TEMPORAL LOCALIZATION OF ERROR RECOVERY IN OPERATING SYSTEMS BY RESTRICTING INFORMATION FLOW

By

JONATHAN M. ASURU

Bachelor of Science

University of Lagos

Lagos, Nigeria

1979

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fullfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1985

Thesis
1985
H86lt
cop.2

OKLAHOMA STATE UNIVERSITY LIBRARY

TEMPORAL LOCALIZATION OF ERROR RECOVERY IN OPERATING SYSTEMS BY RESTRICTING INFORMATION FLOW

Thesis Approved:

F. E. Heart
Thesis Adviser

D. W. Grace

J P Chandler

Norman N. Durham
Dean of the Graduate College

1236316

PREFACE

This study focuses on how to confine error recovery to the immediate environment of a failed computation (process) by restricting information flow through the system. A module called a manager that restricts the access of operations (procedures) to shared data representation is proposed. The use of descriptors to represent address variables (pointers) and procedure parameters is also proposed to restrict the amount of information available to a procedure. A linguistic mechanism to define recoverable data and inverse procedures (procedures that reverse the actions of another procedure) to undo completed actions is presented. A system data structure that defines a recovery environment to support system implemented recovery is presented.

I wish to thank my committee members Dr. S. A. Thoreson and Dr. K. Davis for their contributions and advice. My special thanks goes to Dr. G. E. Hedrick, my major adviser, for his patience, encouragement, and for his assistance throughout my stay at Oklahoma State University. May the Almighty God bless them all.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

With the increasing use of computers in very critical environments such as aircraft control, electronic fund transfer systems, and electronic switching systems a new and important attribute is being required of the software for these systems. This attribute is reliability. A reliable software is one that can detect and recover from most common errors. The current approach to software fault tolerance in those critical systems is by massive redundancy in code, data, and hardware components. General purpose systems cannot afford this level of redundancy because of prohibitive cost. In order to be able to develop reliable software, the operating system itself must be designed to be robust and also provide facilities for the development of fault tolerant software systems.

A possible iterative approach for building fault tolerant software is:

1. design an algorithm that meets the specification of the software assuming an error-free execution environment;
2. derive constraints for the correctness of the algorithms;
3. add algorithms to check for the violation of the

constraints;

4. specify action to be taken when a constraint violation is detected; and

5. repeat steps 1-4 as necessary.

The algorithm developed in step one may contain some design flaws. Some of the flaws can be detected and corrected by applying a correctness proof on the algorithm or during system testing. It is also possible that after testing and application of correctness proof some design flaws still remain which show up in the production environment. Even when all errors and design flaws have been eliminated, a hardware failure can cause a failure of the software. The algorithm to implement step three constitute the error detection code. It may take the form of assertions, or the form of audits and processes that monitor the execution of other processes depending on the type of error being detected [22]. The algorithm for step four is the error recovery code. A recovery action makes the system oblivious of the error by either correcting the error, or repairing the cause of the error or by restoring the system to a previously known error-free state (checkpoint). A recovery action takes the system to a point where normal operation can continue. A software system that is developed by following steps 1-4 is called fault tolerant or reliable software. The code and data that are used to implement steps three and four constitute the redundancy in the software. The inclusion of redundancy increases the cost of

developing and using the software. Development cost is increased from writing more code than is necessary. The usage cost is incurred by executing more code than is necessary in error-free cases.

Error recovery is pivotal for the reliable operation of a fault tolerant software. Error recovery involves changing the state of the data item(s) affected by an error to a state where reliable operation can proceed. To reduce the cost of error recovery the system state (system data) affected by an error should be made as small as possible. This can be achieved through robust system design. The design dimension include system decomposition technique and information exchange mechanisim between modules. The two dimensions taken together can create unforseen interrelationships between modules which can make successful error recovery increasingly difficult. A system decomposition method which reduces the objects accessible to a module and which employs explicit mechanism for intermodule interaction can reduce error recovery cost.

Problem Statement

An operating system is an important piece of software in a computer system. It is the administrator of the run-time environment of a computer system. The complexity of an operating system is such that it is very difficult to eliminate all errors during system testing. Errors undetected during system testing may manifest themselves in

production environment due to rare combination of circumstances, such as unanticipated resource usage pattern. Some errors due to timing are difficult to detect during testing because of the difficulty in making up the test data that can cause such errors. This implies that one cannot be absolutely confident that a complex software such as an operating system is error free. To provide reliable operation, extra code and information for error detection and recovery must be included in the design.

Both error detection and recovery algorithms are redundant actions because they are not necessary in a fault-free system. During error recovery all interactions the faulty process participated in must be identified and undone to restore the system to a state where normal processing can continue. In a concurrent system the success of any error recovery effort hinges on the correct determination of the faulty environment. A faulty environment is the set of processes directly or indirectly affected by an error. The structural relationship of the processes, the form of interaction (information exchange), and the size of the faulty environment affects the cost of recovery. This study is focused on how to reduce the cost of recovery by restricting the information flow among processes.

The temporal localization of error recovery in operating systems is focused on the restriction of information flow. It is concerned with the confinement of

error recovery to the immediate environment of a failed process. An effective means to determine the immediate environment is required. The restriction of information flow can help define a damaged environment. It does not involve a determination of the cause of an error in order to repair or correct the error. The restriction of information flow serves a dual purpose. The first is that it ensures that errors do not have wide coverage. The coverage of an error is the number of execution environments (processes) affected by an error. The second is that it can reduce the cost of error recovery.

There are two aspects of information flow control addressed in the study. The first concerns a method for restricting the amount of information a procedure can access. The objective here is to contain the actions of a process within its immediate environment.

The second aspect is concerned with how to detect and to recover from control errors for processes that retain their execution histories to ensure that future actions are consistent with their recent past. A control error is an incorrect scheduling decision. Processes that belong to this class include resource schedulers. A recovery action involves a repair or a restoration of the system to a previous consistent state to prevent further damage. Error repair involves identification of the cause of an error, determining its location, and then fixing it. The restoration of the system to a consistent state involves

saving enough state information to reconstruct the system state. The study is concerned with the state restoration aspect of error recovery.

The main contribution of this study are:

1. a part of an operating system called a manager to allow the sharing of abstract objects, and

2. the definition of a recovery context.

The manager separates the protection domain of the shared object and the object operations (procedures) and also partitions the data space into shared data and control data with each data subspace being manipulated by specific program units within the manager. This helps to localize error recovery by restricting the source of an error to one subspace. The recovery context contains necessary information for the operating system to initiate error recovery on behalf of a failed process. The recovery context also serves the purpose of defining an immediate recovery environment.

The main thrust of this study has been defined. In the next chapter related work in operating system and software reliability is reviewed. The third chapter focuses on information control mechanisms. Chapter IV focuses on error detection with emphasis on control error and recovery mechanisms. A summary of the study and concluding remarks are presented in Chapter V.

CHAPTER II

LITERATURE REVIEW

The correct operation of a computer system depends on the proper functioning of the hardware components and the program modules comprising the operating system. Hardware components have employed correcting codes such as parity codes, Hamming code, and self-checking circuits to detect and correct errors in transmitted information [43]. Time-outs are also employed to detect malfunctioning hardware devices.

Residual design faults have been attributed to many errors in software systems [38]. Design faults are defects in the specification of a program and its implementation algorithms. They are present from the beginning and manifest their presence when executed with some input data values and some rare combination of circumstances. Design faults in a hardware algorithm can also cause a software program to malfunction. Endres [16] analysis of errors in IBM DOS/VS operating system revealed that interaction among processes and implementation of the design decisions are significant causes of errors. Some of these errors are due to incorrect process switching, incorrect resource allocation, necessary interrupts not responded to, etc.

Some implementation errors were attributed to failure to initialize/reset variables on entry/exit from procedures. A similar investigation of errors in real-time systems by Glass [20] showed that timing errors, omitted logic, and implementation errors were prevalent.

The advent of multiprograming systems increased the protection problem. Processes must be protected from activities of each other to prevent interference. Some operating systems employ two modes of operation called the user state and the supervisor state. This simple protection model protected the operating system from errors in the user programs. The operating system cannot protect itself against its own errors. Example systems include IBM/360 and IBM/370 series of machines. Protection mechanisms such as file password, and memory protection keys have been used to guard unauthorized access to objects and crossing of address space boundaries. These separate protection mechanisms for different system resources increases the complexity of the operating system. This makes verification difficult [37].

The use of capabilities have been advocated as an alternative approach to protection, resource and process control [13, 37, 44]. A capability is an indirect pointer to an object with the permissible access rights on the object explicitly specified. In this scheme the right to access a resource is by possessing a capability for it. Capabilities are used to define execution domain of a proccess. Linden [35] proposed small protection domains as

a means to realize reliable and secure operating systems. In a small protection domain model, a process executes in different execution domains as it runs. Examples of operating systems based on capabilities are Hydra [45], CAP [44], and Intel's iMAX [27]. Capabilities are useful for interdomain addressing. Interdomain protection is one way of restricting information flow. However, there is the problem of intradomain addressing such as overwriting data areas through address variables, redefining parameters in a procedure thereby gaining access to more area of the calling environment. These are some subtle ways of incorrect information flow.

The use of redundant code and data have been employed to detect errors in software. Redundant code is in the form of audit programs and process monitors. Data redundancy is the use of additional data fields for the purpose of error detection and correction. The common form of data redundancy are counts, identifier fields and extra pointers [8]. Data redundancy is widely used to provide software fault tolerance. An audit program is a software module that is responsible for ensuring that a system's data structure is in a consistent state. An audit program is invoked periodically to check the state of a storage structure to detect any erroneous state. An audit program has two functions. One function is to detect an erroneous state in a storage structure while the second is to correct the erroneous state by making use of redundant information in

the storage structure. A process monitor observes the execution of another process to detect incorrect behavior such as excessive use of resources(memory, disk, and CPU time) [22]. The DMERT operating system for Bell system 3B20D duplex processor is one that makes extensive use of audit programs and redundant fields in system data structures to enhance error detection [22]. Audits in DMERT do not verify functions. They are limited to checking the integrity of critical data structures and resources. The operating system provides a facility for system processes to request for initialization after an error has been detected throuh a trap mechanism. The environment of DMERT is a duplicated one. The 3B20D system is a duplex system with loose interconnection between the processors. Other hardware components are also duplicated to ensure availabilty. Memory and disk updates are made to both active and redundant systems. The duplicated system state is one major factor for the high degree of reliability and availability in 3B20D/DMERT. The operating system supports a mechanism for software units to be updated through an update facility. However, it does not implement the recovery block concept.

