# Fault Tolerant Operating Systems*

PETER J. DENNING

*Computer Science Department, Purdue University, West Lafayette, Indiana 47907*

This paper develops four related architectural principles which can guide the construction of error-tolerant operating systems. The fundamental principle, system closure, specifies that no action is permissible unless explicitly authorized. The capability based machine is the most efficient known embodiment of this principle: it allows efficient small access domains, multiple domain processes without a privileged mode of operation, and user and system descriptor information protected by the same mechanism. System closure implies a second principle, resource control, that prevents processes from exchanging information via residual values left in physical resource units. These two principles enable a third, decision verification by failure-independent processes. These principles enable prompt error detection and cost-effective recovery. Implementations of these principles are given for process management, interrupts and traps, store access through capabilities, protected procedure entry, and tagged architecture.

*Keywords and phrases:* capability machine, decision verification, error confinement, error detection, memory access, memory control, multiple domain processes, privilege-state mechanism, process implementation, process isolation, process switching, protected procedure entry, resource control, software error recovery, store access, store control, tagged architecture

*CR Category*: 4.30

## INTRODUCTION

Many modern computer systems seek to provide nearly continuous interactive service for their customers, to protect against accidental or malicious destruction of information in their trust, and to guarantee that confidential information cannot be divulged. These systems are judged as much by their abilities to do these things reliably as by the range of services they offer. Although reliability, in the sense of error tolerance, has long been sought in operating system software, it has always been difficult to achieve. A set of principles of reliable operating systems has begun to emerge.

The full range of approaches to operating systems reliability is not surveyed here. Rather, this paper concentrates on hardware assistance for *error confinement*, without which most other reliability goals are difficult to achieve. With the focus on the capability-based architecture as a particularly attractive form of such hardware assistance, this tutorial concentrates on operating systems for such machines. The capability architecture was originally envisioned as a uniform method of protecting access to data segments and procedures in a computer system [DVH66, Wil75]. More recently, it was advocated also as a uniform method of addressing shared objects [Fab74]. This paper explores the theme that guided the

* Supported in part by NSF Grant GJ-43176.

CONTENTS

INTRODUCTION
Reliability· Fault Tolerance
Error Confinement Principles
1 PROCESSES
Concept
A Single-Processor Implementation
Interrupts and Traps
2. PROCESS ISOLATION
Open versus Closed Environments
Implementing a Capability Environment
Processes of Multiple Domains
Isolating Descriptor Information
Examples and Comparisons
3 RESOURCE CONTROL
The General Principle
Application to Memory Management
Application to Processor Assignment
4. DECISION VERIFICATION
Sender-Receiver Formulation
Application to Memory Access and Control
Application to Encapsulation
5. ERROR RECOVERY
General Concepts
System Error Recovery
User Error Recovery
6 SUMMARY
ACKNOWLEDGMENTS
APPENDIX 1 Procedure Mechanism for a Capability Machine
REFERENCES

designers of the Plessey S/250: the capability architecture is a uniform method of implementing the highest possible degree of error confinement [ENG72].

The concept of a "capability" as a generalized permission to use storage objects (segments) and procedures was introduced in 1966 by Dennis and Van Horn [DVH66]. They suggested that operating system functions could be envisaged as "meta-operations" manipulating a protected data structure comprising a set of "capability lists"; a process could perform a given meta-operation only if a capability for it was recorded in the capability list of that process. The capability concept immediately intrigued implementors. Lampson introduced it into the Berkeley Computer Corporation (BCC) 500 computer [LAM69], but unfortunately that system never went into production. Aspects of the capability concept were used in the Cal 6400 timesharing system at the University of California, Berkeley; however, Lampson and Sturgis report that the full power of the idea could not be implemented on a machine without considerable hardware assistance [LAS76]. Meanwhile, at the University of Chicago, Fabry had a preliminary design of a machine with capability hardware by 1968 [FAB68]; this machine was never completed either, for lack of continued support. At the University of Cambridge, Wilkes and Needham have been constructing their own capability machine (called CAP), which appears likely to reach completion [NEE72, WAL73]. The Hydra operating system, under development at Carnegie-Mellon University, is also based on capabilities [CoJ75, WUL74, WUL75]. The only capability machine in full production is the British Plessey Corporation's System 250 [ENG72], in which many of Fabry's ideas have been incorporated. As of 1976 about 20 of these machines were in use, most in program development for a large military system, the rest under testing as a highly reliable telephone switching computer.

Critics are fond of noting the paucity of capability machines in the open market. The failure of many projects to reach a production stage is easily explained by factors having no connection with the capability concept itself. The common culprit seems to be underestimation of the difficulty of combining many new ideas (not just capabilities) in one design, leading to a hopelessly delayed project. Lampson and Sturgis report that the Cal 6400 project fell victim to this syndrome; and that others have suffered but survived—e.g., Hydra, Multics, OS/360 [LAS76]. The apparent lack of vendor interest in capability machines results in part from the costly trauma experienced by the entry of the computer era into its third generation during the mid and late 1960s: buyers were understandably chary about further innovations, however attractive their principles. Now, in the middle 1970s, we have come to appreciate the serious limitations of our 1960s machines; we are much more sensitive to the issues of security and reliability; we are receptive to proposals for more secure and reliable systems; and, most importantly, we have at hand the technology to construct what once was considered ambitious and expensive hardware. The outlook is optimistic.

## Reliability: Fault Tolerance

Melliar-Smith has suggested some interesting distinctions that clarify the relations among failures, errors, and faults; and between reliability and correctness [MEL75]. A *failure* is an event at which a system violates its specifications. An *error* is an item of information which, when processed by the normal algorithms of the system, will produce a failure. Errors do not always produce failures; they can, for example, be expunged by error recovery algorithms. A *fault* is a mechanical or algorithmic defect which may generate an error. The failure of a component may generate an error which will appear as a fault elsewhere in the system. A considerable time may elapse between a failure and its detection.

Reliability and correctness are not the same. Parnas explains the distinction as follows: a system is correct as soon as it is free of faults and its internal data contains no errors; it is reliable if failures do not seriously impair its satisfactory operation [PAR75]. Reliability means not freedom from errors and faults, but tolerance against them.

Software need not be correct to be reliable. A program module is considered reliable if the most probable errors or faults do not render it unusable, and if those that do are rare and not at times of great need. Similarly, correct software may be unreliable. Correctness proofs often make important implicit assumptions which can be easily invalidated in practice—particularly that

- The underlying machine is correct, i.e., will not fail;
- all local data is consistent and error-free; and
- nothing outside the module can affect its internal behavior except via the interface.

Thus, the correctness proof can demonstrate only that the program in question contains no design faults. Unless the programmer or support system provides redundancy and dynamic checking, errors introduced at run time can invalidate the correctness proof—for example, invalid or inconsistent data can be passed into a procedure, or hardware malfunction can alter instruction code or data. The essential point is that a reliable system must employ many run time mechanisms to keep it as close as possible to a correct state.

The architect of a reliable operating system must provide mechanisms of four types [WUL75]:

- *Error Confinement*: The computing environment is designed so that no procedure has more capabilities than required for its immediate task, no procedure has a large domain of access, and no procedure is allowed to operate on inconsistent data. These properties do not prevent errors, but they will limit the risk that errors do much damage before being detected.
- *Detection and Categorization:* Dynamic verification checks on information, and on the actions of procedures, will detect errors by exposing inconsistencies in data or attempts to violate access constraints. The ideal is precise characterization of the detected error, so that the data can be restored to a consistent state (one from which all subsequent system operations remain within their specifications) and that the fault that produced it can be located. It is essential that the checking algorithms be independent of the system being checked, lest an error prevent its own detection: the checks should be based on the specification, rather than the implementation, of the system.
- *Reconfiguration:* The objective is plac-

ing the system into a consistent state. This may be accomplished by removing from service the failed unit, be it hardware or software; by reconstructing valid copies of data; or by backing up (parts of) the system to a prior, error-free state. These actions need not repair damage, but they will prevent further damage.

- *Restart:* The reconfigured system is restored to service.

Underlying these mechanisms is an *error logging and reporting system* that passes error messages across interfaces and allows system engineers to locate and correct persistent faults [PAR75, RAN75, and WUL75].

Error confinement is the most fundamental of the mechanisms above: full or partial repair cannot realistically be successful without assurance that the damage is localized before repairs are undertaken. Error categorization and fault location are more likely to succeed if the possible extent of damage is small. In principle, early detection gives the means for immediate correction and repair. In practice, however, errors cannot or may not be detected immediately—e.g., a program may not use bad data for a long time after the damage occurred. Error confinement reduces the risk that programs will be applied to bad data, and that external factors can invalidly affect a program's behavior—thus supporting the implicit assumptions underlying most program correctness proofs.

## Error Confinement Principles

This paper focuses on the capability architecture as a "natural" approach to maximal error confinement. The following sections treat four general principles that enhance reliability by confining errors in such systems. Though they do not exhaust all the possibilities, these principles illustrate the techniques of primary interest in the nucleus of an operating system. They include:

- Process isolation;
- Resource control;
- Decision verification; and
- Error recovery.

Process isolation is the principle that each process should have no capability beyond what is required to perform its task. This means that processes cannot interact except along prearranged paths; no unexpected form of interference can occur. It also means that information describing the capabilities of a process must be protected from alteration. Saltzer and Schroeder characterize this as "no-privilege defaults": every action is denied unless explicitly allowed [SAS75].

Resource control implements assignment of physical resource *units* to computational *objects*, processors to processes, for example, or page frames to segments. The primary principle is that when a unit is preempted from an object, the unit's state should be saved and the unit placed in a null state; and when it is reassigned, the unit should be placed in the state it had at the time of last preemption from that object. In short, no unit that contains vestiges of prior use by one object should come under the control of a different object. A secondary principle specifies the ability to verify that commands to assign and release are proper: a unit may be assigned only when free, freed only when assigned.

Decision verification specifies that every decision should be computable in at least two independent ways. A discrepancy indicates an error. In the context of interacting processes, a decision-making process should send a message to a decision-doing process assigned independent resources; before carrying out the decision, the receiver verifies that it is the correct receiver (e.g., by checking a tag field in the message), and that the action requested is consistent with the state of the system. In some applications, a programmer should be able to specify a third process that approves messages between two given processes.

Error recovery seeks to repair damage; if this cannot be done, it seeks to reconfigure the system by removing from service any defective resource units and placing the remaining resource units and processes in consistent states. The principles of process isolation, resource control, and decision verification allow errors to be detected and located before they can spread. Error recovery in such a system can realistically endeavor to correct errors, rather than merely mitigating their effects.

A principal conclusion of this paper is that many operating systems are unreliable because of inadequate hardware assistance for software error detection and confinement. Under the traditional desire for "flexibility," a heavy overhead is associated with imposing the logical structure of the operating system onto amorphous hardware. Consequently, the additional structure for error checking and containment cannot be supported on traditional hardware. Highly reliable operation systems will result only when the nucleus (kernel) is constructed by integrating hardware and software. This means that the abstract machine constituting the nucleus is specified first; only then is the requisite hardware (or firmware) designed, by letting considerations of cost and efficiency determine which portions of the abstract machine will be implemented in the hardware [DIJ68, LIS72, NEU75].

The suggestion that the hardware support essential operating system functions has been made often, but only recently has technological advance made it feasible. [BRH73, HAB76, SHA74]. Since the early 1960s, Burroughs has provided extensive hardware support for its two central operating system structures, the per-process stacks and the segmented virtual memory (see Organick [ORG72]). Though small, the Venus system demonstrated the feasibility of rendering process synchronization and I/O operations as microprograms [LIS72]. Recent interest in virtual machine architecture has stimulated careful reconsideration of the hardware on which operating systems are built [MES70, PK75a, PK75b]. Experimental capability machines are in operation [ENG72, NEE72, WAL73, WUL74]. PRIME stressed integration of hardware and software for high reliability [FAB73]. Although the notion of integrating hardware and software design began from the motivation to simplify operating systems, it is now receiving wide attention as an essential step toward computer systems of high reliability.

