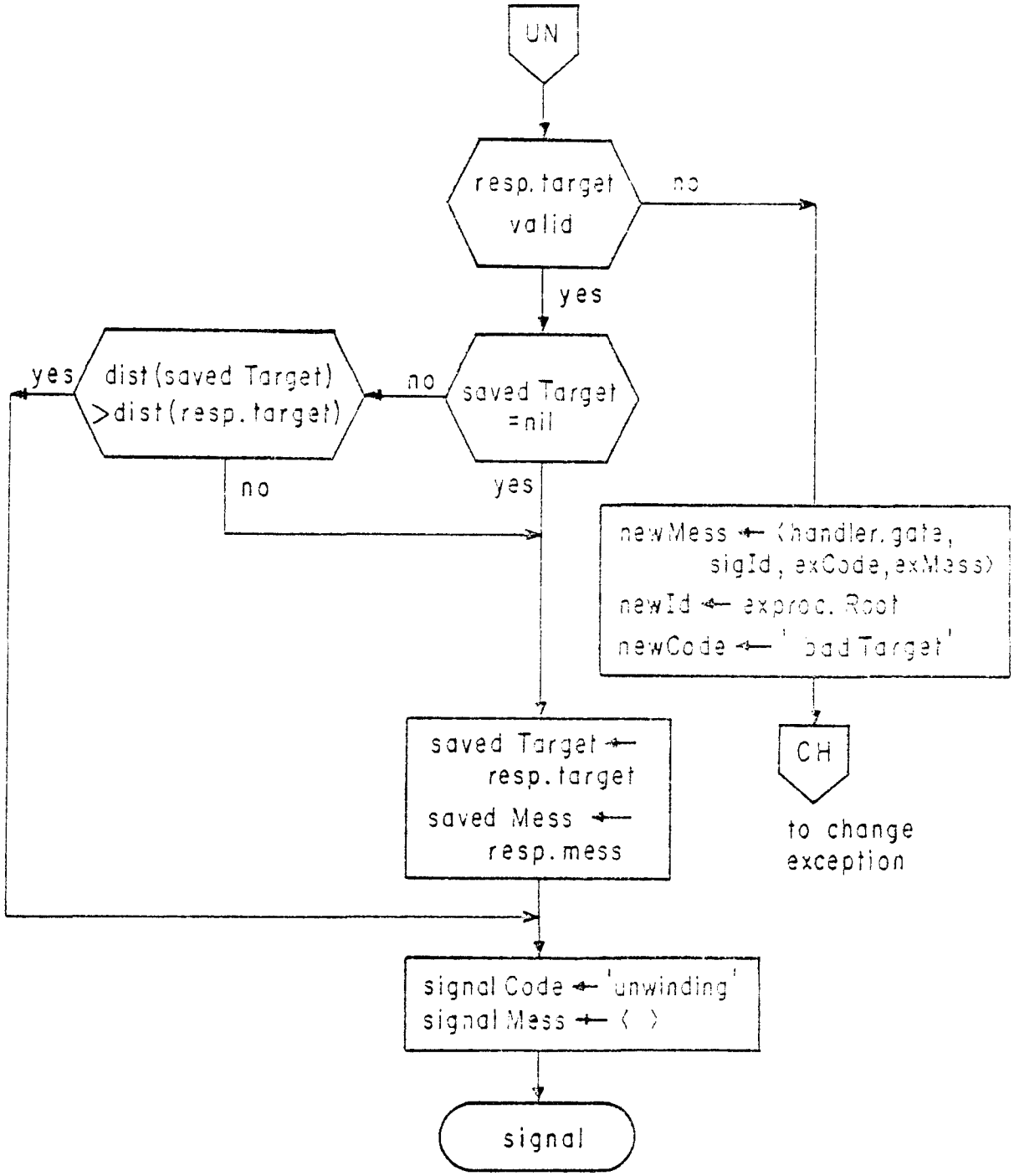

figure 4-21: Unwind Initiation

that whichever target is used, there is an exception processor activation between the current activation and the target which was working on the previous unwind. The intervening exception processor activation will be aborted after being given a chance to cleanup. The exception processor does not need any 'cleanup' handlers. The only exception processor state which can interact with other activations is the unwind target in the incarnation segment. (The fault data buffer is only used during the entry sequences and entry sequences are not subject to unwinds.)

The unwind handler termination initiates unwinding by signalling 'unwinding' after updating the saved target and message. The signal will re-initiate the exception processor. The 'unwinding' exception code will divert the exception processor to its unwinding routine. The exception processor should signal 'unwinding' in order to insure that the invoker reference used to select the cleanup handlers is correct. The current invoker reference may not refer to the subsystem which called the exception processor if the exception processor was activated by a direct call.

4.4.6 Reclassify Exception

A handler may request that the exception be reclassifed to reflect the handler's determination to characterize the exception differently. The handler selects the new exception code and message by returning them with the reclassification request. Figure 4-22 shows how the exception is changed. If the new exception is the same as the current exception, the reclassify sequence transfers to the reject sequence.