Process pairs have been implemented in some distributed systems. In these systems there are two copies of every process on different processors. These systems are mainly designed to provide recovery from hardware failures [5]. Randell [38] proposed the recovery block construct to

provide fault tolerance in software programs. A recovery block consists of a primary block with alternate blocks and a validation test which must be passed. A recovery block is free from errors if one of the blocks (primary/alternate) passes the validation test. To be useful the primary block and each alternate must be of independent design. This construct provides a mechanism for the temporal reconfiguration of a software system. Thus the scheme improves system availability while providing fault tolerance. The recovery block provides fault tolerance against unanticipated errors (design faults) but its use is limited to functional program units. Since many operating systems programs retain their internal state between activations the recovery block construct must be supported with mechanism to reverse the effects of internal state changes when state restoration is required.

A system structuring technique called data abstraction has been employed in the design of some operating systems [45, 27]. Programs modules based on this concept are called abstract data types. Data abstraction hides the internal representation of a data object from external modules but provides entry points to access the data object through the defined procedures. The procedures of the abstract data type module have full access to the data representation. When data abstraction is implemented with capability-based addressing, the resulting system is a set of protected subsystems [19, 35]. A monitor is a language construct that

enforces the correct sequencing of operations and data access protocol on a shared data object. It is employed in operating systems as synchronization device. A monitor and a semaphore are functionally equivalent but a monitor based process synchronization is robust. In a semaphore based scheme, adherence to resource access rules depends on the voluntary cooperation of the users of the resource. In a monitor based scheme, resource access rules and constraints are handled by the monitor on behalf of the users. A monitor provides the means for the safe sharing of abstract data types in an operating system. A monitor is a special implementation of an abstract type by including synchronization code. A monitor enforces mutual exclusive execution of the operations. The shortcoming of abstract data type implementation is in the area of information access restriction. The procedures of these abstract modules have full access to the encapsulated data. If the procedures are only allowed to access the subcomponents of the data objects they require to complete their execution then, sources of errors can be reduced and error recovery can be made less expensive. A modified data abstraction technique that places the shared data object and the operation procedures in different protection domains but still maintains the integrity of the operations is presented in Chapter III.

Error recovery is pivotal to any system that is to provide reliable operation. It is an important aspect of

localizing errors in a system. The recovery block scheme provides a systematic error recovery which is to restore the program to the state that existed before the current activation of the block. This type of recovery is called backward error recovery. The state restoration involves only the nonlocal variables (parameters and global variables) that the failed block modified. State restoration involving a set of interacting parallel processes can lead to a situation called "domino effect". A domino effect occurs when a roll-back of a failed process causes a roll-back of another process which causes further roll-back and so on. It has been shown that a system of interacting parallel processes is free from domino effect if the system is deadlock free [40]. To avoid a domino effect Randell [38] proposes a data transfer mechanism called a conversation. A conversation is a recovery block covering two or more processes [38]. Processes involved in a conversation are required to synchronize their exit. That is a process cannot leave the conversation until all the participating process have passed their acceptance tests. The conversation construct has the potential to deadlock if the process structure exhibit interdependencies which results in complex interactions. Other constructs that have been proposed to coordinate the recovery of a set of cooperating parallel processes are named-link recovery and multiprocess recovery [40].

A named-link recovery block spans one or more recovery blocks while a multiprocess recovery block is a single recovery block spanning one or more processes. In a named-link recovery block, the coupling between processes is loose and this makes avoidance of a domino effect or coversation deadlock difficult. With multiprocess recovery block the linkage between processes is tight. This eliminates both deadlock and domino effect. The disadvantage of the multiprocess recovery construction is that the code for a process is fragmented and scattered. Also, semantics of some constructions are not clear.

An experimental recovery cache to hold recovery data has been implemented on the PDP-11 [32]. The recovery cache provides recovery for only main memory objects and the addresses used are real addresses. Thus the program must not be overlayed during the execution of a recovery block. A software implemented recovery cache with architectural support to speed-up the operation is required to support flexible error recovery.

The majority of failures of computer systems is attributed to transient faults [31] in the hardware and design faults in software. To reduce the overhead processing associated with error recovery, it is necessary that the time for recovery from frequent failure modes be made small. The error recovery time depends on the extent of damage which depends on the constraint on information

flow through the system. The constraint on information flow in turn depends on how the system is structured [3]. To reduce the recovery time the system must avoid implicit interaction and information exchange between processes must be limited to the minimum required for a receiving process to complete its action. Also the actions of a process should not produce unidentifiable side effects. The elimination of unwanted side effects can reduce failures due to some remote causes to a very small proportion. This is to ensure that for most of the time the recovery action performed within the affected process is sufficient to remove the error symptom. How this can be done through information flow restriction is the main thrust of this study.

CHAPTER III

INFORMATION FLOW CONTROL SCHEMES

Information flow pattern among interacting programs can have a significant effect on error recoverability of an operating system. In an uncontrolled information flow environment errors can have wide coverage. This can make error recovery costly and consequently degrade system performance. This chapter is concerned with techniques to restrict the flow of information. The chapter is focused on system structuring and environment control.

System Structuring

An operating system structure has an impact on modifiability, verifiability, and information flow. A well structured operating system should make other modules immuned from changes made to one of its modules. Verifiability is concerned with application of formal techniques to prove the correctness of the system. Information flow concerns the interaction of processes. Modifiability, verifiability, and information flow all affect operating system reliability. To structure an operating system to restrict information flow, relationship between modules comprising the operating system must be well

defined.

The layered approach has been employed in the development of operating systems [15]. In the layered approach the operating system is partitioned into a number of self contained layers. The lowest layer being the kernel. Each higher level layer makes use of the functions provided by the immediate lower level layer. The layered approach leads to the development of modular systems. Also, since each layer is self contained they can be verified and developed independently. The problem with the layered approach is partitioning the system to maintain the strict hierarchical relationship between layers. A structuring technique is proposed in the next section which retains the features of a modular system but in addition improves the least privilege principle. The least privilege states that a procedure should be given the smallest capability it needs to complete its action [19].

Modified Object Manager Structure

The principle of system closure has been cited as the bases for secure and error-tolerant execution environments [13]. The closed system principle states that no process or procedure has any capability which has not been explicitly granted. The implication of this principle is that the effects of all operations on a closed system shall be confined within that system. The ideal situation is a completely isolated and disjointed environments. While this

is not possible because of process interaction the other best alternative is to restrict the interaction.

A structuring method that leads to the development of isolated environments is data abstraction. Data abstraction is a modularization technique that encapsulates a set of data objects and procedures that perform operations on the objects. Access to encapsulated data objects is through the invocation of operation procedures. The operations have full access to the data representation. Constructions similar to data abstraction are monitors, resources in synchronizing resources [1], and packages in Ada* programming language. Architectures and languages that support the use of data abstraction are said to be object based. Some operating systems that have incorporated object orientation in their design are Hydra [45], iMAX [27], and CAP [44]. These operating systems provide a finer degree of protection. The encapsulated data objects and procedures are usually called type managers or object managers. The usual structure of an abstract type module is shown in Figure 1.

The alternative structure divides the abstract type module into two modules. One module contains the shared data object and is called the manager. The second module contains the operations on the abstract type. The two modules exists in separate protection domains. The functions of the manager module are:

1. maintains operations view of the object;

2. makes available the necessary components of an object representation to an operation; and

3. synchronize the operations on the object.

```
type name
     variable declarations;
     statement list;

     procedure opl(parameters)
         body of opl
     end opl

     .
     .
     .

     procedure opn (parameters)
         body of opn
     end opn

 end name
```

Figure 1. Structure of Abstract Data Type Module

To maintain operations view of object representation the manager only needs to identify the components and subcomponents of the data structure used by an operation. To present an operation with the necessary components of an object representation, the manager needs to use the operations view of an object and perform a projection operation similar to a relational database projection operation to construct a sub-object which is then made

available to the operation. Since this sub-object exists in a different memory location, any error that occurs during a type operation affects only the sub-object. Error recovery can be accomplished by discarding the sub-object.

The manager provides synchronization by scheduling operations when the state of the object permits it. Requests for operations that cannot be performed immediately based on the current state of the object are delayed until an enabling state change occurs. The fact that only the manager has access to the object representation further improves the security and integrity of operations. There is no timing error because two operations cannot access the same component of the object representation.

The proposed type manager structure is depicted in Figure 2. The resource variable in the manager module represents the shared abstract object and should be stored in a separate segment from the local variables of the manager. The statement list between begin end pair is an initialization code. The operation module consists of a set of disjoint processes which are invoked by the manager with actual parameters. The operations do not access the shared variables directly. Each process accesses only the information presented to it by the manager plus its local variables.

```
manager:  manager-name

          resource {
            object data structure definition;
          }

          local variable declarations;
          export {list of operation} in op-module;
          manager body;
          begin  statement list; end
end manager-name
```

(A) Manager Module of Proposed Structure

```
operation:  op-module of manager-name

          process:  opl (parameter list)
              body of opl;
          end opl

          .
          .
          .

          process: opn (parameter list)
              body of opn;
          end opn

end op-module
```

(B) Operation Module of Proposed Structure

Figure 2. Manager Structure

Implementation Technique

The implementation proposal is message based. The users of an abstraction send messages to the manager possibly with arguments requesting a type operation. The manager combines the arguments with the necessary components of the object representation before sending it to the

operation process. On completion of an operation the manager updates the state of the shared object and sends a response message to the operation requestor. The form of message communication between a manager and the operation processes should be by reference for efficiency reasons. The form of communication between a manager and the external environment should depend on the architecture. In order not to restrict possible concurrency the operating system should support both blocking and nonblocking communication. A communication is blocking if the sender must wait until the message operation completes. The form of communication between a manager and the operation processes should be nonblocking. The communication between the manager and the external environment should be blocking. A blocking communication has the usual semantics of a procedure call.