## 1. PROCESSES

### Concept

The term "process" is commonly used to denote an abstract entity that demands and releases various resources as it carries out a task. It can be visualized as a program executing on a *virtual processor* (VP), having access to information stored in a *virtual memory* (VM). As suggested in Figure 1, the VP contains four principal components:

- *i*, a unique *index number* identifying the process;
- *d*, a unique identifier of the *domain* of access of the process (in the figure, *d* associates the VP with the VM containing the information of the process);
- *ip*, an *instruction pointer* that specifies the address in the process's domain at which the next instruction to be obeyed can be found; and
- *regs*, a vector containing the values of arithmetic, index, and base registers used by the process.

The vector (*i*, *d*, *ip*, *regs*) is called the *stateword*, or *descriptor*, of the process. The virtual processor of process *i* will be denoted as *VP*(*i*).

A process proceeds in the usual way: its VP alternates between an action that fetches the next instruction (*x* in the figure) and an action that performs its effect on data (such as *y* in the figure). A new value of stateword is defined on completion of an instruction. The dynamic behavior of a process is implicit in the sequence of statewords it generates [BRH73, DEN71, HAB76, HOR73].

### A Single-Processor Implementation

Many systems must implement a large number of processes with a small number of *real processors* (RPs). For simplicity we consider an implementation based on a single RP. The process manager employs a switching mechanism that lets each VP in turn run for an interval on the RP. The term "virtual processor" was invented to

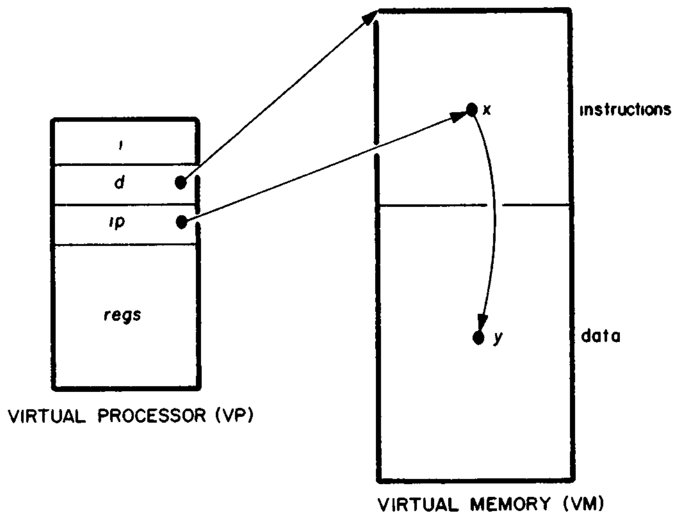

FIGURE 1. Structure of a process. The stateword is embodied in the virtual processor, the instructions and data in the virtual memory. Process *i* is shown with access to domain *d*, about to execute instruction *x* which will affect data *y* and possibly the registers (*regs*).

suggest the simulation inherent in this procedure.

To capture the idea that the RP should be assigned to the most important work first, we extend the stateword to include a *priority number p*:

$$\text{stateword} = (i, d, ip, regs, p).$$

The priority number is usually a nonnegative integer, with lower numbers signifying higher priority. The system observes a *priority scheduling rule*: the running process must always be a highest priority enabled one.

At any given time, a process may be in exactly one of three "execution states": *running,* when its VP controls the RP; *enabled,* when it is awaiting an interval of assignment to the RP; and *blocked,* when it is awaiting some specified signal or event. The process manager for a system with a single RP employs four data structures; the first records the existence of processes, the three others their execution states:

1) a *process list* containing the statewords of all processes in the system;
2) the *virtual process index* (*VPI*) register in the processor, to hold the index of the running process;
3) the *ready queue,* ordering indices of enabled processes according to priority; and
4) a set of *blocked queues,* one for each signal or event on which a process may wait, containing indices of blocked processes.

These structures are stored in the machine's real memory, along with the virtual memories of all the processes. (See Figure 2.) [BRH73, see Chapter 4; NEH75]

A simple implementation of the ready and blocked queues is obtained by linear lists using a link field in each process list entry; the link of entry *i* contains the index of the process following *i* in its queue. (In practice, doubly-linked lists could be used for better reliability [WUL75].) The ready queue is the concatenation of first-in-first-out (FIFO) queues for the increasing priority levels; it can be specified by the auxiliary pointers

$$H,\quad T_0,\quad T_1, \cdots,\quad T_p, \cdots$$

in which $H$ is the index of the head process and $T_p$ that of the tail of the $p$th priority section. To speed up switching the RP to the next ready process, $H$ can be stored in a register of RP. If a process of priority $p$ becomes enabled, it can be inserted in the ready queue following process $T_p$. The

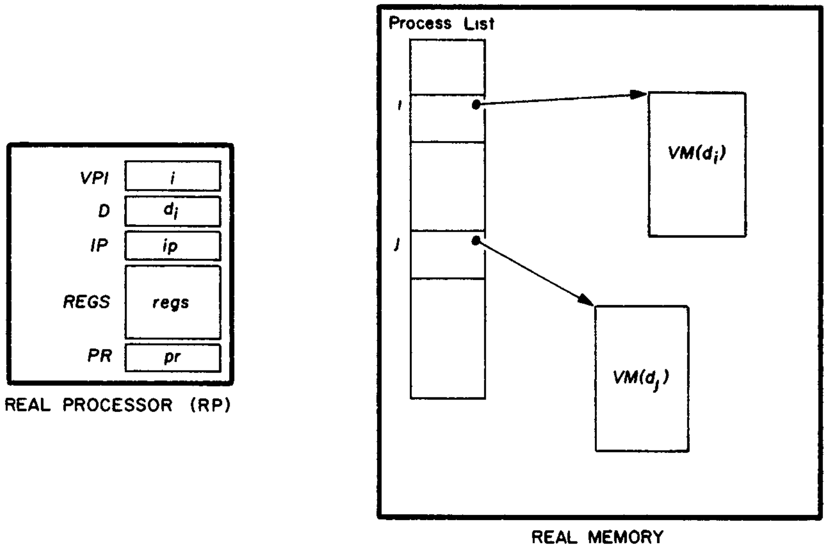

FIGURE 2. Relations among real and virtual objects. The process list contains statewords of all virtual processors. The real memory contains the process list and all virtual memories. The real processor contains the stateword of the running process.

queue of processes for the *k*th signal or event is implemented similarly, though the internal priority ordering is usually not required. Figure 3 illustrates a configuration of such queues, where link value 0 denotes the end of a queue and process 0 is a nullity.

The process manager also contains a set of operations, of three kinds: for switching the RP among VPs, for moving process indices among queues in response to changes in their execution states, and for adding or removing processes or blocked queues in the system. Switching the RP to the first ready process can be implemented by a microprogram of these steps:

1) save the stateword in the process list entry whose index is in the register *VPI*;
2) Load register *VPI* with *H*, the index of the highest priority enabled process, replace *H* with its successor; and then
3) load into the RP the stateword of the process list entry whose index is now in *VPI*.

Hereafter, these steps will be called the SWITCH operation. They are sometimes also referred to as the "context exchange" or "context switch" operation. If the base address of the process list is kept in an RP register, process indices can be interpreted as displacements from this base. Consistent with the priority scheduling rule, any action which enables a process whose priority exceeds that of the running process, must reschedule the running process and generate a SWITCH operation.

The Multics processor has a pair of instructions corresponding to steps 1 and 3 [ORG72]. The IBM 360/370 equipment has instructions for loading and storing program status words (PSWs), which correspond to the (*d*, *ip*) portion of the stateword [GRA75, MAD74]. Neither of these is complete; Multics does not include the scheduling decision. IBM in addition does not save the program registers. In contrast, the Burroughs B6700 has a SWITCH operation that saves a stateword on a process's stack and activates the process of a given next stack [ORG72]. The Control Data 6000 series implements an "exchange jump" instruction that swaps 24 registers of two processes within 5 $\mu$sec [THO70]. The Modular Computer Corporation has a minicomputer (the MODCOMP IV) that stores 16 statewords in a local store to speed

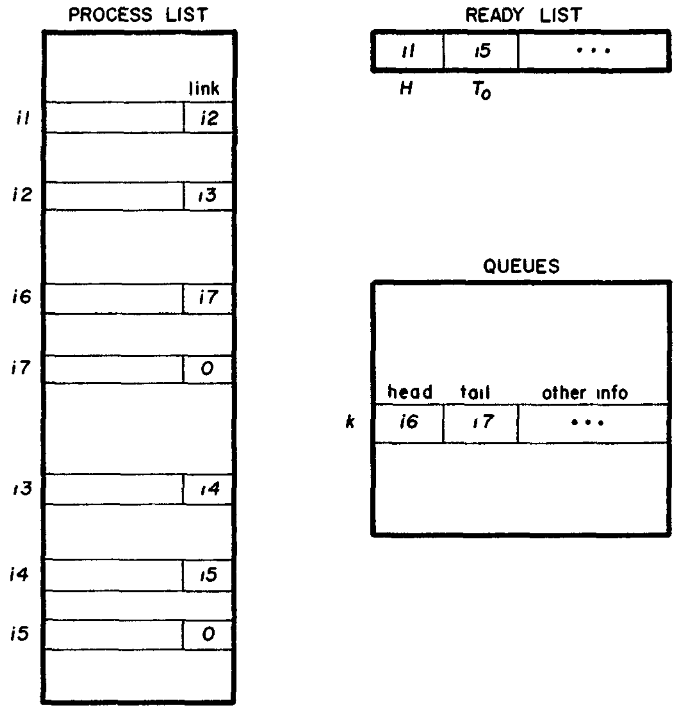

FIGURE 3. A configuration with processes *i*1, ··· , *i*5 on the ready list at priority 0, and *i*6 and *i*7 waiting in the *k*th blocked queue.

up switching among the most active processes.

The second class of process management operations consists of those that cause execution state transitions. These include such well known operations as **send message** (*i*,*m*) and **get message** (*i*,*m*) for sending or receiving a message *m* from the system message queue with index *i*, and **wait**(*s*) and **signal**(*s*) for some counting semaphore *s* [BRH73]. In the latter case, **wait**(*s*) causes *s* to be decremented by 1; if the result is negative, the caller's index (in *VPI*) is attached to the tail of the queue for semaphore *s* and a SWITCH operation generated. The operation **signal**(*s*) causes *s* to be incremented by 1; if the result is not positive, the first index is transferred from the queue of *s* to the ready queue. A SWITCH operation is generated if the signal enables a process of higher priority than the sender. Another state-changing operation is the interval timer, which is used to generate a SWITCH after a time limit—so that no process can monopolize the RP.

The third class of process management operations are used for adding and removing processes, semaphores, or message queues; they include operations like

*i* := **createprocess**(*initial stateword*)
**deleteprocess**(*i*)
*s* := **createsemaphore**(*initial value*)
**deletesemaphore**(*s*)

Their implementation is straightforward manipulation of the process lists or semaphore queues [BRH73, HAB76, NEH75].

## Interrupts and Traps

An executing process may encounter conditions, known as *exceptions* or *faults*, which preclude its further progress until and unless they are corrected. They include conditions checked for in the hardware, such as arithmetic contingencies (overflow, underflow, divide-by-zero, etc.), addressing snags (segment or page faults), invalid accesses to segments, illegal or undefined instructions, or parity check errors in data transmissions

(See Dennis and Van Horn [DVH66].) They include conditions checked for by programming languages, such as array-reference-out-of-bounds or undefined pointer values (see Goodenough [Goo75]). A hardware *trapping mechanism* is usually implemented to stop the process and correct the condition. In simplest form it comprises:

1) *Indicator flipflops.* The *i*th flipflop is set whenever the *i*th condition is detected. It is reset when the condition is responded to.
2) *Trap Microprogram:* At selected points in its instruction cycle the processor examines the indicator flipflops. If at least one is set, it selects one, resets it, and calls a procedure empowered to deal with the corresponding fault condition. A "condition code" indicating the nature of the fault is passed as parameter to this procedure.
3) *Masking:* A mask flipflop, set automatically by the trap microprogram, disables further invocations of the trap microprogram until it is reset. This permits the fault handling procedure to run without interruption. The mask is reset on exit from the fault handling procedure.