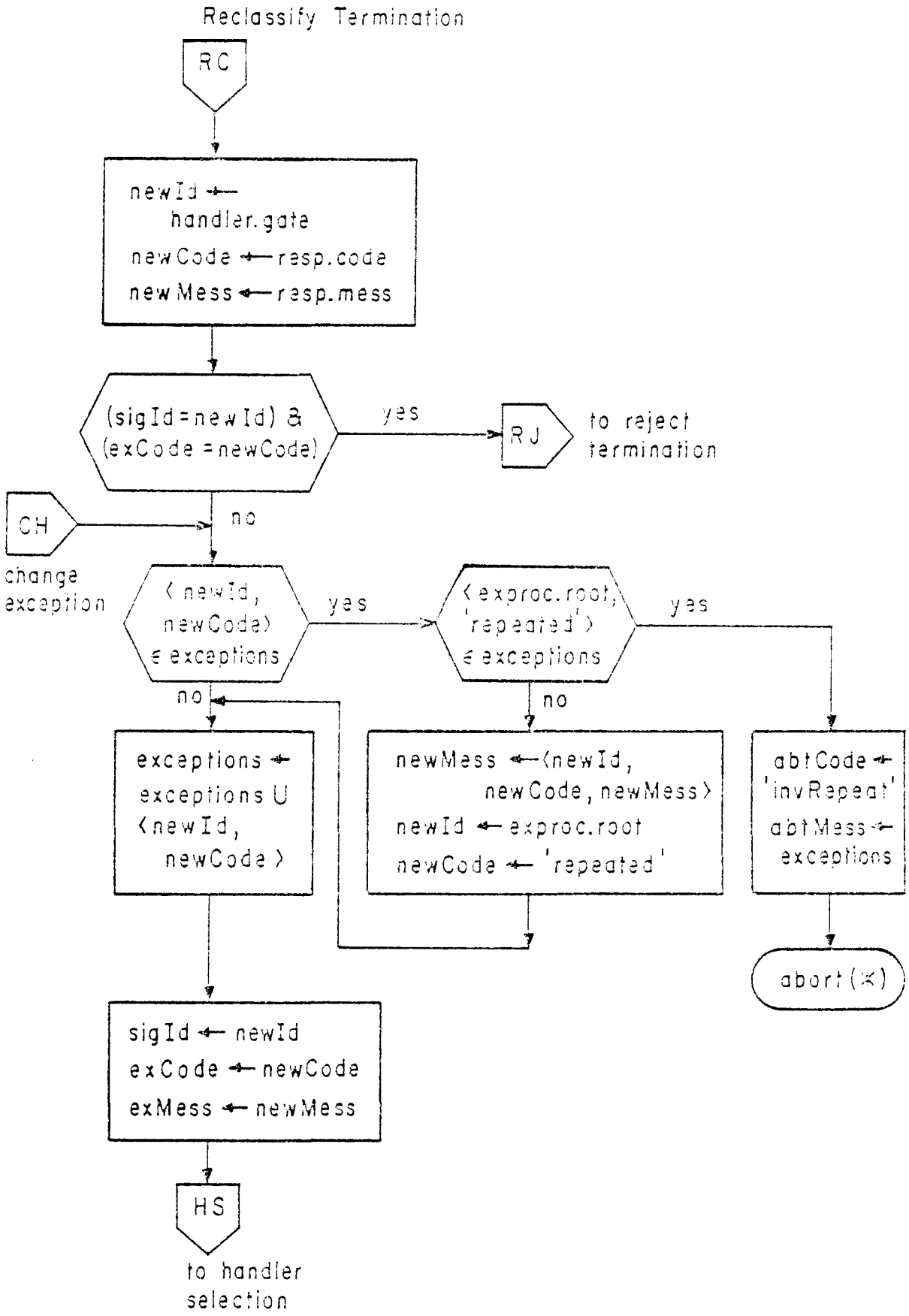

figure 4-22: Reclassify Termination / Change Exception

Part of the reclassify sequence is also used by the exception processor when it changes the exception. Reclassify and change exception should detect attempts to change the exception to be the same as it was earlier in the exception episode. If the new exception is not among the set of already encountered exceptions, the current signaller-id, exception code, and exception message are updated and control is passed to the handler selection sequence.

If the new exception is the same as some earlier exception in the same exception episode, the reclassification should not be allowed. If the new exception was accepted, an endless loop through the same sequence of handlers and exceptions would probably occur. To detect repetition, the reclassify sequence maintains a set of encountered exception names. The number of exceptions remembered can be bounded or virtual memory buffers can be acquired as needed. Before changing the exception, the current exception is added to the set and the new exception is checked to see if it is in the set. If the new exception has already been encountered, the exception is changed to 'repeated' unless there has already been a 'repeated' exception. If a repeated 'repeated' exception is detected, the invoker is aborted with an 'invRepeated' exception.

4.4.7 Reject Termination

A handler can reject responsibility for dealing with an exception and indicate that other handlers for the same exception should be located and activated. A handler termination request with only the reject indicator will cause the exception processor to search for the next

handler for the current exception. Handler requested reclassifications which do not change the exception are converted to reject terminations. Figure 4-23 shows how the exception processor searches for additional handlers using the 'nextHandler' procedure.

If 'nextHandler' finds a handler specification, the reject sequence transfers to the handler call sequence to activate the new handler. If there are no more handlers for the current exception, the reject termination sequence changes the exception. The new exception will be 'noHandler' and the message will contain the original signaller-id, exception code, and exception message. The 'nextHandler' procedure can encounter the usual handler selection exceptions. The handler selection exceptions were discussed in section 4.3.2.

4.5 Conclusions

In this chapter we have developed an exception processor implementation supporting the invoker controlled handler choice policy and a variety of handler termination actions. The implementation consists of the exception processor entry sequences, handler selection, handler activation, and the termination actions. Several interesting implementation issues were discussed. The fault entry sequence, handler specification data structures, and the use of the exception processor itself to process exceptions caused by exception processor activations are the most interesting aspects of the exception processor implementation.