There should be a request communication channel for each operation, a communication channel for each response message, and a communication channel for each operation process. A communication channel should have capacity for one message. The channels for operation requests and response messages interact with the external environment while those for operation processes are internal to the object manager. The external channels should include a field which indicates the state of the channel (empty/full). The operating system message manager sets the operation request channel state to full after depositing a request message while the object manager resets it to empty after

consumming the message. The object manager sets response message channel to full while the operating system message manager resets it to empty after delivering the message. The operation process channels should contain a field to indicate the return status for type operations. The manager sets it to a null value while the operation process sets it to a non null value after performing an operation. The value must differentiate between normal and abnormal terminations.

To schedule an operation, the object manager simply scans the request channels until it finds one with status field set to full. If the operation can be performed at the current state of the object it is scheduled and the status field is reset and the scanning continues. To respond to an operation it scans the status field of the operation process channels until it finds one with a nonnull value. It sends the response message and then resets the return status field to null value. The manager is blocked only when it sends a message to the external environment. To reduce storage overhead, the operation process channels can also be used as response channels. This reduces the degree of possible concurrency since in this case the channel is used in a mutual exclusive way.

Analysis of Object Relationship

After the operating system has been structured into a set of object managers, a formal evaluation of the design is

required to determine if the objects preserve a hierarchical relationship. The analysis technique is to define a dependency relation between pairs of abstract objects. As an illustration the relation --> is used to denote an input-output relation among objects. If A and B are any two abstract objects and A --> B holds, then object A is used as input to produce object B. The analysis is to verify that the following properties are satisfied:

1. reflexive: A --> A holds for every object;
2. antisymmetric: If A --> B, then B --> A must not hold.
3. transitive: If A --> B, and B --> C then A --> C holds.

The reflexive property is trivially true. An object can be both an input and output to itself. The antisymmetric property avoids the possibility of infinite recursion. Symmetric relationship makes error recovery difficult because of the difficulty in knowing how the objects have changed each others state. It is also difficult to determine a consistent previous state.
The transitive property defines a flow path between objects. This path is a recovery path to be traversed when an error is detected in an object. The preservation of these properties ensures that the input-output relation do not form a cycle. Cycle formation can increase cost of error recovery because it makes the determination of a consistent previous state very difficult.

To remove a symmetric relationship between any two objects, the objects can be merged into a single object with a new manager. The formal approach is only part of the design process and should be used whenever new objects are defined.

The specification of a manager must include:

1. legal sequences of operation invocations; this includes precedence, exclusive, and parallel constraints on operation invocations;

2. blocking and nonblocking communications; and

3. type of operations on each message channel (i.e send, receive, and send-receive).

A manager must be subjected to formal proof to verify its logical correctness. The verification of a manager need not take the usual form of sequential proof (i.e., loop termination) because a resource manager or scheduler can be implemented with a nonterminating loop. What is required is to show that the control decisions conform to the specification. The verification should consist of proving:

1. that precedence and exclusive constraints on operation invocations are satisfied;

2. that the right mode of communication is employed at interaction points; and

3. that the correct message operation is applied on each message channel.

There are two types of precedence constraints to consider. The first is total precedence constraint. If A and B are two operations and A precedes B always, then the precedence constraint is total. To prove total precedence constraint between two operations A and B it must be shown that the following four conditions hold:

(a) precondition(A) /\ precondition(B) = false;

(b) precondition(A) /\ postcondition(A) = false;

(c) postcondition(A) /\ precondition(B) = true;

(d) precondition(B) /\ postcondition(B) = false.

The first condition prevents the simultaneous invocation of both operations. Conditions (b) and (d) ensure that neither operation succeeds itself. The third condition ensures that B is invocable after A.

The second type of precedence constraint is a partial precedence constraint. An object that exhibits a partial precedence contraint is a stack with the operations push and pop. The invocation constraints are:

exclusive: push, pop;

precedence: push; (push | pop)*

The precedence constraint specifies that a pop operation cannot be invoked without a previously completed push operation. To prove conformity to partial precedence constraint it is only required to show that condition (c) holds. Also both total and partial precedence proofs must show that precondition(A) becomes true before

precondition(B) does.

Let S be the set of operations on an object and let x be an exclusive operation. Let R = S - {x}. To verify that the execution of the operation x is exclusive, it must be shown that if precondition(x) is true, then for all y in R precondition(y) is false. The verification of correct communication mode and correct application of message operations on message channels do not require formal proof. They can be accomplished by visual inspection by comparing the specification with the implementation code. This can also be complemented with compile-time and run-time checking.

Environment Control

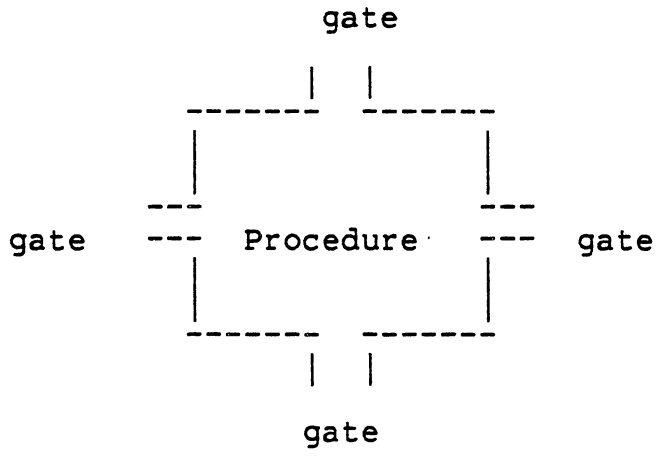
Environment control is concerned with establishing a reliable execution environment for programs. An operating system must provide isolated process environments in order to reduce the scope of an error. To achieve this an operating system should prevent interference among processes. Processes can interfere through improper resource control, process control, and protection mechanism. Also, environment isolation requires that the privilege to access an environment should not be implied from the trustworthiness of a process as is usually the case in systems based on hierarchy of privileges. Protection mechanism based on privilege level implies an inclusion property. A process of higher privilege can access the environment of a process with lower privilege without

restrictions. The implication of this is that an error in a high privilege environment can affect lower privilege environment also. The error coverage cannot be easily determined because the information flow pattern is not well defined.

The key to a reliable environment control is avoidance of implicit interaction between environments and within an environment. The issues discussed in this section are protection, resource control and process control.

Protection

A protection mechanism can be envisioned as an environment in which a procedure or process executes within a protection wall having some exit gates. Each gate leads to a different environment. The gates are the means through which the procedure or process can interact with other environments. The gates represent calls to other procedures or access to global data objects or actual parameters from some calling environment.



The way the gates are defined determines the degree of protection provided by the wall. The gates are information flow outlets. Errors also propagate to neighbouring environments through the gates. The wall is strong when the gates are explicitly defined and also few in number. The number of gates depends to some extent on the system decomposition method and also on the size of a program unit. A system decomposed into a hierarchy of self contained subsystems presents a well defined interdependency among subsystems. A robust protection mechanism should prevent errors in one subsystem from extending to other subsystems.

A protection model that supports environment isolation is the concept of protection domains. A protection domain is an environment that defines all the access rights and operations on objects available to a procedure within the domain [35]. The implementation of protection domains requires a means to express the access rights and operations on objects available to a procedure and also a means to check at run-time that a procedure's actions are consistent with its access constraints.

Capability Based Implementation

An elegant scheme for the implementation of protection domains is the capability based scheme. A capability is an absolute address for a virtual object [18]. Capability based addressing is an addressing scheme in which every

system object is addressed through a capability [14]. A capability includes the set of rights permissible on an object. A right that is not granted cannot be exercised by the holder of a capability. Example operating systems whose protection is based on capabilities are Hydra [45], iMAX [27], and CAP [44].

The environment of a procedure is defined by a list of capabilities for the objects it can access. The only means to access an object is by possessing a capability for the object. Thus a process cannot come into the execution environment of another process except by explicit arrangement (passing capabilities). Also, any detected error during a process execution is confined within the interacting environments.

A capability based scheme has an efficiency problem. Perhaps this could be the reason for the paucity of capability machines. The efficiency problem involves both time and space. It takes several words to represent a capability. Thus many memory accesses are required for capability operations. A software implementation of capabilities can slow down the system. A consequence of addressing all objects by capabilities is that once the capability of an object is destroyed, the object is no longer addressable. This is called the "lost key" problem [37]. An alternative implementation of protection domains based on privilege numbers is presented in the next section.

Privilege Number Implementation of Protection Domains

The capability based scheme though flexible cannot be easily implemented in conventional architectures. The reason being that capability based addressing favors machines with object orientation. An alternative implementation of protection domains that is implementable in conventional architectures is proposed in this section. The proposal is based on the use of privilege numbers.

The structuring of an operating system based on the notion of data abstraction creates a set of subsystems which interact through well defined interfaces. Each subsystem is a protection domain and is assigned a unique privilege number. Every data object and procedure in a protection domain is identified with the same privilege number. The difference between this scheme and other privilege mechanisms such as supervisor/user modes and security classes in security sensitive environments is that the privilege number does not imply a nested or an inclusion property. The basis for accessing any object is similar to capability based scheme. A process must possess the privilege number of the object(domain) and a secondary privilege which specifies the subset of the operations the process can perform on the object. The object manager interpretes the secondary privilege since it is object dependent.

The system (operating system/architecture) embeds the privilege number in the data object and procedure descriptor data structures and this can be checked on first access. Since the assignment of privilege numbers is based on the static structure of the operating system, the set of privilege numbers is fixed and few in number. The operating system kernel can maintain a table which indicates the correspondence between privilege numbers and subsystems. A two-byte privilege number is sufficient for both user and operating system needs. The privilege numbers assigned to users can be reused when a user exits the system. This necessitates a second table to store available privilege numbers.