It is helpful to imagine that the occurrence of a fault produces an *immediate, unexpected procedure call* on the fault handling procedure [ORG72]. Such calls must obey the normal rules of procedure calls (to be discussed in detail later) which include establishing the proper domain for the fault handler, and reestablishing the original domain of the interrupted process when the fault has been corrected.

Besides fault signals, operating systems must respond to another important class of signals, *device signals.* These are typically the completion signals issued by input/output devices or controllers at the end of a transaction. The purpose of a device signal is to enable a *device driver process* that dispatches the next transaction for the device. To maintain high levels of concurrency, device driver processes typically have very high priority. According to the priority scheduling rule, the enabling of such a process will frequently require the immediate preemption of the currently running process. This preemption is usually called an "interruption" of the current process; for this reason, device signals are sometimes called "interrupt signals". Implementation can employ a microprogram or microprocessor to set a semaphore private to the device driver process, thereby using the ordinary process management mechanism to bring the driver into operation. Many older systems handle interrupts through the trap mechanism: the device signal causes a trap to a procedure which places the current process in the ready queue, enables the driver process, and generates a SWITCH operation.

It is important to remember the distinction between fault and device signals. A fault signal is intended to force a procedure call on a fault handler that operates in the environment of the same process. A device signal is intended to enable a high priority, work dispatching device driver process, and, hence to preempt the (real) processor into a different environment (domain). This distinction is missed or blurred in machines that handle device signals via the trap mechanism. Failure to recognize it in the implementation reduces reliability. (This point will be discussed again in connection with the resource assignment problem.)

## 2. PROCESS ISOLATION

### Open versus Closed Environments

A system is a *closed environment* if the normal state of affairs is that no process or procedure has any capability which has not been explicitly granted—that is, constraints on actions must be removed expressly. A system is an *open environment* if the normal state of affairs is that every process and procedure has every capability which has not been explicitly denied—that is, constraints on actions must be imposed expressly. Implementation can be envisaged in terms of a list associated with each process: a list of capabilities in the closed environment, a list of restrictions in the open environment. A process may perform an action only if permission either is contained

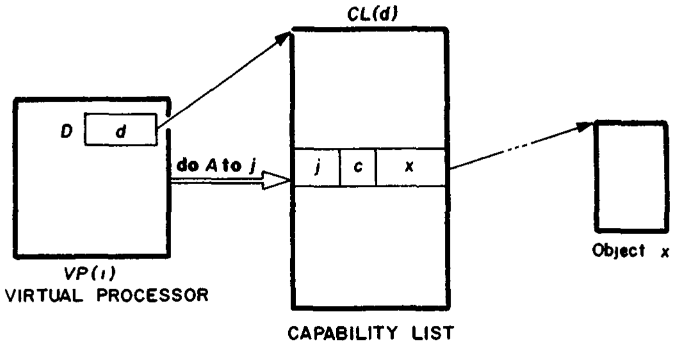

FIGURE 4. An essentially closed environment. The virtual processor specifies action $A$ on object with local address $j$. The system obtains $(c, x)$ from the capability list associated with $VP(i)$. Assuming action $A$ is enabled by access code $c$, the system translates the name $x$ to a reference to an object. The capability list is outside the domain of $VP(i)$.

in some capability, or is not denied by some restriction.

In principle, the two forms of environment are functionally equivalent—the one can simulate the other. In practice, the closed environment is highly error resistant, while the open environment is susceptible both to design errors and to hardware malfunctions. The open environment is susceptible to design errors because a forgotten restriction will enable an invalid action; in contrast, a forgotten capability in a closed environment will prevent a valid action. The open environment is susceptible to malfunctions because a destroyed restriction may enable an invalid action; in contrast, a destroyed capability in a closed environment may disable a valid action. The important point is that in an open environment errors and omissions tend to increase the range of action of a process, whereas in a closed environment they tend to decrease this range. The open enviroment is of little interest when error confinement is important.

The principle of a closed environment is to give each process no more capabilities than it needs to perform its task. The normal state of affairs is completely disjoint, isolated, processes: nothing can be shared or exchanged among processes except by explicit arrangement, all interactions being prohibited unless expressly allowed. No process can attempt to interfere, or communicate, with another in an *unexpected* way. Because a closed environment forces all interactions into the open, it is possible to check them all for consistency and validity as desired.

Figure 4 illustrates the intuitive model of a closed environment. A domain of access is defined by listing all capabilities in a *capability list*; the list for domain $d$, denoted $CL(d)$, is associated with a virtual processor via a domain register ($D$). A capability is represented as a triplet ($j$: $c$, $x$) specifying an association between a *local name* $j$ of the given domain and a unique *system name* $x$ of an object; an *access code* $c$ specifies which actions are enabled for $x$ by any process in domain $d$. The virtual processor invokes an action with a (meta) command "**do** $A$ **to** $j$"; using $j$ as key, a mapping mechanism obtains the pair $(c, x)$ from the capability list, completing the reference to $x$ only if action $A$ is enabled by code $c$.

Although capability lists are a natural (i.e. intuitive) implementation of a closed environment, other implementations are possible. Figure 5 illustrates an important alternate method based on associating an *access list* $AL(x)$ with each object $x$. A virtual processor of domain $d$ may issue a (meta) command "**do** $A$ **to** $x$"; the mapping mechanism completes the reference only if $AL(x)$ contains an entry $(d, c)$ whose access code $c$ enables action $A$. The access list method is functionally equivalent to the capability list method, since the triplet $(d, c, x)$ which grants processes of domain $d$ $c$-access to object $x$, is represented by

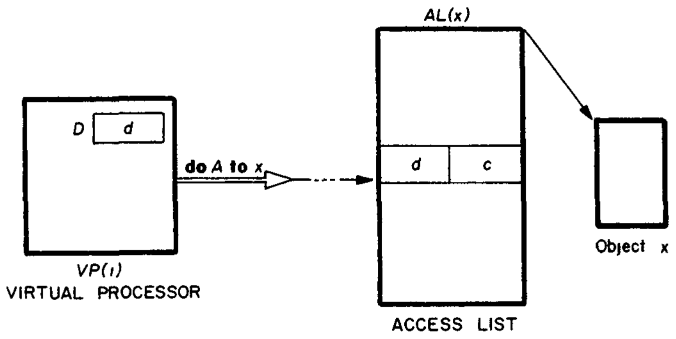

FIGURE 5. Access list implementation of an essentially closed environment. The virtual processor attempts an action $A$ on an arbitrary object $x$. The system translates the name to an access list, permitting action $A$ only if enabled by access code $c$ in an entry corresponding to the domain of $VP(i)$.

storing $(c, x)$ in $CL(d)$ or by storing $(d, c)$ in $AL(x)$ [DEN71, LAM71].

The attraction of the capability list method is its run time efficiency. Suppose each domain has access to an average of $m$ objects, and each object is accessible to an average of $n$ domains. The expected storage requirement for the capabilities of an active process in the capability list method is one capability list, or an average of $m$ locations of fast memory; to guarantee comparable speed in the access list method requires an average of $m$ access lists, or an average of $mn$ locations of fast memory. Put another way, the "working set" of capabilities is stored in one place in the capability list implementation but not in the access list method.

Another distinction between the two forms of implementation is the approach to checking authorization. The access list method associates permissions with objects and checks authorization on each access; the capability list method associates permissions with domains and checks authorizations only once, at the times capabilities are granted. The reduced need for authorization checking contributes to run time efficiency in the capability list method. However, it also means that changes or revocations of permission by an object's owner cannot easily be made to take immediate effect; in an access list method, changes or revocations can take immediate effect and the system cannot become cluttered with capabilities invalidated by such changes. (These points are explored in depth by Saltzer and Schroeder [SAS75].)

In short, the capability list method permits the efficient *use* of capabilities, whereas the access list method permits their efficient *administration*. The most desirable implementation of a practical closed environment should thus employ aspects of both, rather than committing itself exclusively to one. In Multics, for example, capability lists (descriptor segments) are initialized from access lists stored in file directories, and are deleted when processes become inactive [ORG72, ScS72]. The redundancy arising from using both together improves the prospects of recovering from errors that damage capability or access lists.

It is only fair to note that the concept of isolation implied by a closed environment is not so well defined as the above discussion might suggest. Above, two processes are considered as isolated if no action of one can directly affect the other. Lampson [LAM73] has pointed out various ways in which an "isolated" process can transmit information to another process—e.g., by clever use of file interlocks, encoding information into apparently legitimate messages, or by varying its load on resources in encoded patterns. With strong restrictions on a system, information flowing invalidly via variables stored in the system can be blocked [DEN76]; however, information flowing invalidly via resource usage pattern is much more difficult to block [LIP75].

Fortunately, subtle interactions of this type are not important in error confinement.

## Implementing a Capability Environment

The capability mechanism is the most efficient known implementation of a closed environment. Since it is referred to frequently, its implementation is considered here.

Most computer systems have a different access mechanism for each type of resource. Access to segments and pages is controlled by memory addressing hardware; access to procedures and instructions is controlled by the instruction execution mechanism; access to files is controlled by the file system; and so on. Capabilities can be grouped into corresponding classes. They can be placed in separate capability lists according to type; or they can be placed in the same list, distinguished by a type tag. When a process opens a file, for example, the file system can return a capability for the opened file, for example:

$$j := \textbf{openfile}(\textit{filename})$$

where $j$ is the index of a newly created file capability. The process may thereafter invoke particular file operations with such calls as

$$\textbf{readfile}(j),\ \textbf{writefile}(j),\ \text{or}\ \textbf{closefile}(j).$$

The parameter mechanism must verify that the passed capabilities are of the correct types.

The general form of this mechanism amounts to an automatic type-checking mechanism: each argument capability is checked against a "template" associated with the procedure to verify that it is of the type for which the procedure was designed [FEU73]. New types of objects and capabilities for them can be defined if the mechanism is "extensible," such as in the Hydra system [CoJ75]. The CAP system does limited type checking [WAL73].

The two most frequently used types of capabilities, storage and enter capabilities, are often given special treatment. A *storage capability* grants access to a segment (or page) in the virtual memory of a process. Its access code may specify read, write, or execute permissions; the first two permissions apply to memory references during the "execute" portion of a processor instruction cycle, while the third applies to memory read operations during the "fetch" portion. An *enter capability* grants permission to invoke a procedure in a domain differing from that of its caller [DVH66, FAB74, WIL75].

Figure 6 illustrates the access mechanism for storage capabilities. Each domain $d$ has its own capability list $CL(d)$, whose entries $(j\colon c, x)$ specify that local name $j$ (used as a key during address translation) may be translated to any action, on object $x$, which is enabled by access code $c$. The correspondences between system names and the objects themselves is stored in a systemwide central mapping table ($CMT$), whose entries are of two forms:

1) **inform:** $(x\colon 1, b, l)$,
2) **outform:** $(x\colon 0, a)$,

where $x$ is used as a key into $CMT$. The inform entry signifies that segment $x$ occupies $l$ words of main memory beginning at address $b$—thus $(b, l)$ is the familiar base-limit pair [DEN70]. The outform entry signifies that segment $x$ is in the auxiliary memory at address $a$. Suppose the processor requests action $A$ (read, write, instruction fetch) on the $w$th word in segment $j$. The addressing microprogram performs these steps:

```
1) $(c, x) := CL(d)$ entry with key $j$;
2) $(p, b, l) := CMT$ entry with key $x$;
3) if $p = 0$ then MISSING-SEGMENT FAULT;
4) if $A$ not enabled by $c$ then ACCESS TYPE ERROR;
5) if $w \geq l$ then OUT OF BOUNDS ERROR;
6) perform action $A$ at memory address $b + w$.
```

An ordinary segmented virtual memory omits the central mapping table and puts the triplets $(p, b, l)$ in the capability list (which is then called a "descriptor segment"). In contrast, this mechanism allows permanent capability lists, since their entries require no alteration under storage management decisions (see Fabry [FAB74]).