The entry sequences, in which machine level and software reported exceptions are mapped to a common interface, raises interesting issues

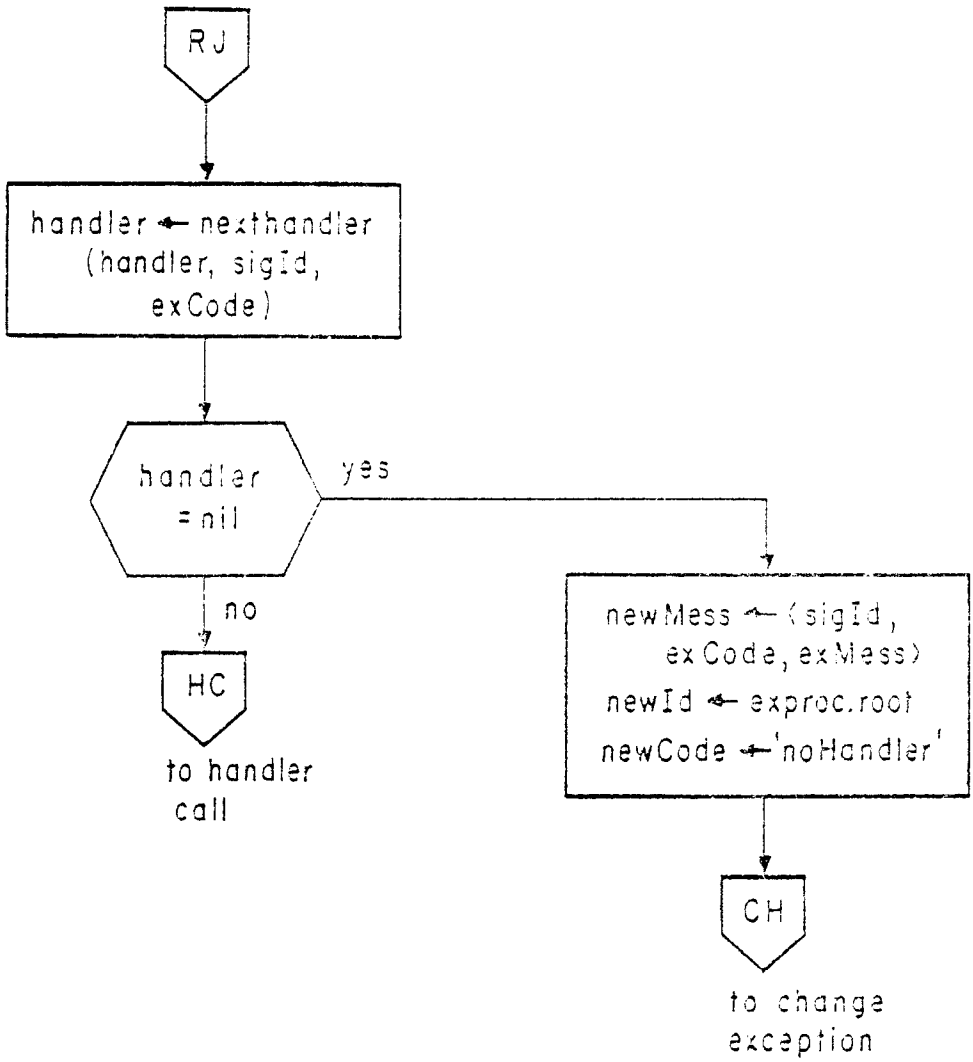

figure 4-23: Reject Termination

concerning the preservation of fault information in the face of additional faults. The complexity of the interactions between different exception processor activations and the system fault handlers is somewhat disturbing but manageable. Using a set of fixed fault buffers, the exception processor entry sequence succeeds in preserving the fault information despite virtual memory and suspended stack full faults.

Handler specifications are interesting because they are supplied by the invoker but processed by the exception processor. The invoker controlled handler choice policy leads to three sets of handler specifications. Some redundancy in the handler specifications is necessary in order to make them safe for the exception processor. The check fields, which prevent the exception processor from looping indefinitely, are a simple yet effective mechanism for protecting the exception processor from malformed data structures.

The most interesting aspect of the exception processor implementation is its ability to sustain a variety of exceptions and to recover from the exceptions using its own exception processing facilities. The exception processor depends upon itself to select and activate handlers for the exceptions which it causes. There is no circularity because the handler specifications used to control the response to an exception are drawn from the root segment of the invoker. When the exception processor is the invoker, its own handler specifications select exception processor gates or system handlers to recover from the exception. The ability of the exception processor to process its own exceptions also raises our confidence in the effectiveness of the handler selection policy and termination actions which were proposed in Chapter Two.

Chapter Five

Summary and Conclusions

5.1 Thesis Summary

In this thesis we have developed an exception processing facility for a system composed of interacting, but mutually suspicious subsystems. We began with the assumption that an exception is the reported failure of a subsystem (or the basic processor) to deliver its specified results in response to a call from another subsystem. We argue that the party most directly affected by an exception is the invoker of the failed operation. Because the invoker directly depends on the outcome of the failed operation, we are led to investigate exception processing strategies which focus on the invoker to determine the course of action in response to the exception.

The called subsystem is the one which determines that it cannot perform the requested operation. Since the failing subsystem cannot complete the call successfully, we argue that it should be retired once it reports its own failure. Terminating the signaller's activation simplifies exception processing since the eventual handler of the exception does not need to be responsible to both the signaller and the invoker. The handler which eventually responds to the reported exception should be initiated in the same way as a subsystem operation. This permits the handler to operate in its own environment and also allow parameters describing the exception to be passed without inventing new communica-

tion protocols.