To control the transfer of privileges, access control bits for read, write, copy, and delete are added to the privilege number. Also, to provide a finer degree of access control four additional mask bits are added to mask those rights that cannot be exported by the holder of the privilege. The masking scheme is proposed by Corsini and Frosini [11] for capabilities. Altogether three bytes are sufficient to represent access privilege to an object with the required restrictions.

The advantage of this scheme over a capability based scheme are simplicity, time, and space efficiency. Simplicity comes from the static assignment. Every object (process/data) belongs to exactly one domain identified by a

fixed privilege number. It is time and space efficient because privilege number operations do not require hashing and few memory references are required for access checking.

Use of Descriptors

In some programming languages the addressing environment of a procedure is determined by lexical nesting level, its placement in a source file, and by performing address arithmetic on pointer variables and procedure parameters. Each of these schemes can increase the address space of a procedure. A procedure can modify unrelated locations through the use of pointer variables. This leads to an incorrect information flow. In C programming language where pointer manipulation is similar to array indexing, passing a scalar variable by reference to a procedure exposes adjoining locations of the calling environment to the called environment. The use of descriptors to describe pointers can prevent such exposure.

A descriptor is a control word that describes areas of data and program storage [7]. The important attribute of a descriptor is that it defines the storage area occupied by an object. It can be used to provide finer degree of protection because any access outside the defined area can be detected by the system. Descriptors have been employed to describe arrays and segments in Burroughs B6700 [7]. The use of descriptors to pass procedure parameters can restrict interenvironment interaction. This is because descriptors

can be used to define scalars, array slices and substrings.

A disadvantage of descriptors is that it requires space to store the descriptor information. However, with efficient coding and support for variable length descriptors the space overhead can be made small. In a reliability conscious environment this small extra storage is worth it. A proposed descriptor layout for a pointer variable is

Field	Bits
type: (scalar, array, string)	2
unit-size: (1, 2, 4, 8)	2
length-code: (0, 1, 2, 3)	2
number-of-units	8 | 16 | 24
base-address	24
free bits	2
total	40 | 48 | 56

The unit-size field specifies one of the primitive machine data types: character (1), two-byte integer (2), four-byte integer/real (4), and double word (8). The length code specifies how many bytes used to represent the length of a nonscalar type. If the length code is one, then size of the array/string is the value stored in the next byte of the descriptor. The number-of-units field gives the length of the array/string. The base-address is the location of the first byte of the storage area. One of the extra bits can be used for access control to permit either read or write. When the type field is scalar the number-of-units field is redundant in which case the descriptor size is reduced to 32 bits. Thus for scalars the use of descriptors do not introduce any storage overhead.

The support for descriptors in an architecture requires two special instructions. The first instruction is for the construction of descriptors and the second is a descriptor decode instruction.

In languages where dynamic memory allocation and deallocation is supported, more than one pointer variable can simultaneously locate the same area. When the area is freed without setting all the associated pointer variables to a null value, subsequent use of the variables can lead to chaos. The area could have been reallocated to a different procedure or could contain garbage. In the first case the procedure is implicitly exposed to an external environment, while in the second situation an incorrect data value is used by the procedure. In either case there is an incorrect information flow. The correction of this type of error is very difficult because of the difficulty in locating the error. The damage to the system state can be extensive.

This type of error can be detected through the use of descriptors and privilege numbers. One of the extra bits in a descriptor can be used to indicate the type of area (dynamic or static) referenced by the pointer. For dynamically allocated storage areas the privilege number of that procedure is prepended to the area. When an area is freed, the contents is cleared and any subsequent reference to that area by the same procedure results in a nonmatching privilege number.

Resource Management Issues

The control of system resources is an important function of an operating system. Resource usage is a major source of interaction between processes in the system. The following are aspects of resource control:

1. maintenance of resource state;
2. keeping track of resource allocation;
3. applying appropriate locks on resources.

The possible states of a resource must be identified. In general terms the states free and allocated must be distinguished. A resource is free if it is not currently assigned to any process and is usable. In a free state, a resource must not contain any information from previous use that can affect another process in an undesired way. This ensures that a process does not inherit any part of another process environment except by explicit arrangement. A resource unit is allocated if it is assigned to a process. The state transitions of resource units must be enforced. In each state the possible state transitions must be defined with their enabling events. For instance, a memory frame changes from free to allocated if these conditions are satisfied:

1. there is a request for memory space;
2. the contents of the frame is cleared;
3. the frame is free.

A state must not be altered before an operation completes. For example, a disk page must not be marked free until it is

deleted from a file map and subsequently cleared.

Keeping track of resource allocation involves maintenance of assignment information. This depends on whether there are identical units of the resource or not. For identical units the allocator must partition the units into equivalence classes based on the units states. One equivalence class must be those units currently allocated to some computations. Another class are those units that are free. Other equivalence classes are possible depending on type of resource. These partitions must be identified and the rules for transition from one partition to another must be enforced by the resource manager. These classes form the possible states of the resource. The sum over the cardinality of each partition must be equal to the number of resource units. This invariant must be true always.

Another aspect of keeping assignment information is associating a resource unit with a process. This requires using some redundant information in the resource descriptor. The approach is to use lock and key scheme. When a resource unit is assigned to a process a lock is generated and included in the resource descriptor. The same lock is given to the process to which the resource is assigned. To a lock holder the lock acts as a key to unlock the resource. The use of the resource is permitted on presentation of the correct key. This ensures that only certified processes gain access to a resource.

As an illustration consider a pool of memory frames controlled by the memory manager. Initially all frames are free and cleared. To allocate a frame the memory manager must take the following steps:

1. select a free frame;
2. generate a lock for the selected frame;
3. place the frame in allocated list;
4. update the resource allocation information; and
5. return (frame-id, lock) to requestor.

To deallocate a frame the following steps must be taken:

1. match key and lock;
2. clear the frame;
3. place the cleared frame in free list; and
4. update the resource allocation information.

In both allocation and deallocation the first step is crucial. If the first step fails in the case of allocation the request cannot be satisfied immediately. If the first step fails in the case of deallocation it represents an error situation and the operation must be rejected. There are three possible states of a frame - free, allocated, and deallocated. The permissible state transitions is shown in Figure 3.

A process should exercise control on a shared data object for the duration of an operation. In the UNIX system, simultaneous editing of a file by two different processes is allowed. An operating system must ensure

integrity of operations while providing concurrency. By viewing file editing as a transaction with many updates that must not be interleaved, the operating system needs to support two different write locks to model the lifetimes of file operations. In edit mode a file should not be shared until it is released at the end of the edit session.

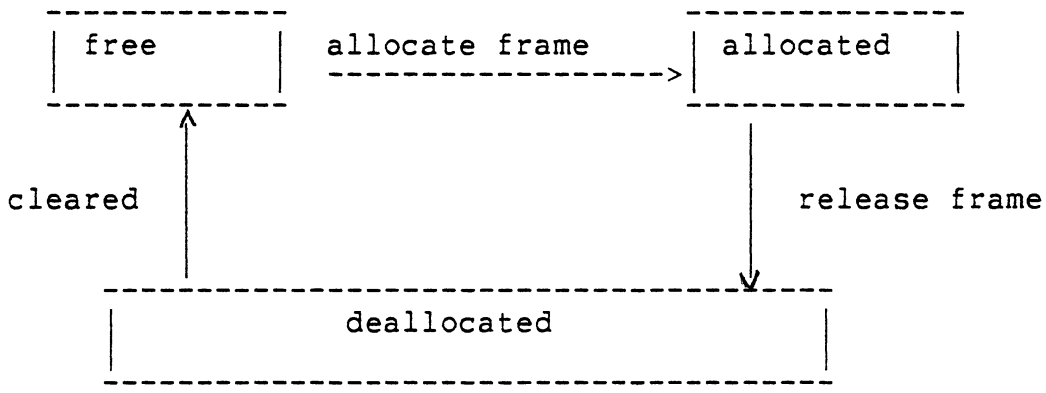


Figure 3. State Transition of a Memory Frame

An example

An important object maintained by an operating system is user files. Let the access (rights) defined on a file object be execute, read, write, edit, and delete. Some systems do not make these distinctions but instead base file operations on read, write, and execute protection used for memory objects. The following file types are distinguished: directory, executable file, stream file, and record file.

The permitted user access for each file type is as follows:

directory:	read, delete;
executable file:	execute, delete;
stream file:	delete, edit, read, write; and
record file:	delete, read, write.

The right to delete a file must be controlled only by the file creator. With this approach, possession of write access by a process does not grant the process the right to delete the file.

To control the use of a file properly by user processes, three types of locks will be admnistered on a file. The locks depend on the semantics of the operations on a file. The operations which do not change contents of a file will be given a read lock. The operations which make a single update at a time will be given a short write lock (swl), while those operations that make many updates at a time will be assigned an exclusive write lock (ewl). The difference in swl and ewl is in the lifetimes of the file usage. A file with an ewl is released after a close request is issued by the current user. The locks for each file operation is:

read:	read lock;
execute:	read lock;
delete:	swl;
write:	swl;
edit:	ewl.

With this information, the manager is be able to prevent an attempt to perform an unauthorized access to a file. It also improves file sharing among users since write access and delete access are separate.

Process Management

The process is the active and schedulable entity in the operating system. It uses various system resources as it performs its action. Processes interact by sharing physical and logical resources. The interaction influences the execution of a process and if not properly controlled processes can interfere with each other in undesired ways. The support for concurrency improves resource utilization and computation speed and also increases the chance of undesired interference. The process control functions are:

1. specification of a process domain; and
2. concurrency control.

The domain of a process is part of the system environment it can sense or alter. This comprises physical resources and virtual objects such as data, code, message channels. A robust means to specify a domain is by use of capabilities or by the proposed privilege number scheme.