As usual, a local associative memory can

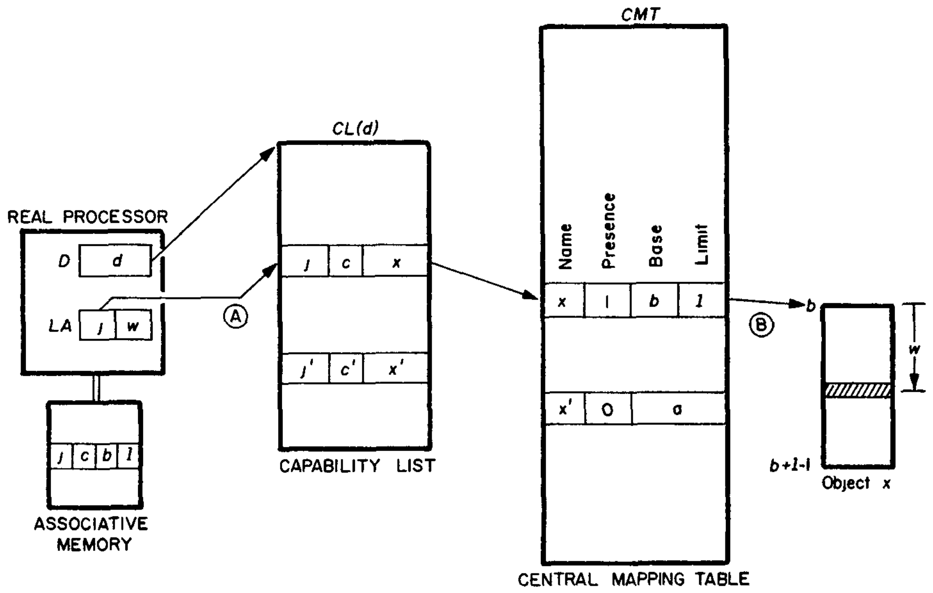

FIGURE 6. Use of storage capabilities. The processor puts addresses of all references in the local address register (*LA*). The local object number (*j*) selects a capability, which is mapped via an inform *CMT* entry to the base of the object; *w* defines a displacement. An attempt to reference local number *j'* would encounter an outform *CMT* entry and generate a missing object fault. Paths like A-B can be stored in a local associative memory to speed up address translation.

be built into the (real) processor to speed up addressing. The associative memory holds a small number of recently used mapping paths (such as A-B in Figure 6). On reference to segment *j*, steps 1–3 can be omitted whenever the associative memory contains an entry with key *j*. (See [DEN70] and [SAS75].)

The implementation of enter capabilities is complicated by the needs to pass parameters, to change the domain register correctly, and to force the instruction pointer to the entry of the new procedure. The central idea of a mechanism is illustrated in Figure 7; the details are given in the Appendix. Suppose procedure *p*1 of domain *d*1 contains a call on procedure *p*2 of domain *d*2. The compiler generates instructions to copy capabilities for the parameters from *CL*(*d*1) to the stack, followed by the special instruction

**enter** *j*2

This instruction causes these actions:

1) The current instruction pointer (*IP*) and domain (*D*) register values are saved in the process's stack;
2) *D* is set to *d*2, the content of the address part of the entry with key *j*2 in *CL*(*d*1); and
3) *IP* is set to (0, 0).

By convention, local name 0 of every domain is reserved for an "entry procedure" whose entry point is at relative location 0; thus step 3 forces the instruction pointer to the entry of the new procedure. When executed by procedure *p*2, a **return** instruction loads *IP* and *D* from the stack, and discards the portion of the stack set up for the previous call (see also [ENG72] and [WIL75].)

## Processes of Multiple Domains

Some procedures in a process require different capabilities from their callers: operating system procedures manipulate private system information inaccessible to users; procedures being debugged have fewer privileges than their creators. Indeed, the ob-

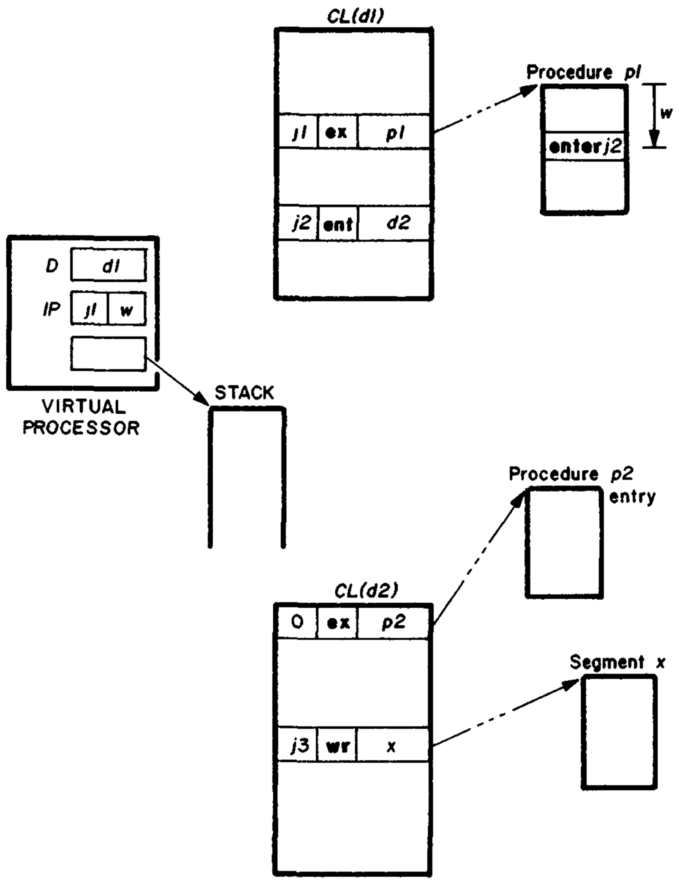

FIGURE 7. Enter capabilities. The current procedure ($p1$) executes the current instruction (at displacement $w$), which causes $D$ to be set to $d2$ and $IP$ to $(0, 0)$. Parameters, and the old $(D, IP)$, are saved on the stack. The codes **ex** and **wr** denote execute and write permissions, while **ent** denotes an enter capability.

jects accessible to a procedure are of at least three distinct types:

- *Local Objects:* those associated with, and private to, the procedure itself;
- *Nonlocal Objects:* those associated with a caller of the procedure in the same process; and
- *Parameters:* those available as arguments of a call.

A computer system meeting the need to separate these types of objects is said to implement *multiple domain processes.* Such implementation must solve four problems:

- *Efficient Representation:* subdomains must be easily represented and accessible when needed.
- *Protected Entry:* on procedure calls, the instruction pointer of the process must be forced to the initial instruction of the called procedure.
- *Domain Changing:* procedure entry (exit) must cause the new (former) local and parameter domains to be established (reestablished).
- *Attenuation of Privilege:* Under normal circumstances it must be impossible for a procedure to perform access to an object passed as a parameter, when that action is not permitted of the caller.

Capability machines solve these problems well. Separate capability lists pointed to by special processor base registers keep track of the domains; the details are given in Appendix 1. Protected entry and domain changing are handled by the enter capability. Attenuation of privilege is automatic, because passing a capability cannot increase the permissions implied by it, and because a "copy flag" can be used to regulate whether a capability can be passed at all [DEN71, LAM71]. (Under special circumstances, a certified procedure may have more permissions for a parameter object than its caller; increasing the privileges denoted by a passed capability is called "amplification" in the Hydra System [CoJ75].)

When the hardware does not permit multiple capability lists to be associated with a process, a *privilege state mechanism* is often used to implement a restricted form of multiple domains. The basic idea is simple [NEE74]. Each object $x$—procedure, instruction, segment, etc.—has associated with it an invariant nonnegative integer $P(x)$, called its *privilege number*. The domain of access of a procedure $u$ comprises all objects $x$ (in its virtual memory) for which $P(u) \leq P(x)$. The privilege number of the currently executing procedure is kept in the instruction pointer register, and can be compared during access to that of an object. Procedure calls and returns must properly update the privilege number field of the instruction pointer.

The classic supervisor/user mode flipflop found in most processors is an example of a privilege state mechanism—certain sensitive instructions can be executed only if this flipflop contains 0. The Multics "ring" mechanism is a more ambitious example extending to segments [ScS72]; by associating three privilege numbers ("ring numbers") with each segment, it defines slightly different domains for reading, writing, executing, and procedure entering. (Priority interrupt mechanisms are not examples; they are simply priority schedulers.)

Special care must be taken to implement attenuation of privilege in a privilege state mechanism. The pointers passed to procedures as parameters are usually addresses in the virtual memory of the process. If procedure $u$ can call procedure $v$ and can pass the address of an object $x$ for which $P(v) \leq P(x) < P(u)$, $u$ may be able to cause $v$ to perform an action on $x$ illegally on its behalf; in other words, simple passing of addresses may amplify, rather than attenuate, privilege. Multics solves this with a rather complex indirection mechanism: the privilege number of a reference is defined as the greatest of the privilege numbers encountered on the indirect address chain.

The defect of a privilege state mechanism is its serious violation of the principle of a closed environment. Since the privilege numbers imply a nesting structure, or inclusion property, on domains of access, it is impossible to implement mutually disjoint domains within the same process. It is therefore impossible to force each procedure to have the minimal capabilities it requires for its task. Errors committed during a procedure of high privilege can be spread widely. This hazard, together with the lack of flexibility implied by strictly nested domains, has motivated Wilkes to argue forcibly for entirely replacing the concept of privilege state with the more general concept of capability lists [WIL75].

## Isolating Descriptor Information

Information defining or limiting the actions of processes is known as *process descriptor information*. Examples include the process list, the various queues of process indices, capability lists, and access lists. Information defining the structure, type, or format of data (used during access to that data) is known as *data descriptor information*. Examples include so-called array dope vectors, file description and index tables, directories, and the capability machine's central mapping table. The correct operation of the system obviously depends on the continuing correctness and consistency of all descriptor information.

The basic principle governing the correctness of authorized changes in descriptor information is: *Descriptor information is treated as private, accessible only to certain certified operations. The authority to invoke these operations is strictly limited.* This principle is easily implemented in the context of Figure 7. Suppose $p2$ is a certified

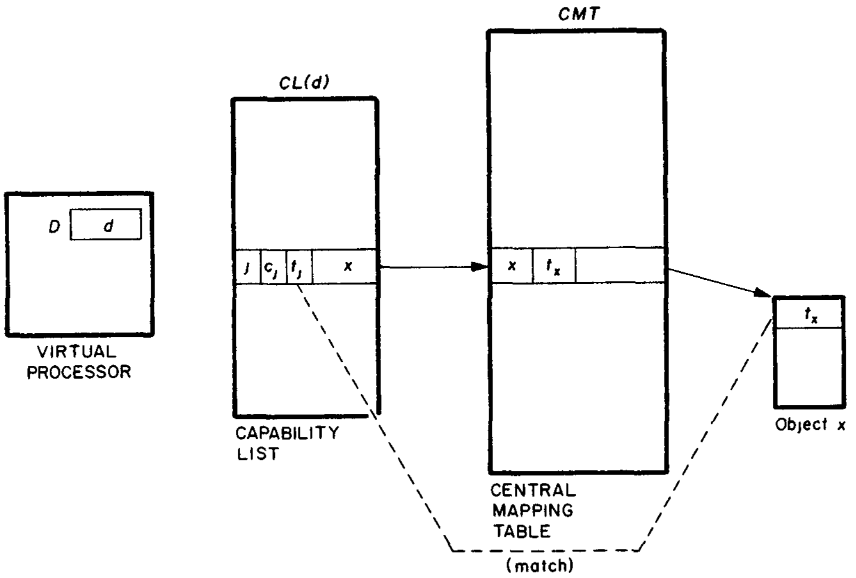

FIGURE 8. Redundancy tags in descriptor information. Each object has a tag ($t_x$) that initially matches a tag in each capability pointing to it ($t_j$). The tags are not mapped, but are compared for a match after the capability is mapped to the object.

procedure, and the segment $x$ to which it has write access contains descriptor information. Prior to invoking $p2$, the caller ($p1$) places parameters describing the requested change on its stack.