A major thrust of this thesis is the idea that the invoker of the failed operation, being the subsystem most directly affected by the failure, should control the selection of the handler which responds to the exception. Permitting any other subsystem to control handler selection violates the principles of programming generality either by making use of global information or by reflecting the dynamic state of the computation. In either case, the response to an exception would be controlled by circumstances outside the purview of the invoker. If information derived from other than the invoking subsystem is used to select the handler, the semantics of the calls which the invoker makes will depend upon which handlers are selected in case of exceptions. This makes it hard to define the semantics of a call to the invoker without considering the global or dynamic conditions controlling the selection of the handlers.

When considering how the invoker's control over the selection of a handler can be specified, we were led to consider three classes of handler specifications: default, local, and imposed handlers. Default handler specifications are intended to supply the "system standard" response whenever there are no local or imposed handlers. Local handlers allow the programmer to supply the response to an exception. Imposed handlers allow the system to supply the handler for system sensitive exceptions which are related to virtual machine interfaces or global resource utilization.

In order to provide mechanisms by which default and imposed handlers are specified, we discussed the program preparation process and

the relationship between the program in preparation and the subsystems which perform the actions of preparing the program for execution. Language translators, for example, can enforce the run time environment and provide standard responses to anticipated exceptions by supplying imposed and default handler specifications. The subsystems which participate in the preparation of a subsystem can add default and imposed handler specifications to the subsystem representation when they are called to work on the not yet executable subsystem. The imposed handlers reflect the right of the supervisory subsystem to have first crack at an exception. Default specifications relieve the implementor of having to supply a response for every possible exception. Because imposed and default handler specifications are supplied during program preparation, they are statically associated with the subsystem and do not lead to global or dynamic dependencies at run time.

Besides selecting and activating the exception handler, an exception processing facility must control the resumption of normal processing following the completion of handler execution. Supporting a variety of termination modes allows the handler to have some control over the continuation of the interrupted computation. Based on the decision of the handler, the exception processing facility may resume the invoker, continue exception processing by calling additional handlers, or propagate the exception by reporting the failure of the invoker to the invoker's invoker. Of course, the exception facility should enforce invoker specified controls over the terminations allowed for each handler.

The handler can resume the invoker following the failed operation, at the failed operation (retry), or at some other point in the invoker's

program. At the handler's discretion, exception processing can be continued either by selecting another handler for the same exception or by changing the exception and selecting a handler for the new exception. The handler can also abort the invoker in order to propagate the exception to the invoker's invoker. This causes the invoker's activation to be terminated without regaining control. If the results of several subsystem activations are no longer needed because of the exception, the handler may request that the exception be propagated several levels back in the subsystem call sequence. The various handler termination modes reflect different outcomes of the handler's attempts to deal with the exception.

Chapter Three describes a processor model supporting the exception facility described above and in Chapter Two. The processor model supports protected subsystems and exception processing. Basic processor operations for calling and returning from subsystems along with virtual address space separation between subsystem activations protect subsystems from mutual interference. In order to maintain isolation between subsystems and also support exception processing, the processor must supply several exception processing operations. The signal operation allows a subsystem activation to report its own failure and, at the same time, terminate its activation. The signal operation causes a distinguished subsystem, the exception processor, to be activated in order to control the processing of the exception. Additional operations, used only by the exception processor, are needed to support some of the termination modes.

Chapter Four presents the implementation of the exception processing subsystem. This subsystem is responsible for selecting and activating the handler. The exception processor also implements the termination actions which reflect the outcome of the handler's attempts to recover from the exception. The exception processor decouples the handler selection policy and termination protocols from the basic processor mechanisms for reporting exceptions and for transfering control from one subsystem to another.

Besides making exception processing facilities available to other subsystems, the exception processor makes use of its own exception processing facility to control recovery from the exceptions which the exception processor itself causes. The exception processor implementation is almost completely insensitive as to whether it is invoked on behalf of itself or for some other subsystem.

This thesis defines an exception as the reported failure of an operation to produce its specified results and side effects. Invoker control over the response to a reported exception was shown to be important. Imposed, local and default handler specifications permit flexible system and user control over the selection of a handler. A variety of handler termination modes was developed to reflect the various outcomes of the handler's attempts to recover from the exception. The implementation of the exception processing facility was partitioned into processor level operations for reporting exceptions and for transfering from one subsystem to another, and the exception processor subsystem which implements and enforces the handler selection policy and the termination protocols.

5.2 Some Directions for Further Research

We have discussed the design of a system level exception processing facility for protected subsystems which interact thru calls and returns. This thesis has not addressed the problems of exception processing in a non-procedural environment. Real systems contain interacting processes as well as interacting procedures. When some subsystem in one process depends on results or work being done in another process, the failure of the second process should be communicated to the proper party in the first process.

Communication between processes can be based on polling or interrupts. Polling requires co-operation from both processes and does not lead to any unusual transfers of control. Under a polling protocol, processes must check for the occurrence interprocess events. Interrupts, on the other hand, force a transfer of control in the target process. For processes which are composed of sets of protected subsystems, interrupts pose many problems. Unfortunately, there are a number of situations which to not lend themselves to a polling solution. These applications include the management of external devices, the user console attention key, interval or watch dog timers, and some resource preemption problems. Exception processing in a multi-processing environment would seem to be related to the interrupt facility.

Handling asynchronous events without violating the protection of the subsystems of a process is not an easy task. The selection and activation of the handler (interrupt routine) pose a number of interesting questions. If the subsystems of the process are hierarchically organized, the determination of whether to allow a forced transfer of control

can be based on the relative positions in the hierarchy of the currently executing and the interrupt handling subsystems. Organizations which do not require a static hierarchy among the subsystems should also be considered. Other considerations include the necessity of synchronizing the manipulations of data shared by interrupt handlers and main line routines. The development of a facility for controlling the response to asynchronous events in a multi-processing environment with protected subsystems would be most interesting and useful.