Concurrency Control. A concurrency control mechanism must ensure that processes do not interfere with each other in undesired ways. The desirable attributes of a concurrency control mechanism are:

1. noninterference from concurrent execution;

2. proper ordering of operations on shared objects;

3. unsuccessful and incomplete operations should not alter an object's state; and

4. freedom from deadlock.

The noninterference requirement is necessary to prevent the invalidation of an action by one process due to a concurrent action by another process. This is possible if an object is subject to concurrent access. To guarantee noninterference it is imperative that the operating system locks an object for the duration of an operation. Alternatively the operating system must ensure that two processes do not have access to the same parts of a shared object where concurrent access is permitted. This is the approach taken in the proposed manager structure.

The ordering of operations on shared objects is necessary to ensure that operations conform to a legal sequence. Operation ordering is concerned with enforcing dependency constraints between operations and also delaying an operation until a certain enabling event occurs. The proper ordering of operations can avoid extensive backup in the event of an error. The ordering of operations should depend on both the current state of the object and on the type of operation requested. As an example, a FIFO queue object with the operations insert and remove should delay a request for remove operation if the queue is empty. It could allow parallel execution of remove and insert provided there are previously completed insert operations.

The third requirement ensures that an object is not observed in an intermediate state by other concurrent operations. It also ensures that operations that terminate unsuccessfully do not damage the object state. This guarantees that an object moves from one consistent state to another consistent state. The enforcement of this requirement can reduce the scope of an error. This property is refered to as recoverability property in transaction based systems [41].

The fourth desirable property of any synchronization technique is to ensure the entire system is deadlock free. In a fault tolerant operating system a deadlock can be very costly. A deadlock recovery involves rolling back one of the processes involved in a deadlock to a safe state. The determination of a safe state is not simple because interactions must be undone. If a rolled back process has interacted with other processes then the affected processes must also be rolled back. Thus a single rollback can lead to a chain of rollbacks. Taking into consideration the difficulty of deadlock recovery it is safer to adopt a deadlock avoidance policy in the design of a fault tolerant operating system.

Structuring an operating system based on the manager construct reduces this possibility because protocols and constraints on resource use are enforced in the manager. The fact that managers are processes not subject to

exclusive access also reduces the possibility of circular wait which is a necessary condition for deadlock. However, to avoid deadlock there is need to impose a call hierarchy on intermanager communication. A formal technique is necessary to reduce the amount of run-time checks. The following steps are suggested:

1. determine the dependency relations of managers;
2. verify the dependency relations;
3. derive a synchronization graph from the dependencies; and
4. implement a graph manager to enforce the dependencies.

To determine dependency relations of managers it is only necessary to know the communication partners of each manager. Two managers are communication partners if there is a direct information transfer between them via sending of messages. If a manager M can send a message to another manager N, then M > N (read M calls N). It means N obtains input from M. The send operation should not be symmetric. That is if M calls N, then N must not call M. To avoid this the system kernel must support send and receive message primitives.

The verification of dependency is limited to interaction across interfaces. The steps to be taken are:

1. verify the existence of matching communication between partners;
2. derive preconditions and postconditions for

communication;

3. consider all possible states of the managers taken together.

A matching communication occurs when a send statement in one process/manager has a corresponding receive statement in a second process at the point of communication. As an example, consider the processes M, and N:

```
Process M

  . . .
  receive from N (argument list)
  . . .
end M

Process N

  . . .
  receive from M (argument list)
  . . .
end N
```

In this example M and N do not form a matching communication because each is waiting to receive data from the other. In fact both processes are permanently blocked (deadlocked). When a matching pair is established, the next step is to verify that precondition(send) /\ precondition(receive) and postcondition(send) /\ postcondition(receive) are true. This guarantees that the managers will make progress in the absence of deadlock. The final step is to take all managers together by combining all the preconditions(send) and postconditions(receive). The verification in this step is to show that some states are impossible to reach and those reachable are valid. A proof technique for sequential processes is presented in Levin and Gries [34].

The synchronization graph is simply a call graph based on the established dependencies between managers. It confirms or disproves the formal proof of the previous step. The system is deadlock free if the graph is cycle free and the formal proof result shows a deadlock free system. If there is a disagreement between the graph and result of the formal proof the system design should be reviewed and the formal analysis repeated until a deadlock free system emerges. The advantage of the formal analysis is that it is done during the design stage.

The implementation of a graph manager is to provide run-time checks for interprocess communications. The graph manager allows a manager to send a message to another manager if the nodes corresponding to the two managers are adjacent (path length between nodes is 1) and the interaction is a valid transition along the path. The graph manager is able to detect illegal interactions with this approach.

Semaphore Based Process Control. The use of semaphores for controlling access to shared data was proposed by Dijkstra [15]. A semaphore is an integer variable on which two indivisible operations P(S) and V(S) are defined. Given a semaphore S, P(S) is defined as :
If S > 0 then S := S-1 else wait; while V(S) is defined as: S := S+1. A process that wants to access a shared variable x, executes P(S) and if unsuccessful is blocked until S > 0.

On the other hand, if P(S) is successful, then P1 proceeds and after accessing the variable x executes V(S) which frees x. In using a semaphore the synchronization code becomes part of a process algorithm. There is thus no separation between the algorithm defining a process action and the constraints on its execution. This is a drawback of semaphore based process control mechanism. A second drawback is that its use is error prone. If either of P(S) or V(S) step is omitted, there could be concurrent access to a variable or a deadlock. Lastly semaphore programs are not well structured. The use of global variables by concurrent processes expose the processes to erroneous actions of each other. The semaphore based synchronization scheme does not provide a good information flow model.

Monitor Based Process Control. The monitor concept was developed to enforce mutual exclusive access to shared resources. A monitor is a program module with a set of shared variables and a set of procedures that define operations on the shared variables. A monitor is a passive entity. A process that wants to use the resource maintained by the monitor invokes the appropriate monitor procedure that performs the action. The execution of the monitor procedures are mutually exclusive. Hoare [23] proposed the use of condition variable to provide condition synchronization. The operations defined on a condition variable are signal and wait. The condition variable provides the means to order operations.

The enforcement of mutual exclusion in monitor use can have undesirable side effects depending on the implementation scheme. There are two common implementation schemes - disabling of interrupts and use of semaphores [36]. The disabling of interrupts can have a drastic effect on the system if critical device signals are not responded to on time. Interrupt inhibition has the effect of global exclusion on all monitors whereas intergrity of monitor operations only requires exclusion on individual monitors. There are two pontential problems with this approach. The first is the possibility of missing some necessary interrupts and the second is the unnecessary restriction of possible concurrency.

The use of semaphores to control entrance to monitors introduces another level of synchronization. The semaphore approach provides mutual exclusion on individual monitors. The semaphore scheme is prone to deadlock in the presence of nested monitor calls.

To meet the third requirement of synchronization the monitor must take extra steps because it is not structured to meet error recoverability requirement. The steps to be taken by a monitor are:

1. save the current state of the shared data at the commencement of an operation; or

2. provide an inverse procedure to undo the effects of a normal procedure to be called when an error is detected.

If an operation is not invertible then the only option is step 1. Without augumenting a monitor with the extra steps suggested above, processes are not immuned from the effects of an unsuccessful operation by another process. Monitors have the same expressive power with semaphores. The advantage of a monitor over semaphore is that monitors enhace program modularity.

Proposed Manager Scheme. The proposed type manager structure is both a synchronization and a fault tolerant device. The manager only encapsulates a shared data object. The operations on the shared data are implemented as processes independent of each order and the manager. The problem of contention among operations is eliminated because an operation can only be invoked by a manager.

The manager is able to order operations properly because all requests for object operations are channeled to the manager which then calls the appropriate operation. An operation request that cannot be serviced immediately is left in the request message buffer until such a time that the shared object state permits it. Thus operation ordering is effected without scheduling queues associated with monitors and semaphores.

The proposed manager has a recoverability property. Incomplete operations cannot modify the state of an object. The manager is able to distinguish between successful and unsuccessful operations through the return code field in the operation process message. The fact that an operation does

not have an exlusive control of a manager reduces the chance of deadlock. By designing the system of managers to have a hierachical structure deadlock can be avoided. Of the three synchronization techniques the proposed manager scheme is the most elegant.

Summary

Information flow issues that can improve error recovery have been discussed. A modified structure for sharing abstract data objects called a manager is presented. The structure hides the full representation of a shared object from the operations, but presents partial representations to operations. A formal technique for analyzing relationships between subsystems (managers) based on input-output relation is also discussed. The analysis checks if the object relations form a partial order. Proof technique for verifying individual manager scheduling algorithm is also presented. The use of privilege numbers to implement protection domains is proposed as an alternative to expensive capability based implementation. Also presented is a descriptor based scheme for representing pointer variables which can be used to pass parameters in procedure calls.

CHAPTER IV

ERROR DETECTION AND RECOVERY

The reliability of a system is increased by built-in redundancies to detect errors and to recover from the errors. These redundancies take the form of additional components and actions which are not necessary for correct system behavior in a fault free system. The redundancies are unavoidable overheads in a fault tolerant system. The usual forms of redundancy are functional, information, and time redundancies.

Functional redundancy is the use of standby components (program or hardware) to take over when a primary module fails as a result of a fault. The recovery block is type of functional redundancy for software modules. Information or data redundancy is the use of additional data in data structures to detect and correct an erroneous data state. Common forms of information redundancy are counts, identifier fields, and extra pointers. Information redundancy is perhaps the most applied form of redundancy in software systems. Time redundancy is the use of more time to perform an action. This include instruction, and function retries. All forms of redundancy are necessary to produce software of high reliability.

Error Detection

The detection of error in a system is a crucial step in any reliable system. The process involves distinguishing between an acceptable and unacceptable data states in the various stages of a computation. Run-time checks are then applied to check conformance of a program to its specification. Any deviation is then signalled as an error which is then followed by a corrective or recovery action. Since these run-time checks increase execution time of a program these checks should be minimized. The checks can be reduced by restricting them to interaction points such as procedure calls, and interprocess communications. A program module must check the validity of information received from other modules and information transmitted to other program units. Thus every program has at least two points to detect errors. The first is a test of the program's input data and the second is a test of the program's output (See Figure 4).