Privilege state mechanisms implement this principle only in a limited way, for *all* procedures whose privilege number does not exceed that of a given one will have at least the same access. In other words, data cannot be guaranteed to be accessible strictly by the procedure certified to manipulate it. To make matters worse, many privilege state mechanisms do not couple privilege-number setting tightly with procedure calls and returns, it being possible for random errors or clever programming to cause a presumably unprivileged procedure to come into execution in a highly privileged state.

Capability machines require no privilege state for software procedures [WIL75]. The procedure mechanism automatically associates the proper capability lists with the current procedure. Descriptor information can be made accessible only to the procedure(s) authorized to change it. No privileged instructions for loading or unloading processor registers containing descriptor information are needed because a) most such information is changed during process switching and procedure entry or exit by (certified) microprograms, and b) the design can be constrained so that such registers may receive only exact copies of the (certified) information stored in capability lists. Special capabilities, rather than privileged instructions, can be designed for initiating input/output operations. (See Wilkes [WIL75], Feustel [FEU73].)

The problem of unauthorized changes in descriptor information, as would be caused by hardware or software failures, has received less attention. How can we prevent the use of a capability which, because of some failure, no longer denotes the object to which it originally authorized access? An approach is suggested in Figure 8. Capabilities and objects are extended to contain tags. Associated with each object $x$ is a tag $t_x$, a copy of which can be kept in the central mapping table. When a resource unit is assigned to object $x$, its tag field is set to $t_x$. Similarly, when a capability for $x$ is created, its tag is set to $t_x$. The access mechanism translates a capability ($j$: $c$, $t_j$, $x$) by mapping $x$ to a base address through the central mapping table; no tag bits are mapped, but the hardware compares the source tag ($t_j$) against the resource unit's tag ($t_x$), raising an error condition in case of mismatch. Note that $t_j$ plays the role of a "key" and $t_x$ that of a "lock". This method will detect any single error of these types:

- *Capability List:* One or more bits of $t_j$ or $x$ is changed.
- *Object:* One or more bits of $t_x$ is changed.
- *Mapping:* An error translates $x$ to the wrong object.

Multiple, reinforcing errors may not be caught—e.g., both $t_j$ and $t_x$ could be changed, or all of $t_j$, $t_x$, and $x$, without detection. However, multiple errors are much less likely than single ones [FAB73].

In principle, one can use $t_j = t_x = x$ for the tag values; in practice the number of tag bits can be smaller, determined by the maximum number of objects that can be active at once. If, for example, Figure 8 represents a mapping table for a virtual memory, the tags need distinguish only the segments (or pages) loaded in main memory. If a virtual processor has access only to private objects, a single tag value (stored in a processor register) for the entire process is sufficient; this method is used in PRIME [FAB73]. Few systems today have any such mechanism to serve as a redundancy check against unintentional errors.

Another safeguard is to assign the unique object names sparsely in the space of all possible names. For example, if every two names have Hamming distance at least $k + 1$ (i.e., they differ in least $k + 1$ bits), then any error that changes no more than $k$ bits of a name will render a capability useless. The addressing mechanism will detect the error because there is no central mapping table entry whose key matches the name in the capability presented. This supplements, rather than supplants, the tag mechanism, which relies in part on the fact of tag bits *not* being processed by the address translator.

## Examples and Comparisons

Most practical systems should, and do, employ *both* the access list and capability list approaches. A file directory hierarchy which attaches to each file a list of authorized users, consulted whenever a file is opened, is a common example of an access list approach. Indeed, this approach originated in the file systems of the early 1960s [DAN65, DVH66,WIL75.] An interprocess message facility, in which senders may attempt transmissions to any message queue, is another example, for each receiver process determines which messages to accept by inspecting a message's identity tag and consulting an internal access list [BRH70, DEN71, HAB76, LAM71, ORG72]. A virtual memory mapping mechanism is a common instance of a capability list approach; each mapping table entry behaves as a capability authorizing access to a given segment (or page). The Multics virtual memory illustrates a compromise: when a process attempts initial access to a segment, the file system generates a capability (in the form of a new entry in the process's descriptor segment) only if the access list of the segment so authorizes[1] [ORG72, ScS72]. This compromise allows all virtual memory accesses except the first to proceed efficiently.

Only a few experimental systems implement a fully closed environment. Most existing machines, which use a privilege state mechanism, violate the closure principle in the sense that privileged procedures typically have much wider domains of access than they require for their tasks. There is serious danger of errors committed in highly privileged states being propagated throughout the system.[2] If the virtual memory concept is closed, why is it used side by side with the privileged state concept? The answer appears to be inertia. The priv-

[1] In the notation of Figures 4 and 5, the virtual process in domain $d$ presents a symbolic file name $X$ which locates the file in a directory hierarchy. If the access list $AL(X)$ contains entry $(d, c)$, the capability $(j: c, x)$ is placed in $CL\ (d)$, $j$ being an unused local address and $x$ the unique system name for segment $X$

[2] This hazard reinforces the popular apprehension that security is a myth because there is no protection against highly privileged supervisor procedures. Even as capability systems reduce the risk of irreparable damage by design errors or hardware malfunctions, they reduce the risk of malicious pilferage—e.g., by the procedures that create capabilities or pass them among processes—because they restrict the possible interfaces at which purloining can occur to a few easily monitored ones.

ilege state concept was invented before virtual memory, to solve early protection problems; the virtual memory concept was invented to solve storage overlay problems. Though it is implicit in the early works of Dennis and Van Horn [DVH66] and Fabry [FAB68], Wilkes seems to be the first (1968) to explicitly recognize that the principles of virtual memory could be extended to solve both classes of problems [WIL75]. This realization of the late 1960s was hardly in time to have much influence on the design of machines conceived during the early 1960s, machines which still dominate today.

Another quasi-open architectural principle is the faddish "recursive virtual machine" (see [PoG74, PK75a, PK75b]). A virtual machine is nothing more than a virtual processor and a virtual memory. As discussed in the literature, this approach emphasizes three properties:

- *Strong Isolation.* There is normally no provision for shared instruction code or files, or for otherwise exchanging information efficiently between virtual machines.
- *Recursiveness.* In support of the objectives of developing new systems under an old one, or of continuing an old one without alteration when a new one is installed, a virtual machine should be able to support replicas of itself or of similar machines (the replicas have fewer resources).
- *Exact Replication.* No running program should be able to distinguish a replica from the real machine (except perhaps by observing the speed of its execution). To achieve this, intricate privilege state mechanisms have been designed. Should a virtual machine attempt invocation of any operation which may affect, or be affected by, a system state variable, a trap to a privileged procedure must occur.

Except for the third, these objectives have been present in operating systems design since the early 1960s. Early implementation of all three objectives appeared by 1964 on the MIT PDP-1, and by 1966 on the M44/44X time sharing system at IBM Watson Research Laboratory [ONE67]. The strong current interest seems to derive from IBM's CP-67 project, which has entered production as VM/370 [NES70]. By allowing no sharing to be explicitly arranged, the strong isolation property of itself poses a serious practical limitation. The exact replication and recursiveness requirements can in principle be implemented on the more flexible capability architecture without resorting to the hazardous privilege state, though in fact no one has explored this implementation. The principal attraction of the virtual machine approach seems inertial: the ability to implement the three objectives with least modification of existing hardware and software.

There are at least five important differences between a capability based system (such as CAP, Hydra, or Plessey S/250) and one using virtual memory or privilege-state virtual machines (such as Multics or VM/370).

- Capabilities are permanent; they are system addresses for objects [FAB74]. They can be stored in programs or in the auxiliary memory for an indefinite period. In contrast, entries in standard virtual memory tables are strictly temporary and cannot be placed in the auxiliary memory.[3]
- Capabilities may be passed efficiently as addresses to procedures in different domains, and as components of interprocess messages. In a virtual machine system, a complex indirect addressing mechanism may be required to pass addresses safely among procedures of different domains in the same process, and information can be passed between virtual memories only by copying it.
- The capability hardware handles capabilities for nonstorage objects (e.g., processes, message queues, files) with comparable efficiency as those for storage objects (e.g., segments, pages).

[3] As noted in Subsection "Implementing a Capability Environment," the form of a capability may depend on where the object it denotes is stored—e.g., informs and outforms. Virtual memory table entries are of inform only, making them inherently too transient for holding in auxiliary memory.

- The capability hardware can handle small objects in very small domains; most existing virtual memory systems cannot deal efficiently with small numbers of small segments in a single process.[4]
- The capability mechanism is a natural basis for a closed environment. The privilege-state mechanism is inflexible and not closed: it does not permit disjoint domains in one process, and it exposes all objects to erroneous actions of highly privileged procedures.

That capabilities are permanent, confer authorization of access, and can be grouped arbitrarily to define access domains, are the primary reasons for the advantages described above. Yet the very permanence of capabilities is a liability: a system may bebecome cluttered with unusable or invalid capabilities which cannot easily be located and expunged [SaS75, WiL75].

## 3. RESOURCE CONTROL

### The General Principle

The following discussion concerns the reliable control of resource units subject to exclusive assignment to objects—i.e., reusable resources [HoL72]. Although we think normally of just two possible valid allocation states for such a unit—free and assigned—many allocation strategies, notably those controlling memory, permit a third. The third state arises from splitting the assigned state in two, according to whether the unit is accessible to a process or not. A memory page frame is an example. It makes transition 1 of Figure 9 when extracted from the pool and assigned to some segment; it makes transition 2 when the paging algorithm detaches it from the segment and enters it in the page-out queue; and it makes (solid) transition 3 when its contents have been cleared. Transition 4 is possible if a page fault occurs while the page is still in the page-out queue. For some resource types, e.g., processors, only transitions 1 and (dotted) 3 are possible.

[4] Needham tells me of an interesting side benefit. Efficient small domains have been a significant debugging aid in the CAP operating system. Most programming errors quickly violate access rules, causing the program to stop before the error has done much damage and aiding the rapid diagnosis of the cause.

Reliable resource control is achieved when the units of a given resource type are forced to conform to a well-defined state transition diagram, e.g., Figure 9. Let us imagine implementing a tag field in each unit of resource: tag 0 means that unit is free and its internal state null, and tag $t_x$ means it is allocated to object $x$. (This scheme agrees with that discussed in connection with Figure 8.) The following rules must be observed:

1) If a process, in exercising its $j$th capability with tag $t_j$, makes access to a unit with tag $t_x$, then $t_j = t_x$ must hold. This verifies that a unit remains assigned to the object with which it is presumed associated.
2) A command to allocate a unit to object $x$ is valid only if the unit's tag is 0. The effect of such command is to change the tag to $t_x$, and to restore the unit's internal state to a former nonnull value, if such value is defined by an initial condition or by a prior preemption of a unit from object $x$.
3) A command to preempt a unit of object $x$ is valid only if the unit's tag is $t_x$. The effect of such command is to change the tag to 0, save the unit's internal state in some protected place, and then "clear" the unit by setting its internal state to some null value. (A command to release units omits the state-saving step).