Another area which we have not pursued is the embedding of exception processing facilities into a programming language. A syntax for exception declarations (in the signaller) and handler specifications (in the invoker) must be developed. The syntax should permit the programmer to lexically separate the main line from the exception handling code. In strongly typed languages (e.g. Euclid, CLU, Alphard), exceptions must be declared by the signaller. The signaller can then export the exception name along with the type declaration for the exception message. In typed languages, the formal parameters of the handler procedure should be checked to insure compatibility with the types of the actual exception parameters. The syntax of the handler specifications should allow the programmer to control which termination modes will be allowed for the handler. It is important that the syntax make it easy to tell what will happen in case of an exception.

5.3 Concluding Remarks

Our investigations of exception processing in computer systems have led to several conclusions. A single, uniform exception reporting and

processing facility allows independently developed subsystems to cooperate without risking their individual integrity. Terminating the signaller of the exception is important because it simplifies the exception processing environment. Since the signaller does not need to be resumed, the handler can operate strictly on behalf of the invoker.

Perhaps the most important conclusion we have drawn from our investigations is that it must be possible to determine statically which handlers will be called to deal with the failure of a particular operation invocation. This leads to the conclusion that the handler specifications used to select a handler at run time should be associated (statically) with the invoker of the failed operation. Two problems which have traditionally led to non-local control over the response to an exception are 1) the desirability of providing for default actions whenever the implementor fails to specify a handler, and 2) the fact that the "system" must supply the response to some exceptions (e.g. page fault or real resource problems)

By considering the program preparation process as a sequence of transformations on a not yet executable subsystem we are able to statically associate default and imposed handler specifications with the subsystem. At the programming language level, an analogous concept is the use of a standard prologue to define parts of the language environment. Static handler specifications make it possible to determine exactly what will happen in response to the failure of any operation invocation. The final subsystem representation contains the information necessary to tell what will happen. Also, the program text and the published specifications of the supervisory subsystems provide the same information in a

more convenient form.

Default and imposed handler specifications allow the subsystems which participate in program preparation to define and enforce the run time (virtual machine) environment without inventing new or different exception reporting and processing protocols. Any system level exception processing facility which does not allow for default and imposed exception handlers will be inconvenient (no defaults) or will require additional mechanisms to force control to system handlers when certain failures occur.

The implementation of an exception processing facility can be decomposed into a set of low level subsystem transfer operations for initiating and terminating exception processing and a subsystem responsible for the overall control of the exception episode. The exception processing subsystem can implement and enforce the handler selection policy and control the termination of the exception episode if it is activated by the exception reporting operation. Because the processor level exception reporting operations initiate a new activation of the exception processor, the exception facility can deal with exceptions during an exception episode. The invoker controlled handler selection policy and the suggested episode termination modes are sufficient to support the processing of exceptions caused by the exception processor during exception processing. The ability of the exception processor to deal with its own exceptions demonstrates the utility and versatility of the exception facility.

The analysis of exceptions during exception processing exposes exception processor dependencies on other system facilities. We were

unable to eliminate dependencies between the exception processor and the virtual memory system and between the exception processor and the processor. If the processor cannot activate the exception processor following a processor fault, it must abandon the process. The exception processor depends upon the virtual memory system to repair virtual memory faults while the virtual memory system depends on the exception processor to call it in response to virtual memory faults. Careful design of the exception facility is necessary to avoid true circularities at the base levels of the system.

The importance of an exception processing facility stems from the fact that in order to be robust, a program must be able to deal with failures encountered during execution. The robust program must specify the response to the failure of the operations which it invokes. The response may be to report failure to the next level or to somehow circumvent the failure. In any case, the important point is that the response to a reported failure should be specified by the program which initiated (called for) the failed operation. The invoking program is the one which is most directly affected and the one best qualified to control the response to the failure.

A uniform and flexible exception processing facility encourages the programmer to think about and supply responses for run time failures. An exception processing facility helps the programmer deal with unusual or exceptional cases by providing a mechanism for separating the main line actions from the recovery and special case code needed to handle operation failures at run time.

References

[ALGOL 60] Nauer, P. (ed.), "Revised Report of the Algorithmic Language ALGOL 60," Communications of the ACM, Vol. 6, No. 1 (January 1963), pp. 1-17.

[Ambler 77] Ambler, Allen L., et.al., "GIPSY: A Language for Specification and Implementation of Verifiable Programs," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 1977, pp. 1-10 (reprinted in ACM SIGPLAN Notices, Vol. 12, No. 3 (March 1977), and in ACM Operating Systems Review, Vol. 11, No. 2 (April 1977), and in ACM Software Engineering Notes, Vol. 2, No. 2 (March 1977)).

[Anderson 76] Anderson, T. and Kerr R., "Recovery Blocks in Action: A System Supporting High Reliability," Proceedings 2nd International Conference on Software Engineering, October 1976, pp. 447-457.

[Baker 72] Baker, F.T., "Chief Programmer Management of Production Programming," IBM Systems Journal, Vol. 11, No. 1 (1972), pp. 56-73.

[Bensoussan 72] Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics Virtual Memory: Concepts and Design," Communications of the ACM, Vol. 15, No. 5 (May 1972), pp. 308-318.

[Bernstein 71] Bernstein, A.J., and Sharp, J.C., "A Policy Driven Scheduler for a Time-Sharing System," Communications of the ACM, Vol. 14, No. 2 (February 1971), pp. 74-78.