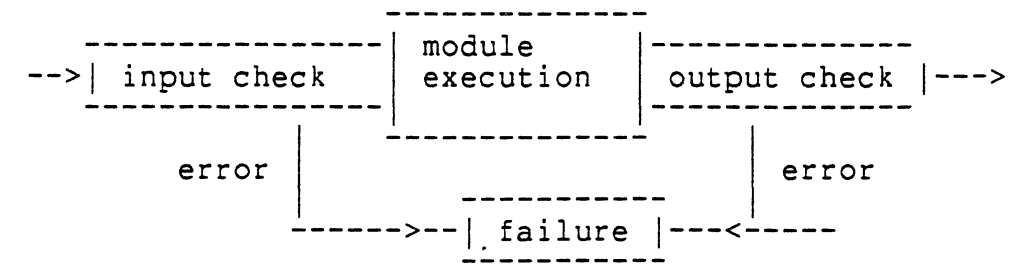


Figure 4. Possible Error Detection Points in a Module

There are three basic methods employed for error detection. These are structural, algorithmic, and temporal methods [31]. The structural method uses a redundant system structure to detect errors. An example is a duplex system operating in parallel. A computation is correct if output from both systems are identical and satisfy an acceptance criteria. The algorithmic method is a dynamic verification scheme. It employs run-time assertions to detect errors. The temporal method consists of monitoring the execution time of processes. A common mechanism is the use of time-outs.

The choice of a method depends on the size and type of a module. A module is classified as either functional or nonfunctional. A functional module does not retain its internal state between activations. The output of a functional module is always dependent on the current input. A nonfunctional module on the otherhand, retains its internal state between activations. Examples of nonfunctional modules are process schedulers, monitors, and resource controllers. The output of a nonfunctional module depends on the current input and the internal state.

An error in a program unit is due to error in the input data, or error in the internal state for a nonfunctional module, or a fault in the algorithm defining a program. The algoritmic fault is due to a design fault in the algorithm (software or hardware), or a hardware failure. The detection of input error can be accomplished by the use of

assertions (validity tests) while audits and other detection methods are required to detect errors in the internal state of a nonfunctional program unit. The use of audits for error detection and correction is described in [8, 22].

The error detection algorithm should be well designed and subjected to formal verification techniques. The verification must show that the code always terminates and does not modify the internal state of the processing program. It must also show that valid states of programs are not rejected and that only erroneous states are rejected. All functional modules should be formally verified. The proof must show that a functional module terminates for both valid and invalid input data. Time-outs can be employed to detect incorrect behavior of any functional program unit induced by a fault in a hardware component.

Control Error Detection in Nonfunctional Processes

A mechanism to detect control errors within a nonfunctional module such as the manager construct proposed in chapter three is presented. A nonfunctional process that schedules the use of a shared object must include exclusive, precedence, and parallel constraints on operations invocations. The method is based on separating the control algorithm from the processing algorithm and then adding

assertion statements affecting only control data to detect any violation of scheduling constraints. The control algorithm is called a "decision-maker". The manager must call the decision-maker to make the next decision based on the current control data state and the requested operation. The manager confirms or rejects the decision by executing an assertion that must be true for the decision made.

The decision-maker implementation consists of defining a finite state control device that models the operations invocation constraints. The state transitions are augmented with preconditions and events relating the control state data. On invocation, the decision-maker either allows the requested operation to proceed immediately, or delays the request, or signal an error if the control state data is in error.

An example is a process that controls the access to stored data. There are two groups of processes that access the stored data called readers and writers [23]. The readers read the current data values and the writers update the state of the shared data. Readers are allowed to proceed in parallel but only one writer is permitted to gain access to the stored data when there are no readers. Also, waiting readers have higher priority than waiting writers at the end of a write and no new readers should be permitted if there are waiting writers.

To model this control problem, the decision-maker defines a finite state control that satisfies the

constraints, a set of decision variables, the actions associated with each state transition, and the preconditions for each state transition. The states of the finite state control and their meanings are:

nrnw: no readers and no writers (initial state);

rnww: reading without waiting writers;

rww : reading with waiting writers;

wnwr: writing without waiting readers; and

wwr : writing with waiting readers. The decision variables are:

nw : number of writers (0 or 1);

nr : number of readers (0 or > 0);

nwr: number of waiting readers (0 or > 0);

nww: number of waiting writers (0 or > 0). The following action codes are defined:

ok : operation can proceed;

qw : enqueue write request in waiting list of writers;

qr : enqueue read request in waiting list of readers;

dqr: allow a waiting reader to proceed;

dqw: allow a waiting writer to proceed. The finite state control is shown in Figure 5. The arcs are labelled with triples (a, b, c), where

a: is a transition number;

b: is a requested operation (r = read, w = write); and

c: is an action (ok, or qw, or qr, or dqr, or dqw).

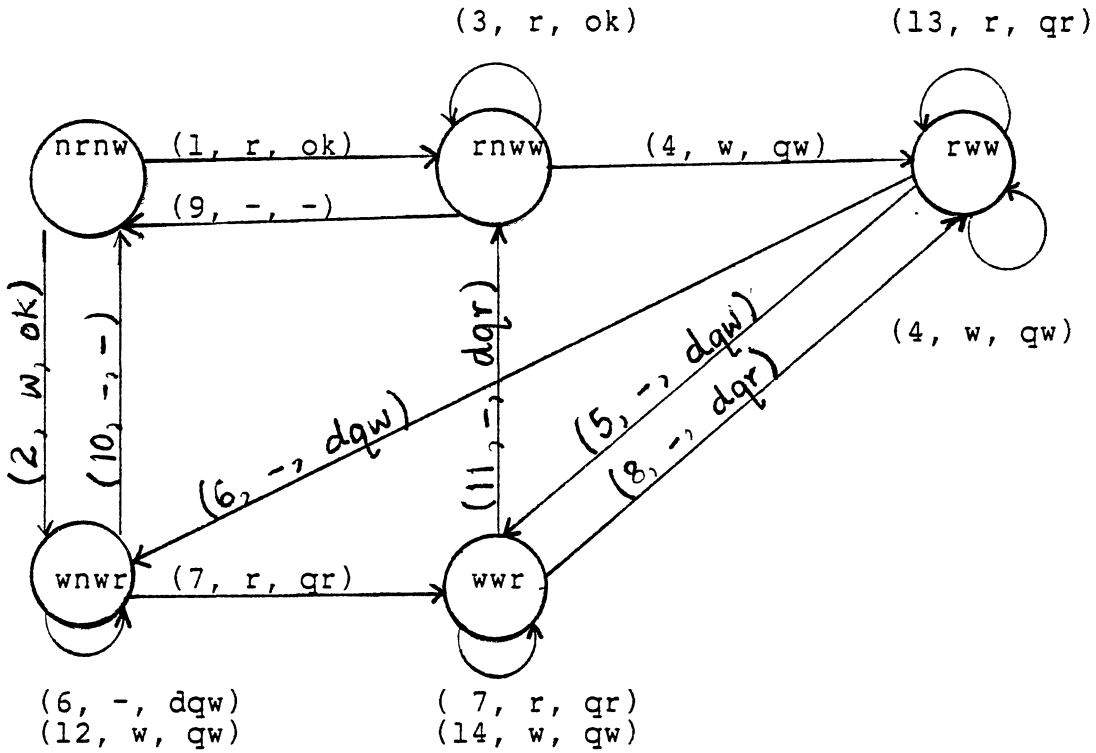


Figure 5. Finite State Control of Readers and Writers Synchronization Problem

The preconditions for each state transition are:

state	transition	precondition
nrnw	1	nw = nr = nwr = nww = 0 & b = r
	2	nw = nr = nwr = nww = 0 & b = w
rnww	3	nr > 0 & (nww = nw = nwr = 0) & b = r
	4	nr > 0 & (nw = nwr = nww = 0) & b = w
	9	nr = nw = nwr = nww = 0
wnwr	7	(nr = nwr = 0) & nw = 1 & b = r
	12	(nr = nwr = 0) & nw = 1 & b = w
	6	(nr = nw = nwr = 0) & nww > 0
rww	6	already defined
	5	(nr = nw = 0) & nwr > 0 & nww > 0
	13	nr > 0 & nww > 0 & nw = 0 & b = r
	4	already defined

wwr	8	(nr = nw = 0) & (nwr > 0 & nww > 0)
	11	(nw = nr = nww = 0) & nwr > 0
	7	already defined
	14	nw = 1 & nwr > 0 & nr = 0 & b = w

In order to be able to recover from control errors, redundant data fields should be added to the headers for the waiting lists. One redundant field that is necessary is the number of elements in the list. Another necessary redundant information is the previous state of the finite state control.

One advantage of this approach is that the internal state of a nonfunctional module is partitioned into control data state and shared data state. With the manager construct the type of error can be determined from the program unit that detects it. The manager and the decision-maker manipulate the control data while the operation procedures manipulate the shared data. This helps to restrict the sources of internal state error and also helps to categorize an error. Errors detected by a manager or a decision-maker are classified as control error, while errors detected by operations and users of the shared data are classified as data error. Another advantage of this scheme is that it facilitates error correction because the program units to check for type of error are known. This has a beneficial effect on error recovery. The error recovery action is restricted to the part of the internal state affected by an error.

Error Recovery

After an error is detected a recovery action must be taken to undo the effects of the error and possibly repair the error. Error recovery is essentially restoring the system to a state where processing can resume. To restore the system to a globally consistent state a knowledge of the recovery space is required. In concurrent systems such as an operating system the interactions of processes can complicate error recovery. An error in one process in a set of parallel processes can lead to extensive state restoration.

There are two types of error recovery mechanisms called backward and forward error recovery. Backward error recovery consists of restoring the system to the state that existed at the beginning of a recovery point. A recovery point is a point in a program where the current state of a program is saved. Backward error recovery provides recovery for unanticipated errors. An unanticipated error is a design error present in a software or hardware from the outset that remained undetected during testing but manifests itself due to rare combination of circumstances. The system programmer does not specify any action for the handling of the error because of lack of knowledge for its existence. These errors are attributed to design faults in hardware and software. Forward error recovery is applied to anticipated errors. The recovery involves a corrective action to remove

the error symptom. Forward error recovery can be handled through exception handling facility of a programming language. These recovery methods are complementary.