The implementation of Rule 1 was discussed earlier as part of a method for detecting in-

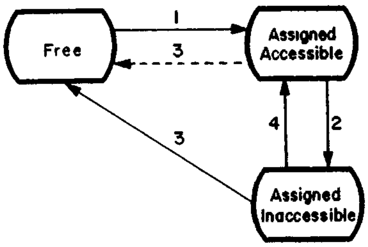

FIGURE 9. Allocation states of a reusable resource unit. Some memory allocation strategies make objects inaccessible to a process without setting the state of the object's unit(s) to a null value.

valid uses of capabilities; the same method is part of reliable resource control. Rule 2 implements transition 3 of Figure 9 and guarantees that no free unit contains information about prior uses. Rule 3 implements transition 1 of Figure 9 and guarantees that each unit upon assignment is in an internal state depending at most on prior uses of the object to which it is assigned. Rules 2 and 3 implement together the additional requirement that reusable units can be assigned to at most one object at a time. These rules permit transitions 2 and 4 if they exist. Hardware that observes and sets unit tags in accordance with these rules is easy to design (see Fabry [FAB73]).

These rules are needed for process isolation, which requires that there be no inadvertent, undetected exchanges of information between processes through residual values left behind in the internal states of resource units. How simple this principle is, yet how seldom it is observed!

## Application to Memory Management

Units of memory resource include pages (or other blocks) of main memory, and tracks or records of auxiliary memory. Though phrased primarily for main memory page frames and disk or drum tracks, the following ideas apply to all units of memory allocation.

Consider page frames. The general principle requires that the preemption of a page frame from a segment be accompanied by saving that frame's content; it is saved, usually on the drum. As long as the frame's content is nonnull, the frame must be treated as *still assigned to the segment*, even though the preemption may have made it inaccessible by setting a presence bit to 0 in some mapping table. How many systems actually do this? Few. Some treat preempted page frames as free even before their content is saved on the drum, and many do not even clear page frames after placing them in the free pool. Not only can private information be divulged by this violation of principle, but a process can acquire what it anticipated as an "empty" page, only to be plunged into error because the page contained nonnull information. (Fabry discusses this point at some length [FAB73].)

Many paging mechanisms are designed to clear page frames, but only just before reassignment—to be prepared for a possible fault seeking the page contained in that frame. This is acceptable only if the page is treated as continuously assigned to the original segment during the entire interval from its detachment from the segment until its clearing. It is unacceptable if the page ever enters the free pool without being cleared, for then the danger exists of an undetectable protection violation. Today's memory technology permits additional page-clearing circuitry at insignificant cost: setting a tag field to 0 (Rule 2) can trigger a reset line in a semiconductor memory; or the regeneration cycle of a core memory can be disabled during a read-out operation by a data channel. To my knowledge, only the PRIME machine proposed this [FAB73]. When the need for the simple hardware assist was not anticipated, the only mechanism of clearing is the zero-storing iteration, a method whose expense has deterred the most prudent operating systems designers from observing the basic principle.

These ideas carry over to the control of secondary memory units such as drum tracks. The drum hardware can be designed to check tags during access (Rule 1), in order to reject track assignment requests for tracks whose tag is nonzero, and to overwrite 0's onto a track when a command is received to reset a tag to 0.

## Application to Processor Assignment

The process switching mechanism, which assigns the real processor to virtual processors, must conform to these rules (see in Section 1, the subsection A Single-Processor Implementation). The process index and domain numbers stored in processor registers serve as tags to identify the virtual processor to which the real processor is assigned. The SWITCH operation completely saves a processor state and completely replaces it with a new one.

Many systems do in fact perform process switching operations correctly—*except for* *switches associated with device signals* (interrupts). If the hardware design did not anticipate the need for an efficient operation of saving and restoring statewords, a large amount of time would be required to execute a procedure that saves the stateword-containing registers one at a time. Because a next device transaction must be dispatched quickly—e.g., before a rotating drum progresses too far from its current position—such machines cannot afford the delay of a complete switch to the device overseer process. Instead, they do no more than save the instruction pointer value and transfer to the entry point of a device driver procedure. The flaw is obvious: *the device driver procedure has come into execution in the environment of whatever process was interrupted*; it has the responsibility of saving whatever registers it uses, restoring them before returning control to the interrupted process. The opportunities for undetected protection errors, for interference with another process's correct execution, or for propagation of errors originating in the interrupt routine—are too numerous to mention here. Given the frequency of interrupts in most systems, and the critical nature of many interrupt tasks, this can be regarded as a major hole in many contemporary operating systems, through which untold errors have doubtless flowed.

To sum up: the general principle of resource assignment is not observed carefully with respect to process switching in many contemporary systems, especially in switching to device driver processes. The solution is straightforward, but requires a carefully engineered SWITCH operation as part of processor microcode, an operation that saves the current stateword, determines the next process to run and loads its stateword—all indivisibly. The vast simplifications which would result throughout the operating system, and the tremendous reduction in exposure to error and error propagation, easily justify the cost of a SWITCH operation.

## 4. DECISION VERIFICATION

Most system actions can be regarded as consequents of decisions taken by its processes. Even if previously certified, decision-taking procedures may produce incorrect decisions because malfunctions alter their instruction code or data. *Decision verification* refers to a class of redundancy checks on decisions; these checks verify that the action decided upon is authorized for its initiator, is applicable to the particular object for which it is intended, or is consistent with the system state. These checks constitute an important means of error detection.

### Sender-Receiver Formulation

The principle of decision verification can be examined in the context of senders and receivers exchanging messages. Let us imagine that process $u$ signifies its decision to perform action $a$ on object $v$ by transmitting the message $(a, v)$ to the controller (access mechanism) of $v$. The system tags the sent message with the identification of the sender, delivering the triplet $(u, a, v)$ to the controller. The most important check performed by the controller is the

1) *Consistency Check.* Verify that action $a$ is consistent with the state of $v$ and is allowable in the state of the system.

Three other possible checks are

2) *Source Check.* Verify that $u$ is permitted access to $v$ with action $a$. (This is already done in capability systems.)
3) *Destination Check.* Verify that $v$ is an object under the control of the controller receiving the decision.
4) *Transmission Check.* Verify that the received, encoded representation of $(u, a, v)$ is valid. (This can be done by including a checksum in the message and verifying that the checksum of received $(u, a, v)$ is the same as the received checksum.)

If the decision passed all the checks, the controller would perform the requested action. If it failed, the controller would have to send an error report back to the decision sender. Unless the decision sender and decision receiver are very carefully designed, it may be quite difficult for the sender to back up and retry the decision [RuB75, Ran75].

The purpose of this mechanism is locating errors by spotting inconsistencies. This implies that, ideally, a decision's generation is *independent* of its verification—the sender and receiver have different algorithms, data, and hardware. Only the message channel is shared.

Under these assumptions, this mechanism will be capable of *detecting single errors* —i.e., an incorrect decision, an erroneous transmission, a malfunction in the accessing mechanism or objects under its control. Multiple errors may not be detected if they "reinforce" (cancel each other)—e.g., sender $u$ determines action $a$ on object $v$, but the decision $(u', a', v')$ reaches the controller, where $a'$ is a valid action for $u'$ on object $v'$. If an error is corrected rapidly after its detection, the problem of multiple errors existing simultaneously will be extremely small. With properly designed coding schemes, the probability of reinforcement, given that a multiple error exists, can be made small. In other words, the simple idea of detecting single errors is both powerful and useful.

If the sender and receiver of a decision are not totally independent, single-error-detection capability may be reduced. For example, if the sender and receiver processes are executed (at different times) on the same processor, a recurrent processor malfunction may go undetected. Thus the collection of processes on the RC4000 (see Brinch Hansen [BRH70, BRH73]; also Lampson [LAM71]), while implementing a message facility of this type, is not fully capable of implementing this error detection principle. Two important cases in which the fully capability of this method can be realized are exemplified in the PRIME system [FAB73]. In one case, all operating system decisions are taken by a control processor and implemented by a distinct process that never runs on the control processor. In the other case, additional circuitry is placed in the memory hardware to verify all decisions made by any processor on memory objects.

### Application to Memory Access and Control •

The mechanisms outlined earlier for resource assignment and release (in Section 3, Application to Memory Management) and tags on capabilities and objects (in Section 2, Isolating Descriptor Information) illustrate the concepts of decision verification for memory management. To review, each unit of memory (a page frame of main memory, a track of auxiliary memory), has a tag. If the unit's tag is zero, the unit is free and its internal state null. If the unit's tag is $t_x$, the unit is assigned to object $x$ and its internal state may be nonnull. A decision to assign a unit to object $x$ is consistent only if the unit's tag is zero; the assignment, implemented in the memory hardware, sets the tag to $t_x$. A decision to release a unit assigned to object $x$ is consistent only if the unit's tag is $t_x$; the release, implemented in the memory hardware, sets the tag to zero and clears the unit's internal state.

For main memory page frames, assignment and release actions can be coupled, respectively, with those that load and save pages, leading to additional confidence that the resource control principle is correctly implemented. For example, a page fault prompts a decision that allocates some frame $v$ to object $x$; the data channel will accept the request to copy the required page of $x$ from auxiliary memory track $u$ only if $\text{tag}(u) = t_x$ and $\text{tag}(v) = 0$, whereupon it copies the entire content of track $u$ (tag included) to frame $v$. A page replacement prompts a similar series of checks and actions.

By storing copies of tags in capabilities, the mechanism can also verify all access decisions of processes. When a process attempts access to a page of its $j$th segment, the memory hardware permits access only if tag $t_j$ of the $j$th capability matches that of the page frame whose address is generated by the mapping mechanism. Since the source tag $(t_j)$ and the destination tag (of the referenced frame) are not handled by the mapping mechanism, this check will detect single errors in the source tag, the destination tag, or the mapping operation.

A full discussion of these ideas in the context of a particular system, PRIME, has been given by Fabry [FAB73].

### Application to Encapsulation

For the purpose of this discussion, a process is *encapsulated* if its communication outside its environment must abide by stricter rules than can be expressed directly in the system. In a capability system, for example, the normal rules allow the control, via enter capabilities, of access to procedures; but they do not allow constraints to be placed on the values passed between a process and such procedures. A mechanism for "message approval" is useful in this case: the initiator of a given process may declare that any interactions between that process and the system be approved by another process. Fabry discusses a message approval mechanism for PRIME [FAB73]; Gaines discusses another such mechanism for the IDA system on the CDC 6000 computer [GAI74, GAI75].

Specifically, suppose process $p$ is declared as encapsulated, and process $q$ is to approve any information input to, or output from, $p$. Whenever $p$ attempts invocation of some procedure, say $x$, that may communicate outside of $p$, the system should send a copy of the parameters to $q$ for approval. If $q$ approves, the procedure $x$ is activated as usual. If $q$ disapproves, the system returns to $p$ without executing $x$ (i.e., $x$ appears as a no-operation to $p$). It is important that $q$ approves a *copy* of the parameters of $x$, otherwise an error in $q$ could be passed to $x$.

Implementation of these ideas would require a bit in a process stateword to indicate whether or not the process is encapsulated. Associated with an encapsulated process must be the index of the approver process. In a capability machine, this would correspond to trapping attempted uses of enter capabilities. In a message-based system, such as RC4000 [BRH73], it would involve trapping attempted uses of send or get message operations. In a conventional system, it would involve trapping all supervisor calls.

## 5. ERROR RECOVERY

The mechanisms of the prior sections emphasize error confinement and rapid detection, the basis for cost effective error recovery. Each has been implemented somewhere. The mechanisms to be described below concern recovery itself, a subject about which we have much less experience. The techniques are under active discussion. There is considerable agreement that they will be useful when implemented. Nonetheless, it is well to keep in mind that the following ideas are much more speculative than the preceding.