[Bjork 72] Bjork, L.A. and Davies, C.T.Jr., "The Semantics of the Preservation and Recovery of Integrity in a Data System," IBM System Development Division, San Jose, TR02-540, December 1972.

[BLISS] BLISS-11 Programmer's Manual, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., undated.

[Bobrow 72] Bobrow, D.G. et al, "TENEX, a Paged Time Sharing System for the PDP-10," Communications of the ACM, Vol. 15, No. 3 (March 1972), pp. 135-143.

[Boehm 75] Boehm, B.W., McClean, R.K., and Urfrig, D.B., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," Proceedings International Conference on Reliable Software, Los Angeles, Calif., April 1975 (reprinted in ACM SIGPLAN Notices, Vol. 10, No. 6 (June 1975), pp. 105-113).

[Brooks 75] Brooks, Fred P. Jr., The Mythical Man Month, Addison-Wesley, Reading, Mass., 1975.

[CAL 69] CAL-TSS Internals Manual, Computer Center, University of California, Berkeley, November 1969.

[CAL 69b] CAL-TSS User's Guide, Computer Center, University of California, Berkeley (November 1969).

[CAP 76a] CAP Hardware Manual, Version 1.3, ed. C.J. Slinn, University of Cambridge Computer Labortory, Cambridge England, July 1976.

[CAP 76b] CAP System Programmers Manual, Version 1.8, ed. C.J. Slinn, University of Cambridge Computer Labortory, Cambridge England, June 1976.

[CAP 76c] CHAOS Manual, Version 1.1, ed. C.J. Slinn, University of Cambridge Computer Labortory, Cambridge England, September 1976.

[CLU 75] Schaffert, C., Snyder, A., and Atkinson, R., The CLU Reference Manual (revision 0), Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., June 1975.

[Dahl 70] Hahl, Ole-Johan, Myhrhaug, B., and Nygaard, K., "SIMULA 67 Common Base Language," Norwegian Computing Center, Oslo, Norway, 1970.

[Daley 68] Daley, R.C. and Dennis, J.B., "Virtual Memory, Processes, and Sharing in MULTICS," Communications of the ACM, Vol. 11, No. 5 (May 1968), pp. 306-312.

[Denning 76] Denning, Peter J., "Fault-Tolerant Operating Systems," Computer Science Dept. TR-175, Purdue University, West Lafayette, Indiana, April 1976 (to appear in Computing Surveys special issue on software reliability, 1976).

[Dennis 66] Dennis, J.B. and Van Horn, E.C., "Programming Semantics for Multi-programmed Computations," Communications of the ACM, Vol. 9, No. 3 (March 1966), pp.143-155.

[Dennis 68] Dennis, Jack B., "Programming Generality, Parallelism and Computer Architecture," Computation Structures Group Memo No. 32, Project MAC, Massachusetts Institute of Technonogy, Cambridge, Mass., 1968 (early version reprinted in Proceedings IFIP 1968, North Holland, Amsterdam, 1968, pp. C1-C7).

[Dijkstra 68] Dikjstra, Edsger W., "The Structure of "THE" Multiprogramming System," Communications of the ACM, Vol. 11, No. 5 (May 1968), pp. 341-356.

[Dijkstra 72] Dijkstra, Edsger W., "The Humble Programmer," 1972 ACM Turing Award Lecture, Communications of the ACM, Vol. 15, No. 10 (October 1972), pp. 859-866.

[Dijkstra 74] Dijkstra, Edsger W., "A Time-wise Hierachy Imposed on the Use of a Two-level Store," EWD 408 unpublished, 1974.

[Dijkstra 76] Dijkstra, Edsger W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[ECL 72] ECL Programming Manual, Ben Wegbreit et.al. eds., Center for Research in Computing Technology, Harvard University, Cambridge, Mass., September 1972.

[England 74] Endland, D., "Capability Concept, Mechanism and Structure in System 250," IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August 1974, pp. 63-82.

[Euclid 77] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.L., "Report On The Programming Language Euclid," ACM SIGPLAN Notices, Vol. 12, No. 2 (February 1977).

[Fabry 68] Fabry, R.S., "Preliminary Description of a Supervisor Organized around Capabilities," Quarterly Progress Report No. 18, Institute of Computer Research, University of Chicago, Chicago, Illinois, 1968, pp. 1-97.

[Fabry 73] Fabry, R.S., "Dynamic Verification of Operating System Decisions," Communications of the ACM, Vol. 16, No. 11, November 1973, pp. 659-668.

[Fabry 74] Fabry, R.S., "Capability-Based Addressing," Communications of the ACM, Vol. 17, No. 7, July 1974, pp.403-411.

[Ferrie 74] Ferrie, J., Kaiser, C., et.al., "An Extensible Structure for Protected Systems," IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August 1974, pp.83-105.

[Gligor 76] Gligor, Virgil D., "A Study of Extensible Architectures," Ph.D. Thesis, University of California, Berkeley, Calif., 1976.

[Goldberg 73] Goldberg, Robert P., "Architectures of Virtual Machines," Proceedings NCC 1973, AFIPS Press, Montvale, New Jersey, 1973, pp. 309-318.

[Goodenough 75] Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation," Communications of the ACM, Vol. 18, No. 12 (December 1975), pp. 683-696.

[Goodenough 75b] Goodenough, J.B. and Gerhert, S.L., "Toward a Theory of Test Data Selection," Proceedings International Conference of Reliable Software, Los Angeles, Calif., April 1957 (reprinted in ACM SIGPLAN Notices, Vol. 10, No. 6 (June 1975), pp. 493-510).

[Graham 71] Graham, Martin, "Time as a Control Parameter," personal communication, 1971.