The recovery from errors affecting system data structures is handled effectively by audits. Audits can be used to recover from control state error. Since every state of a finite state control has a unique assertion, the control state audit can determine which control variable causes the failure of an assertion. It then uses the redundant fields in the waiting request list and previous state values to correct the error. Suppose the finite state control of Figure 5 is in state nrnw (initial state) and the value of nwr is not zero. The assertion statement for the state nrnw requires the variable nwr to be zero. The variable nwr makes the assertion fail. The audit makes use of the fact that the queue length for waiting readers must be equal to nwr. If the value of this redundant field in the queue header is zero and the header pointer is null, then the error is corrected by setting nwr to zero.

Recovery Block Interface

The recovery block was proposed to provide software fault tolerance against residual design faults in both software and hardware algorithms. The syntax of a recovery

```
block is

    ensure <acceptance test>
    by <primary alternate>
    else by <second alternate>
    .
    .
    .
    else by <nth alternate>
    else error
```

The semantics of the recovery block is as follows: on block entry, the primary alternate is tried. This is then followed by the execution of the acceptance test algorithm. If the test yields true, then the results are accepted and a block exit is taken. However, if the acceptance test fails, backward error recovery is initiated which consists of restoring the program state to what it was before entering the block. The backward error recovery is then followed with an automatic transfer to another alternate and the sequence repeated. If no alternate passes the acceptance test, then the block has failed and an error condition is raised.

One good advantage of the recovery block is that it provides a convenient checkpoint. When recovery blocks are nested, the application of backward error recovery on a failed enclosing block is costly. Consider N nested blocks with Ni alternates per block. The maximum number of trials

for the N blocks is N1 * N2 * ... * Nn. The recovery block interface is proposed to reduce the number of trials in certain situations.

When a block P1 calls another block P2 there are two failure possibilities. The first is the failure of the caller (P1) and success of the callee (P2). The success of P2 implies that the input data passed to P2 by P1 and the results computed by P2 are valid since P2 must pass its acceptance test before sending any output to P1. The semantics of a recovery block requires the reexecution of P2 when the next alternate of P1 is tried. If the result of P2's execution can be saved the repeated execution of P2 can be avoided thereby reducing the cost of error recovery. The second possibility is the failure of the callee (P2). The failure of P2 implies the failure of P1 also. A recovery action must be taken to restore the state of the program on entering P1.

The recovery block interface is only useful for the first failure type (i.e., failure of an outer block). The recovery block interface is a program unit that contains algorithms and communication variables common to alternates of a recovery block. The algorithms are assertions on input and output variables. The communication variables are those variables which are passed as parameters to other recovery blocks. It also includes values returned from calls to other recovery blocks. The recovery block interface also takes over the responsibility of invoking the next

alternate. The programs that use a recovery block call the interface which then calls an alternate.

When P1 (outer block) fails, its input (nonlocal variables) is restored to their initial state. The states of the communication variables are preserved for the next alternate of P1 to use where result of P2's execution is needed. To determine if the previous (failed) alternate executed the call statements to other blocks a progress variable is included as one of the communication variables. The progress variable should be updated by P2 after passing its acceptance test. By using the recovery block interface the number of trials where P2 succeeds is N1 + N2 instead of N1 * N2. Thus the recovery block interface can reduce the time for executing nested recovery blocks when the inner block succeeds and an enclosing alternate fails.

Software Support for Recovery

The recovery block scheme and its implementation with a recovery cache [32] provides fault tolerance against functional program units. In operating systems where nonfunctional modules are common, restoring the state of nonlocal objects alone may not result in a consistent system state. The internal state of a nonfunctional module must be restored when an operation fails due to a failure of an operation or a revocation of a successful operation due to a failure of a higher level module in a nested call. Language exception handling mechanism cannot provide this type of

recovery because the errors are not anticipated.

To provide error recovery in nonfunctional modules, operations which change the internal state must provide a corresponding undo procedure to reverse the change. The undo procedure should be part of the specification of the normal operation. A possible syntax for such a language construct is

Procedure <name1> (<parameters>) undo <name2>, where name1 is the normal operation procedure and name2 is the undo procedure. The implementation should ensure the automatic invocation of name2 when name1 fails or when its action is being revoked. Another useful language mechanism is a facility to declare recovery data and statements to save and restore such data. A proposed mechanism is the inclusion of a backup data type in a language. A declarative syntax for such a type is

backup <var> { <id-list> }, where var is a variable of type backup and id-list is a nonempty list of variables to be saved. The operations to be permitted on a backup data type are save and restore. The syntax for save and restore statements are

save <var> | save <var>.id

restore <var> | restore <var>.id

where var is a backup data type and id is an element of id-list. When the save or restore statement does not specify any id-list element the entire backup variable is implied.

A possible implementation of the backup data type is to represent a backup variable as a structure (see Figure 6). The size field is the number of bytes required to store all variables in id-list. The length field gives the number of variables in id-list. The base field is the address of the area to save the variables in id-list. The triples field is an array whose dimension is the length of id-list. A triple is represented as (loc, disp, nbyte), where loc is the storage address of a variable in id-list, disp is the displacement of the variable from base (save area), and nbyte is the number of bytes occupied by the variable.

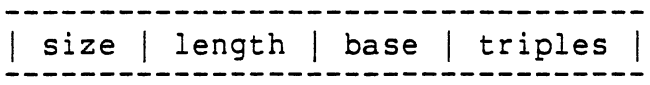


Figure 6. Backup Data Type Representation

The advantage of a language mechanism over the recovery cache [32] is that it can be used in a virtual memory environment. The save address in the recovery cache scheme is real address while it is a virtual address in the software approach. The combination of undo procedures and backup variables can be used to restore the internal state

of a nonfunctional module.

Architecture Assistance for Error Recovery

Architecture assistance for software error recovery is required to speed up and to automate the recovery process so that it is transparent to users. The architecture should provide two types of assistance for effective recovery. The first is for error recovery within a single process or procedure and the second is recovery for interacting processes.

To provide error recovery within a single process or procedure, the processor should include instructions to perform save and restore operations and an instruction to discard a recovery data when the executing context terminates in its instruction set. A system data structure called a recovery context to facilitate automatic error recovery is shown in Figure 7. A recovery context should be created for each procedure.

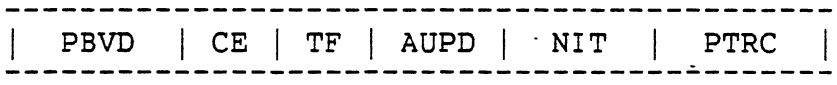


Figure 7. Recovery Context Data Structure

The recovery context data structure contains information to enable the architecture perform a recovery action when an error is detected or to reclaim the storage used to store recovery information when it is no Longer needed. The PBVD field is an address of a backup variable descriptor for any backup data defined in the procedure. The CE field is a flag to indicate if a process terminated with (without) error. The TF field is a flag that indicates if the procedure or process is the root of a recovery chain. If the flag is not set, then the procedure or process is called by some other program unit. The AUPD field is the address of an undo procedure descriptor if one is specified. The NIT field gives the number of different procedures called by a procedure. Lastly the PTRC field is an array of addresses to other recovery contexts. The size of the array is equal to NIT. For nested procedure calls the recovery contexts constitutes a chain of recovery contexts. These contexts indicate the environments to be affected by an error in one of the procedures/processes.

The operating system should implement a recovery process to traverse the recovery contexts and perform the necessary error recovery actions. The recovery context chain defines a recovery space. This space is bounded if there are no implicit interactions with other environments. A bounded recovery space ensures that the system is restored to the state that existed before an erroneos state was entered. To come close to a bounded environment ideal the

operating system design and implementation must meet the following requirements:

1. processes do not interact by means of global memory;

2. called procedures do not access memory areas outside the passed parameters locations; and

3. system decomposition is hierachical. The first requirement prevents uncontrolled interaction which can result in a domino effect (unbounded recovery space). The use of abstract objects implemented as software managers can enforce this requirement. The second requirement ensures that a called procedure modifies a well defined part of a calling procedure's address space. This requirement guarantees that when a restore operation is executed the calling procedure's environment is completely reestablished. The second requirement can be met by using address descriptors to pass parameters. The third requirement defines an interaction path that can be represented by a recovery context chain. A system that is deadlock free is also known to be free from the domino effect [40]. The satisfaction of requirements one and three help to reduce the probability of deadlock considerably. If a system is domino free, then the recovery context chain is a bounded recovery space.

Summary

A general error detection and recovery approach has been presented in this chapter with emphasis on backward error recovery. A mechanism to detect control errors in a nonfunctional module based on decomposing the module into three types of submodules called manager, decision-maker, and operations is presented. The main advantage of this scheme is that the internal state of a nonfunctional module is partitioned into control state and data state which helps to localize error recovery. The use of a recovery block interface is proposed to reduce the cost of backward error recovery in nested procedure calls. Also, a linguistic mechanism for a programmer defined recovery is discussed. A recovery context which defines a recovery space for an executing program is proposed as a means to automate system error recovery.

CHAPTER V

SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

Summary

The study has presented methods to restrict the flow of information in a computer system and its impact on error recovery in an operating system. There are two dimensions of information flow control. The first concerns how to make the access environment of a program as small as possible while the second focuses on the control of interaction among competing and cooperating processes.

On the restriction of access environment a program structure called a manager is proposed. This structure provides two levels of restriction. The first is hiding the representation of an abstract object from the users of the object which is the only form of restriction provided by monitors. The second level is the presentation of a partial view of an object representation to operations. This level limits the damage an operation can cause to the state of a shared data. The manager structure partitions the internal state data of an encapsulated object into control and shared data states manipulated by specific program units. The benefit of a partitioned data state is that error categorization and recovery can be localized.