### General Concepts

The prior sections have emphasized error confinement and rapid detection. The penalties for each error propagation and sluggish detection show up as costly, incomplete error recovery procedures that at best mitigate rather then eliminate the effect of errors. A not uncommon extreme case illustrates. Suppose that an undetected error changes a page table entry, giving some process write access to the region of memory containing page tables; after a while the mapping information becomes too inconsistent for the virtual memory mechanism to translate addresses, and the system comes to a stop. It may sometimes be possible to recover by purging the active processes from the system; should this fail to put the system into a consistent state, a full restart and initialization may be required. If the error was caused by a program bug, the state of the memory when the system stopped may be so jumbled to make locating the source of the error impossible. In short, the high cost of error recovery in an unconstrained (open) environment is a strong motivation for undertaking the cost of the hardware needed to support a highly constrained (closed) environment.

The steps in error recovery are often listed as detection, location, and repair or reconfiguration. Detection refers to discovering that an error exists; location to identifying it; and repair or reconfiguration to either fixing the error or returning the system to a consistent state and restarting it. A great deal of literature on recovery from hardware failure exists, an excellent overview of recent developments being given in 1974 by Infotech [INF74]. Commonly used, well known detection methods in hardware include: parallel operation of duplicated circuits; error detecting and correcting codes for storing and transmitting information; error detecting circuits; and time-out mechanisms, which signal an error if a unit has acknowledged nothing within a specified interval. Fault location methods for hardware include: detection mechanisms reporting the nature of the error; diagnostic programs run to locate a detected error of unknown nature; and automatic retry mechanisms to prevent full system stops from transient errors. Hardware reconfiguration methods include circuits that switch an offending unit off line when an error is detected.

Recovery from software error is less well understood. The literature contains various guidelines (see [WIL75], [WUL75]), each based on some form of redundancy. The ideal of redundancy is at least two copies of each piece of information in the system, each copy being stored separately so that a single error will leave the other intact. It is not necessary that both versions be in the same form; the original can be structured for efficient use, the copy for compact storage. Unfortunately, many of the guidelines for software error recovery are vague ("prepare for the most probable errors," "minimize references to critical data," "keep a last valid copy of data," or "get it right the first time"), while others are likely to be used anyway by system programmers conscious of the possibility of errors ("use self identifying data structures," "use doubly linked lists," or "use checksums for detection"). Some guidelines are in fact recommendations for elaborate backup subsystems which mitigate against probable errors. One example is a *file archiving system* [WIL75]. When a user logs off a time sharing system, all files which he has modified since log-on are copied to tape. The tapes are processed periodically to retain only the most recent copies of files. In case of failure, a user may request the retrieval of the recent copy from the archive tapes. Another example is *protection against abnormal disconnections*. If the user is for any reason disconnected from the system without issuing a log-out command, an image of his process is saved in a file, ready for resumption when next he logs on.

Another recovery mechanism often encountered is *checkpoint/restart*. Its principle to make a record of the system state at intervals; should an error be detected, the system can be restored to the most recent state of record and restarted. Even if the checkpoint intervals are chosen optimally [YOU74], the overhead of classical checkpointing methods is high. These methods are not used except in special purpose operating systems.

Randell has recently introduced an intriguing new approach to checkpoint/restart, which significantly reduces the overhead of checkpoint by saving values of variables only at the moments they are modified [RAN75]. To enable backing up to be well defined, Randell proposes a programming language construction called a *recovery block*. Each recovery block comprises an "acceptance test" (a predicate) and a collection of alternative procedures ("spares") for accomplishing the same task. On entry to a recovery block, the first alternative is tried; if it succeeds (passes the acceptance test), a normal block exit follows. If it fails, all variables are restored to their values on entry to the recovery block; then the second alternative is tried; and so on. A hardware mechanism called a "recovery cache" is used to maintain chains of values of variables; in case backup to the start of a recovery block is needed, the recovery cache yields the proper former values. Parallel processes must be programmed to cooperate with respect to recovery points, since backing up past an interaction with another process requires "undoing" that interaction by backing up the other process. (That this can be nontrivial is illustrated by Russell and Bredt in their discussion of backing up a producer and consumer process after an erroneous message has been discovered [RUB75].) Randell's proposed restriction for this case is called a *conversation*, and is represented by parallel processes within a given recovery block. All processes of a conversation must pass their individual acceptance tests before any can disengage the conversation. Relevant to a given conversation, any process that enters a (sub)recovery block is incommunicado for the duration of that block.

## System Error Recovery

The methods of this paper are designed to detect errors in system software, especially in data structures. Once an error is found, the object is to recover either by repairing it, or at least by reconstructing a consistent system state so that the damage cannot be spread further.

The principles of system error recovery are most easily illustrated in the context of a level structured operating system. This principle was first used by Dijkstra in the system described in [DIJ68]; it was implemented with the help of microprograms by Liskov on the Venus operating system [LIS72]; and it is recently being exploited by Neumann and his colleagues in a provably secure operating system at Stanford Research Institute (SRI) [NEU75]. The idea is to define the nucleus of an operating system in terms of a hierarchy of nested *abstract machines*, each of which manages privately a specific set of resources not handled on any machine contained within it. If we denote the machines by $M_0$, $M_1$, $\cdots$, $M_K$, then the new operations defined on $M_i$ are implemented as procedures which operate on the local data structure of $M_i$ and may employ operations defined in $M_{i-1}$; these procedures and data structure are known as *level i*. We usually interpret $M_0$ as the basic hardware and $M_K$ as the environment in which users run their programs. The number of levels ($K$) depends on the system and its objectives; Dijkstra used five, Liskov four, and Neumann thirteen. A good discussion of the overall philosophy appears in Habermann et al. [HFC76].

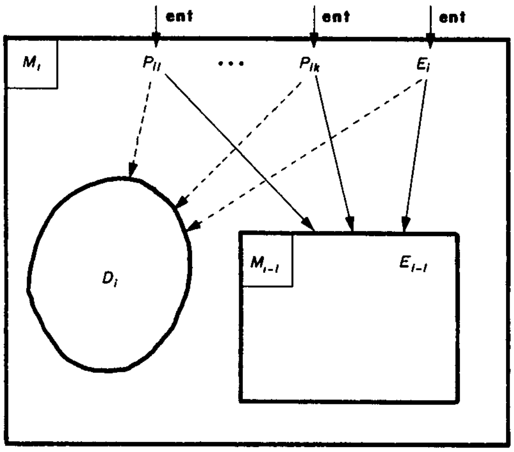

FIGURE 10. Abstract machine view of $i$th level of an operating system Procedures $P_{i1}$, $\cdots$, $P_{ik}$ manage the resources and local data structure ($D_i$) of level $i$, and may invoke operations of $M_{i-1}$. $E_i$ is an error recovery procedure.

Figure 10 illustrates the $i$th level. Procedures $P_{i1}$, $\cdots$, $P_{iK}$ of $M_i$ implement new system operations not available on $M_{i-1}$. They manipulate resources and data represented as a structure $D_i$, and they may invoke operations defined in the lower level $M_{i-1}$. The figure shows access to $P_{i1}$, $\cdots$, $P_{iK}$ controlled by enter capabilities. Execution of an operation of $M_i$ is thus indivisible in the environment of its caller, and the data $D_i$ can be made inaccessible outside of level $i$.

An important component of error recovery is a method or *error reporting* among procedures [PAR75, RAN75, WUL75]. As part of its normal output, each procedure should return an "error report" to its caller; an error report can indicate such outcomes as no errors detected, irreparable structural error in the private data of the procedure, inconsistent values found stored in the private data of the procedure, bad parameter passed, or an irreparable error reported at some lower level procedure. These error reports are normal *responses* from system levels. Also needed are error reports as normal *stimuli*. A procedure of one level, having detected an error in its data structure, should be able to inform lower level procedures that bad values may have been passed and that possibly corrections should be made [PAR75]. To this end we can imagine an error recovery function built into each level ($E_i$ in Figure 10). An error can be reported to level $i$ via $E_i$; $E_i$ can deal with this report recursively:

1) If there is a possibility that the reported error could have been propagated to $M_{i-1}$, $E_i$ calls $E_{i-1}$, which will put $M_{i-1}$ in a consistent state; then
2) $E_i$ performs tests on $D_i$ placing it if need be in a consistent state; it reports finally to its caller the degree of success in repairing the error.

In Step 2, $E_i$ may safely use operations of $M_{i-1}$, which is known from Step 1 to be in a consistent state. Until $E_i$ has reported that $M_i$ is consistent, no process should be allowed to execute instructions at level $i$ or higher.

How is error recovery initiated? Clearly an authorized user process could initiate error recovery by calling $E_K$. If a procedure $P$ of $M_i$ detects an inconsistency in the data structure $D_i$, or receives an error report from $M_{i-1}$, it can call $E_i$; $P$ will report an error to its caller only if $E_i$ fails to make repairs. With modest supporting hardware, the error detection methods of earlier sections can also be used to initiate error recovery. We need to include a *level register* $L$ in each virtual processor to indicate the operating system level at which the current procedure is executing; $L$ must be saved and restored as part of the enter and return operations for protected procedures. An error detected when $L = i$ causes a trap to $E_i$.

Portions of these ideas are being used in Hydra [WuL75] and in the SRI system [NEU75]. A full implementation requires the level structure to be reflected in the hardware by means of the level register and associated mechanism. Such implementation is less ambitious than the scheme of Bernstein and Siegel [BES75]; it is more ambitious that the scheme of Habermann et al [HFC76], who were concerned with certification but not error recovery.

### User Error Recovery

Users define hierarchies among their processes or procedures according to "father-son" or "master-slave" relations [BRH70, BRH73, PA72a, PA72b]. Since these relations have different interpretations than the "levels" relation among the seldom changed system operations, error recovery methods beyond those discussed must be available for the users. In particular, lower level system operations are logically independent of higher level ones, while the lower level user procedures are often totally dependent on higher level user procedures.

The user needs of debugging create three additional requirements not present among the system operations: 1) A user must be able to place an undebugged procedure in a highly constrained domain; 2) he must be able to place such procedures under the control of higher level ones, arranging in particular for erroneous actions of lower level procedures to be trapped and reported to the higher levels; and 3) he must be able to stop the execution of a "runaway" lower level procedure. The first need can be met by either a capability or a privilege state mechanism. The second can be met by languages that allow users to express "exceptional conditions" and state how they are to be handled [Goo75]. The third can be met by allowing creator processes to specify time limits for their offspring and to suspend them. These tools can, of course, be used by the operating system designer in developing new components of the operating system.

## 6. SUMMARY

The objective of this tutorial has been to outline some principles of structuring operating systems to make unauthorized actions impossible, to confine errors to the immediate contexts of their occurrences, and to detect and correct damage before it spreads. The basis is an essentially closed architecture within which every process has no more capabilities than needed for its task and can interact with no other process in an unexpected way. This architectural principle interdicts interprocess interference in three ways: processes cannot commit unexpected actions on one another, they cannot unexpctedly alter descriptor information, and they cannot exchange information in residual values left behind in resource unit states. By hastening error detection via such mechanisms as decision verification, this environment renders realistic the goal of complete error recovery.

The closed environment is not intended to isolate processes irrevocably. It is intended merely to force all interactions into the open where they can be verified. It facilitates, rather than impedes, sharing.

Throughout the discussion it has been emphasized how certain principles would overcome the limitations of third generation operating systems:

1) The operating system should be decomposed into autonomous processes.

Both user and system processes should be controlled by a single mechanism. Although many third generation operating systems implement processes for their users, they do not employ the concepts deep within the system itself.