[Gray 72] Gray, J.G., Lampson, B.W., Lindsay, B.G., and Sturgis, H.E., "The Control Structure of an Operating System," IBM Research Report RC3949, July 1972.

[Hoare 73] Hoare, C.A.R. and Wirth, N., "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, Vol. 2, pp. 335-355 (1973).

[Horning 74] Horning, J.J., Randell, B. et al, "A Program Structure for Error Detection and Recovery," International Symposium On Operating Systems, Colloques IRIA, April 1974, Roquencourt, France, pp. 177-193.

[HYDRA 74] Cohen, Ellis (ed.), et.al., "Hydra User's Manual (Preliminary Version)", Dept. Computer Science, Carnegie Mellon University, November 1974.

[IBM 370] IBM System/370 Principles of Operation, IBM System Reference Library, GA22-7000-4, Poughkeepsie, 1974.

[Janson 74] Janson, Philippe A., "Removing the Dynamic Linker from the Security Kernel of a Computing Utility," MS Thesis, Project MAC TR-132, Massachusetts Institute of Technology, Cambridge, Mass., 1974.

[Jones 73] Jones, Anita K., "Protection in Programmed Systems," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa., June 1973.

[Knuth 68] Knuth, Donald E., The Art of Computer Programming, Volume 1 / Fundamental Algorithms, Addison-Wesley, Menlo Park, Calif., 1968.

[Lampson 69] Lampson, B.W., "On Reliable and Extendable Operating Systems," NATO Science Committee Workshop Material, Vol. II, Rome (October 1969) (reprinted in State of the Art Report 1, Infotech Ltd., Maidenhead, Berks, England, 1971).

[Lampson 71] Lampson, B.W., "Protection," Proceeding Fifth Annual Princeton Conference on Information Sciences and Protection Systems, Princeton, N.J., March 1971, pp.437-443 (reprinted in ACM Operating Systems Review, Vol 8, No. 1 (January 1974), pp.18-24).

[Lampson 74] Lampson, B.W., "Redundancy and Robustness in Memory Protection," Proceeding IFIP 1974, North Holland, Amsterdam, 1974, pp. 128-132.

[Lampson 74b] Lampson, B.W., Mitchell, J.G., and Satterthwaite, E.H., "On the Transfer of Control Between Contexts," Lecture Notes on Computer Science 19, Springer-Verlag, Berlin, 1974.

[Lampson 76] Lampson, Butler W. and Sturgis, Howard E., "Reflections on an Operating System Design," Communications of the ACM, Vol. 19, No. 5 (May 1976), pp. 251-265.

[Lauer 74] Lauer, H.C., "Protection and Hierarchical Addressing Structures," IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August 1977, pp. 137-148.

[Levin 75] Levin, R. et al, "Policy/Mechanism Separation in Hydra," Proceedings Fifth Symposium on Operating Systems Principles, Austin, Texas, November 1975 (reprinted in ACM Operating Systems Review, Vol. 9, No. 5 (Special Issue 1975), pp. 132-140).

[Levin 77] Levin, Roy, "Protection Structures for Exceptional Condition Handling," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa., June 1977.

[Lindsay 73] Lindsay, Bruce G., "Suggestions for an Extensible Capability-Based Machine Architecture," International Workshop on Computer Architecture, Grenoble, France, June 1973.

[Liskov 74] Liskov, Barbera and Zilles, Stephen, "Programming with Abstract Data Types," ACM SIGPLAN Notices, Vol. 9, No. 4 (April 1974), pp. 50-60.

[Liskov 76] Liskov, Barbera, "Exception Handling," CLU Design Note 60, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., August 1976.

[Lynch 74] Lynch, H.W., and Page, J.B., "The OS/VS2 Release 2 System Resources Manager," IBM Systems Journal, Vol. 13, No. 4 (1974), pp. 274-291.

[McJones 73] McJones, Paul, "CRMS APL Processor Reference Manual," Center for Research in Management Science, University of California, Berkeley, Calif., February 1973.

[Melliar-Smith 77] Melliar-Smith, P.M. and Randell, B., "Software Reliability: The Role of Programmed Exception Handling," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 1977, pp. 95-100 (reprinted in ACM SIGPLAN Notices, Vol. 12, No. 3 (March 1977), and in ACM Operating Systems Review, Vol. 11, No. 2 (April 1977), and in ACM Software Engineering Notes, Vol. 2, No. 2 (March 1977)).

[Miller 73] Miller, Lance A., "Harlan Mills on "The Psychology of Quality," IBM Research Report RC 3779, May 1973.

[Morris 73] Morris, J.H.Jr., "Protection in Programming Languages," Communications of the ACM, Vol. 16, No. 1 (January 1973), pp. 15-21.

[MPM 75] Multics Programmer's Manual - Reference Guide, Section VII, Honeywell Information Systems Inc., 1975.

[Needham 71] Needham, Roger M., "Handling Difficult Faults in Operating Systems," Proceedings Third Symposium on Operating Systems Principles, Stanford University, October 1971, pp. 55-57.

[Needham 72] Needham, R.M., "Protection Systems and Protection Implementations," AFIPS Conference Proceedings 41, Vol. 1, 1972, pp.572-578.

[Noble 68] Noble, J.M., "The Control of Exceptional Conditions in PL/I Object Programs," Proceedings IFIP 1968, North Holland, Amsterdam, 1968, pp. C78-C83.

[Organick 71] Organick, E.I. and Cleary, J.G., "A Data Structure Model of the B6700 Computer System," SIGPLAN Notices, Vol. 6, No. 2 (February 1971), pp. 83-145.