Further restriction of the access environment of a process can be achieved through the use of a descriptor to represent address variables. The use of descriptors to represent address variables ensures that unrelated memory cells are not mutilated through addressing error. Its use in passing parameters confines a called procedure's actions to a well defined subset of a caller's address space.

The orderly interaction of processes can be achieved through robust resource and process control strategies. The proposed manager structure is both a resource management and a synchronization module. Its structure separates the control algorithm from operations. Its implementation results in a self-checking software because a decision is verified with an assertion before it is carried out. The separation of control algorithm from operations reduces the sources of internal state error of resource and process controllers. A good resource control strategy should avoid implicit interaction by ensuring that resource units make the right transitions from one state to another under proper circumstances.

The recovery from an error depends on knowing what is to be restored, the scope of an error, and the type of failure. The use of a language mechanism to specify recovery data and procedures can solve the problem of knowing what to restore. The recovery context chain is sufficient to determine error scope in a well structured

system without implicit interaction. The recovery block is the most general technique for software fault tolerance. It is adequate for providing fault tolerance in functional modules. Error recovery in functional modules can be achieved by saving the values of nonlocal variables on block entry and then restoring these saved values on failure of the module. The recovery block provides automatic reconfiguration of the modules since an equivalent program unit (alternate) is tried when a module fails. The specification of undo procedures to reverse actions of operations provides the means to apply backward error recovery technique in a nonfunctional module. The recovery block interface proposal can reduce the amount of backward error recovery in nested block calls when an outer block fails and the inner block succeeds.

Architectural support is necessary to implement a general and automatic fault recovery in an operating system. The minimum assistance required is the inclusion of instructions to create recovery context and backup data descriptors, save and restore recoverable data. The operating system should provide an error recovery process to be invoked when an error condition is raised.

Conclusions and Recommendations for Further Work

Error recovery in operating systems can be confined to the immediate environment of a failed process by employing design and implementation techniques that restrict

information flow through the system. The manager construct combined with a descriptor-based parameter passing mechanism can control the flow of information and can avoid implicit interactions. However, for pervasive errors (errors whose damage extends beyond the failed environment), the operating system should employ some elaborate techniques such as system reinitialization and use of audits to examine the system data structures. With good system structure and explicit information sharing mechanism, the frequency of pervasive errors can be drastically reduced. The proposals made in this study involve some space and time overhead but their impact on performance is not known. An area for further study is an implementation of these proposals to ascertain their effectiveness..

BIBLIOGRAPHY

1. Andrews, G. R. "Synchronizing resources", ACM Trans. on Prog. Lang. and Syst., 3-4 (Oct., 1981), 405-430.

2. Ancilotti, P. et al. "Language features for access control", IEEE Trans.on Software Eng., SE-9, 1 (Jan., 1983), 16-24.

3. Anderson, T., Knight, J. C. " A framework for software fault tolerance in Real-time systems", IEEE Trans.on Software Eng., SE-9, 3 (May, 1983), 355-364.

4. Ayache, J. M., et al. "Observer: A concept for on-line detection of control errors in concurrent systems", Fault Tolerant Computing System Symposium - 9, (June, 1979), 76-86.

5. Bartlet, J. F. "A nonstop kernel", ACM Proceeding of the 8th Symposium on Operating Systems Principles, (Dec., 1981), 22-29.

6. Basili, R. V., Perricone, T. B. "Software errors and complexity: an empirical investigation", Communications ACM, 27-1 (Jan., 1984), 42-52.

7. Bishop, J. M., Barron, D. W. "Principles of descriptors", The Computer Journal, 24-3 (1981), 210-220.

8. Black, J. P., et al. "A case study in fault tolerant software", Software-Practice and Experience, 11 (1981), 145-157.

9. Bryant, R. E., Dennis, J. B. "Concurrent programming", Lecture notes in computer science, 143, Springer Verlag, (Oct., 1980), 426-451.

10. Charlton, C. C., Leng, P. H. "Aids for pragmatic error detection", Software - Practice and Experience, 13-1 (Jan., 1983), 59-66.

11. Corsini, P., Frosini, G. "The implementation of abstract objects in a capability based addressing architecture", The Computer Journal, 27-2 (1984), 127-134.

12. Deitel, H. M. An Introduction to Operating Systems, Addison Wesley, (1983).

13. Denning, P. J. "Fault tolerant operating system", Computing Surveys, 8-4 (Dec., 1976), 359-389.

14. Dennis, J. B., Van Horn, E. C. "Programming semantics for multiprogrammed computations", Communications ACM, 9-3 (March, 1966), 143-155.

15. Dijkstra, E. W. "The structure of the THE multiprogramming system", Communications ACM, 11-5 (May, 1968), 341-346.

16. Endres, A. "An analysis of errors and their causes in system programs", IEEE Trans.on Software Eng. SE-1, 2 (Feb., 1975), 140-149.

17. Fabry, R. S. "Dynamic verification of operating system decisions", Communications ACM, 16-11 (Nov., 1973), 659-668.

18. Fabry, R. S. "Capability-based addressing", Communications ACM, 17-7 (July, 1974), 403-412.

19. Gligor D. V. "Architectural implications of abstract type implementation", The 6th Annual Symposium on Computer Architecture Proceedings (April, 1979), 20-30.

20. Glass, R. L. "Persistent software errors", IEEE Trans.on Software Eng. SE-7, 2 (March., 1981), 162-168.

21. Goodenough, J. B. "Exception handling: issues and proposed notation", Communications ACM, 18-12 (Dec., 1975), 683-696.

22. Hansen, R. C., et al. "The 3B20D processor & DMERT operating systems: fault detection and recovery", The Bell System Technical Journal, 62-1 (Jan., 1983), 348-365.

23. Hoare, C. A. R. "Monitors: an operating system structuring concept", Communications ACM, 17-10 (Oct., 1974), 549-557.

24. Hoare, C. A. R. "Communicating sequential processes" Communication ACM, 21-8 (Aug., 1978), 666-677.

25. Jammel, A. J., Stiegler, H. G. "Managers versus monitors", Information Processing, (1977), 827-

830.

26. Jones, A. K. "The narrowing gap between language systems and operating systems", Proc.Information Processing (1977), 869-873.

27. Kahn, K. C., et al. "iMAX: a multiprocessor operating system for an oject-based computer", ACM Proc.of the 8th Symposium on Operating Systems Principles (Dec., 1981), 127-147.

28. Kieburtz, R. B., Silberschatz, A. "Access-right expressions", ACM Trans.on Prog. Languages and Syst. 5-1 (Jan., 1983), 78-95.

29. Kim, K. H. "An implementation of a programmer-transparent scheme for coordinating concurrent processes in recovery", Proceeding COMSAC (1980), 615-621.

30. Kohler, W. H. "A survey of techniques for synchronization and recovery in decentralized computer systems", Computing Surveys, 13-2 (June, 1981), 149-183.

31. Kopetz, H. "Software design for fault tolerance", Proceeding COMSAC (1980), 591-595.

32. Lee, P. A., et al. "A recovery cache for the PDP-11", IEEE Trans.on Computers, C-29, 6 (June, 1980), 546-549.

33. Lee, P. A. "A reconsideration of the recovery block scheme", The Computer Journal, 21-4 (1978), 306-310.

34. Levin, G. M., Gries, D. "A proof technique for communicating sequential processes", Acta Informatica, 15, (1981), 281-302.

35. Linden, T. A. "Operating system structures to support security and reliable software", Computing Surveys 8-4 (Dec., 1976), 409-445.

36. Lister, A. M., Sayer, P. J. "Hierarchical Monitors", Software-Practice and Experience, 7 (1977), 613-623.

37. Myers, G. J., Buckingham, R. S. "A hardware implementation of capability-based addressing", ACM Operating System Review, 14-4 (Oct., 1980), 13-25.

38. Randell, B. "System structure for software fault

tolerance", IEEE Trans.on Software Eng., SE-1, 2 (June, 1975), 220-232.

39. Russel, D. L. "State restoration in systems of communicating processes", IEEE Trans.on Software Eng. SE-6, 2 (March, 1980), 183-194.

40. Russel, D. L. "Multiprocess recovery using conversation", The 9th.Annual Intl. Symposium on Fault-Tolerant Computing (June, 1979), 106-109.

41. Schwarz, P. M., Spector, A. Z. "Synchronizing shared abstract types", ACM Trans.on Computer Syst. 2-3 (Aug., 1984), 223-250.

42. Seifert, M. "Reconfiguration and recovery of multiprocess systems in fault tolerant distributed systems", Proceeding COMSAC (1980), 596-602.

43. Siewiorek, D., et al. "C.mmp: The architecture and implementation of a fault tolerant multiprocessor", The 7th Intl.Symposium on Fault-Tolerant Computing (June, 1977), 37-43.

44. Wilkes, M. V. "Hardware support for memory protection: capability implementations", Proc.ACM Symposium on Architectural support for programming languages and operating system (March, 1982), 108-116.

45. Wulf, W. A., et al. "Hydra: the kernel of a multiprocessor system", Communication ACM. 17-6 (June, 1974), 337-345.

46. Wulf, W. A. "Reliable hardware/software architecture", IEEE Trans.on Software Eng. SE-1, 2 (June, 1975), 233-240.

VITA

Jonathan Asuru

Candidate for the Degree of

Master of Science

Thesis: TEMPORAL LOCALIZATION OF ERROR RECOVERY IN OPERATING SYSTEMS BY RESTRICTING INFORMATION FLOW

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Ogbakiri, Rivers-Nigeria, September 25, 1954, son of Walson and Lavender Asuru.

Education: Graduated from County Grammar School, Ikwerre-Etche, Rivers-Nigeria, in December, 1973; received Bachelor of Science degree in Computer Science from University of Lagos, Nigeria, in June, 1979; completed requirements for the Master of Science degree at Oklahoma State University in December, 1985.

Professional Experience: Teaching Assistant, University of Port Harcourt, Nigeria, November, 1980 to August, 1982; Teaching Assistant, Oklahoma State University, August, 1984 to Present.