[Organick 72] Organick, Elloitt I., The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Mass., 1972.

[OS/VS2 75] OS/VS2 Supervisor Services and Macro Instructions, IBM Systems Reference Library, GC28-0683-1, Poughkeepsie, 1975.

[Panzl 76] Panzl, David J., "Test Procedures: A New Approach to Software Validation," Proceedings 2nd International Conference of Software Engineering, San Francisco, Calif., October 1976, pp. 477-485.

[Parnas 72] Parnas, D.L., "A Technique for Software Module Specification with Examples," Communications of the ACM, Vol. 15, No. 5, (May 1972), pp. 330-336.

[Parnas 76] Parnas, D.L. and Wurges, H., "Response to Undesired Events in Software Systems," Proceedings 2nd International Conference on Software Engineering, (October 1976), pp. 437-446.

[PL/I 74] OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, IBM Program Product Library, GC33-0009-3, White Plains, 1974.

[Popek 74] Popek, Gerald J. and Goldberg, Robert p., "Formal Requirements for Virtualizable Third Generation Architectures," Communications of the ACM, Vol. 17, Vol. 17, No. 7 (July 1974), pp. 412-421.

[Randell 71] Randell, B., "Operating Systems: the Problems of Performance and Reliability," Proceedings IFIP 1971, North Holland, Amsterdam, 1971, pp. I100-I109.

[Redell 74] Redell, David D., "Naming and Protection in Extendible Operating Systems," Ph.D. Thesis, University of California, Berkeley, 1974 (reprinted as Project MAC TR-140, Massechussetts Institute of Technology, Cambridge, Mass., November 1974).

[Ritchie 74] Ritchie, D.M. and Thompson, K., "The UNIX Time-Sharing System," Communications of the ACM, No. 7 (July 1974), pp. 365-375.

[Robinson 75] Robinson, L. et al, "On Attaining Reliable Software for a Secure Operating System," International Conference on Reliable Software, April 1975 (reprinted in ACM SIGPLAN Notices, Vol. 10, No. 6 (June 1975), pp. 267-284).

[Ross 67] Ross, D.T., "The AED Free Storage Package," Communications of the ACM, Vol. 10, No. 8 (August 1967), pp. 481-492.

[Rotenberg 74] Rotenberg, L.J., "Making Computers Keep Secrets," Ph.D. Thesis, Project MAC TR-115, Massachussetts Institute of Technology, Cambridge, Mass., February 1974.

[Saltzer 75] Saltzer, J.H. and Schroeder, M.D., "The Protection of Information in Computer Systems," Proceedings of IEEE, Vol. 63, No. 9 (September 1975), pp. 1278-1308.

[Schroeder 72] Schroeder, Michael D., "Cooperation of Mutually Suspicious Subsystems," Ph.D. Thesis, Project MAC TR-104, Massachusetts Institute of Technology, Cambridge, Mass., September 1972.

[Shils 68] Shils, A.J., "The Load Leveler," IBM Research Report RC2233, October 1968.

[Simon 68] Simon, Herbert A., "The Architecture of Complexity," (reprinted in Simon, Herbert A., The Sciences of the Artificial, MIT-Press, Cambridge, Mass., 1968.

[Spier 73] Spier, M.J., et.al., "An Experimental Implementation of the Kernal / Domain Architecture," Proceedings Fourth Symposium on Operating Systems Principles, Yorktown Heights, N.Y., October 1973 (reprinted in ACM Operating Systems Review, Vol. 7, No. 4 (October 1973), pp.8-23).

[Sturgis 73] Sturgis, H.E., "A Postmortem for a Time Sharing System," Ph.D. Thesis, University of California, Berkeley, Calif., 1973 (reprinted as Xerox PARC Technical Report TR 74-1 ,1974).

[Thomas 75] Thomas, R.H., "JSYS Traps -- A TENEX Mechanism for Encapsulation of User Processes," Proceedings NCC 1975, AFIPS Press, Montvale, New Jersey, 1975, pp. 351-360.

[Thompson 74] Thompson, K. and Ritchie, D.M., UNIX Programmers Manual, fifth edition, June 1974.

[VM 74] IBM Virtual Machine Facility / 370: Introduction, Idm Systems Reference Library, GC20-1800, Poughkeepsie, 1974.

[Walker 73] Walker, R.D.H., "The Structure of a Well-Protected Computer," Ph.D. Thesis, University of Cambridge, Cambridge, England, December 1973.

[Wasserman 77] Wasserman, Anthony I., "Procedure-Oriented Exception-Handling," University of California at San Francisco, Laboratory of Medical Information Science Report, 1977.

[Wegbreit 74] Wegbreit, Ben, "The Treatment of Data Types in EL1," Communications of the ACM, Vol. 17, No. 5 (May 1974), pp. 251-264.

[Weinberg 71] Weingerg, Gerald, The Psychology of Computer Programming, Van Nostrand Reinhold Co., Computer Science Series, New York, 1971.

[Wulf 73] Wulf, W. and Shaw, M., "Global Variable Considered Harmful," ACM AIGPLAN Notices, Vol. 8, No. 2 (February 1973), pp. 28-34.

[Wulf 74] Wulf, W., et.al., "HYDRA: the Kernel of a Multiprocessing Operating System," Communications of the ACM, Vol. 17, No. 6 (June 1974), pp. 337-345.

[Wulf 76] Wulf, William A., London, Ralph, and Shaw, Mary, "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology," Carnegie-Mellon TR, June 1976.

[Zilles 74] Zilles, Steve N., "Working Notes on Error Handling," CLU Design Note 6, Project MAC, MIT, Mass., Cambridge, January 1974.