2) The process manager should implement all process synchronization operations efficiently. The process switch should provide complete, automatic swapping of process statewords. The interrupt strategy should be integrated in, so that process switches, rather than traps, result from device signals. Inadequate hardware assistance makes efficient implementation of these ideas on third generation systems impossible; the innermost parts of most systems contain many implementation shortcuts, loopholes through which untold errors have flowed.
3) The concept of the essentially closed environment should pervade all levels of the operating system. Access to both user data and system descriptor information should be controlled by a single mechanism without a privileged state. (The most efficient such mechanism known is the capability based machine.)
4) The concept of resource control requires that the assignment of a resource unit to an object be accompanied by initializing the unit to some expected state. Releases should be accompanied by resetting the unit's state to a null value. Because few systems have adequate hardware assistance for this, processes may interfere via residual values left in these units.
5) Many operating system decisions can be verified, but hardware assistance is required for efficiency. A simple tagging mechanism can provide the basis for verifying not only accesses, but also resource unit assignments and releases. Decision verification is a powerful tool of error detection.
6) Process, or operation, hierarchies are common in both operating systems and user subsystems. The two most common ordering relations are "levels" and "father-son," the former being used for organizing operating system operation, the latter for user processes and procedures. With proper hardware assistance, it is possible to guarantee the indivisibility of operations by level of a hierarchy and to provide a systematic approach to error reporting and recovery.

I have stressed that considerable hardware support is required if the principles of process isolation, resource control, decision verification, and error recovery are to be implemented efficiently—or at all. It is doubtful whether all the places where contemporary systems lack proper hardware assistance have been identified. Overcoming these limitations in future systems will require that both hardware and software are specified in an integrated design project, rather than, as has been the custom, more or less independently. The plummeting costs of hardware allow great optimism.

Though many operating systems today are explained with process concepts, few use these concepts consistently throughout their structure. The reason seems to be, in part, that the classic view of an operating system as a collection of procedures (subroutines) has been slow in giving way to the view of a collection of processes and, in part, that the classic view of nested protection domains implemented with privileged states has been slow in giving way to processes with multiple disjointed domains. These older views tend to focus attention on manipulations of the processor's instruction pointer and privilege-mode register. Thus we find interrupts treated as traps rather than as wakeup signals for high priority processes; we find few efficient facilities for process synchronization or communication at the lowest system levels; we find domain changing loosely coupled with process switching or protected procedure entry. In contrast, the process/capability view tends to force synchronization, domain changing, and sharing actions into the open where they can be checked. This is not to suggest that the hardware mechanisms derived from the two views always differ greatly. It does suggest that the process/capability viewpoint can produce and *has* produced, significant overall improvements in system architecture. Perhaps the only significant difference in the two views is one of abstraction—above the lowest level of the system, the details of instruction-pointer movements and of domain changes are no longer of concern. But that may make all the difference.

This paper does not pretend to give complete or uniform coverage of all structural principles that can be or have been used to maximize error confinement or facilitate error detection; nor are the principles it suggests universally accepted. This is a difficult subject about which to write. If readers have been stimulated to think about old problems in new ways, the paper has achieved its purpose.

## ACKNOWLEDGMENTS

This paper has many virtual coauthors. Roger Needham spent many hours in discussions about operating systems for capability machines, and he provided much information about the experience with the CAP project. Dorothy Denning and Kevin Kahn helped clarify the distinctions between open/closed environments, and capability/access list based authorization checking, thus excising a string of confusions from an earlier draft. A discussion with P. M. Melliar-Smith was of considerable help in clarifying the properties of errors and faults. K. J. Hamer-Hodges of the Plessey Corporation saved me from misrepresenting the status of System 250, and helped clarify my thoughts on interrupt systems. Jack Dennis suggested the phraseology of approaching a computer system as an integrated design project with no prior commitment to a distinction between hardware and operating system software. James Peterson's critical review of a draft led to substantial revisions and clarification of many points in the paper. Jeffrey Buzen, Robert Fabry, Stockton Gaines, Scott Graham, David Redell, Michael Schroeder, and Maurice Wilkes gave freely of their time while I struggled with clarifying some of the ideas presented here. I am indebted to the National Science Foundation for its Grant GJ-43176, which supported some of this work.

## APPENDIX 1—Procedure Mechanism for a Capability Machine

A procedure mechanism provides for activating procedures and passing parameters to them. The mechanism to be discussed is patterned after the one of the CAP machine [WAL73]. The complete current domain of a process is defined by three capability lists, pointed to by three base registers in its virtual processor: a) Base register 0 (*BR*0) points to $CL(d_0)$, which specifies a *stack* used by all procedure calls in the process; a *stack pointer* (*SP*) register defines the base of the current "frame" of the stack. b) Base register 1 (*BR*1) points to $CL(d_1)$, which specifies the *global domain*, i.e., the set of capabilities belonging to every procedure of the process. c) Base register 2 (*BR*2) points to $CL(d_2)$, which specifies the *local domain*, i.e., the set of capabilities in a process that are private to the current procedure. Within a process, all procedure calls and returns modify the stack and SP register, and some modify the local domain base register; but none modifies the global domain register.

A local address consists of three parts $(t, j, w)$, where the tag $t$ selects one of the capability lists. The addressing mechanism translates $(t, j, w)$ to memory address $b + w$, where $b$ is the base address of the segment whose capability is at position $j$ in $CL(d_t)$ when $t$ specifies the global or local domain ($t = 1$ or $2$), or otherwise at position $j + SP$ in $CL(d_0)$. The instruction pointer (*IP*) register contains addresses $(t, j, w)$, but $t$ may not refer to the stack.

An ordinary procedure call invokes, without any change of domain, a procedure whose capability is in the local or global domains. Prior to the call, parameter capabilities are copied onto the stack. The call instruction is of the form "**call** $t, j, k$" where $t = 0$ or $1$; its effect is

1) $(c, x) :=$ entry of $CL(d_t)$ with key $j$; verify execute access in $c$;
2) Save *IP*, *SP*, and *BR*2 (local domain) in the stack;
3) $SP := SP + k$;
4) $IP := (t, j, 0)$.

Step 4 forces the *IP* to the entry point (displacement 0) of the called procedure. An instruction "return" restores *IP*, *SP*, and *BR*2 from the values found in the top stack frame.

When the called procedure operates in a local domain different from that of its caller, a more complex mechanism is required. A special capability, called an *enter capability*, is needed; if $(j: c, x)$ is an enter capability, $x$ is interpreted not as a procedure name but as the name of a new local domain. The 0th entry in the capability list of this new domain is an ordinary procedure capability for the desired procedure. The instruction "**enter** $t, j, k$" is used to invoke the new procedure; its effect is:

1) $(c, x) :=$ entry of $CL(d_t)$ with key $j$; verify enter capability;
2) Save $IP$, $SP$, $BR2$ on the stack;
3) $SP := SP + k$;
4) $BR2 := x$; $IP := (2, 0, 0)$.

Step 4 forces the local domain to be the called domain, and $IP$ to be the entry point of the domain's entry procedure. Note that enter capabilities can be passed on the stack. The same return instruction as before will work.

Dynamic type checking of parameters can be performed when their capabilities are used. Specifically, any reference to $CL(d_0)$ should cause the capability type to be checked against that expected by the procedure.

The mechanism can be generalized to accommodate statically nested local domains, patterned after ideas used on the B6700. The local and global domain registers are replaced by a *display stack* of base registers $BR1, BR2, \cdots$, where $BR1$ is the outermost domain. If at a given time the display contains $d_1, d_2, \cdots, d_n$, the current procedure is at the $n$th level of (static) nesting. A reference to an object in the $k$th enclosing domain uses tag $t = n - k$ $(k \geq 0)$. Each procedure call and return manipulates the display to preserve this property. (See Organick [ORG72].)

In Subsection Error Confinement Principles of the Introduction it is noted that fault signals cause procedure calls on fault handlers. In the present context, fault signals should generate enter operations for fault handlers. The indices of enter capabilities for fault handlers can be reserved, being the same for all processes and part of the global domain; they can be stored in special registers accessible to the trap microprogram. The mask settings in effect at the time of the fault can be saved on the stack along with the $IP$, $SP$, and $BR2$, so that they are automatically restored when the fault procedure executes a return.

## REFERENCES

[BES75] BERNSTEIN, A J.; AND SIEGEL, P., "A computer architecture for level structured operating systems," *IEEE Trans. Computers* **C-24,** 8 (August 1975), 785–793.

[BRH70] BRINCH HANSEN, P. "The nucleus of a multiprogramming system," *Comm. ACM* **13,** 4 (April 1970), 238–241, 250.

[BRH73] BRINCH HANSEN, P. *Operating systems principles*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[CoJ75] COHEN, E.; AND JEFFERSON, D. "Protection in the Hydra operating system," in *Proc. 5th ACM Symp. on Operating System Principles*, 1975, ACM, New York, 1975, pp. 141–160

[DAN65] DALEY, R. C.; AND NEUMANN, P. G. "A general-purpose file system for secondary storage," in *Proc. AFIPS Fall Jt. Computer Conf.*, Vol. 27, Thompson Book Co., Washington D.C., 1967, pp. 213–229.

[DEN76] DENNING, D. E. "A lattice model for secure information flow," *Comm ACM* **19,** 5 (May 1976), 236–242.

[DEN70] DENNING, P. J. "Virtual memory," *Computing Surveys* **2,** 3 (Sept. 1970), 153–189.

[DEN71] DENNING, P. J. "Third generation computer systems," *Computing Surveys* **3,** 4 (Dec. 1971), 175–216.

[DEN74] DENNING, P. J. "Structuring operating systems for reliability," in *Infotech state of the art report 20: computer systems reliability*, 1974, Infotech International Ltd., Maidenhead, Berkshire, England, pp. 481–504.

[DVH66] DENNIS, J. B.; AND VAN HORN, E. C. "Programming semantics for multiprogrammed computations," *Comm. ACM* **9,** 3 (March 1966), 143–155.

[DIJ68] DIJKSTRA, E. W. "The structure of THE multiprogramming system," *Comm ACM* **11,** 5 (May 1968), 341–346.

[ENG72] ENGLAND, D. M. "Architectural features of system 250," in *Infotech state of the art report 14: operating systems*, 1972, Infotech International Ltd., Maidenhead, Berkshire, England, pp. 395–428.

[FAB68] FABRY, R. S. "Preliminary description of a supervisor for a computer organized around capabilities," in *Quarterly progress report*, No. 18, Sect. IIA, Inst. Computer Research, Univ. Chicago, Chicago, Ill., 1968.

[FAB73] FABRY, R. S. "Dynamic verification of operating system decisions," *Comm. ACM* **16,** 11 (Nov. 1973), 659–668.

[FAB74] FABRY, R. S. "Capability based addressing," *Comm. ACM* **17,** 7 (July 1974), 403–412.

[FEU73] FEUSTEL, E. A. "On the advantages of tagged architecture," *IEEE Trans. Computers* **C-22**, (July 1973), 644–656.

[GAI74] GAINES, R. S. "A new boss/slave relation between processes," in *Proc. 8th Princeton Conf. on Information Science and Systems*, 1974.

[GAI75] GAINES, R. S. "Control of processes in operating systems: The boss–slave relation." in *Proc. 5th Symp. on Operating Systems Principles*, 1975, ACM, New York, 1975.

[GOO75] GOODENOUGH, J. B. "Exception handling: issues and a proposed notation," *Comm. ACM* **18**, 12 (Dec. 1975), 683–696.

[GRA75] GRAHAM, R. M. *Principles of system programming*, John Wiley & Sons Inc., New York, 1975.

[HAB76] HABERMANN, A. N. *Introduction to operating systems design*, Science Research Associates Inc., Chicago, Ill., 1976.

[HFC76] HABERMANN, A. N.; FLON, L.; AND COOPRIDER, L. "Modularization and hierarchy in a family of operating systems," *Comm. ACM* **19**, 5 (May 1976), 266–272.

[HOL72] HOLT, R. C. "Some deadlock properties of computer systems," *Computing Surveys* **4**, 3 (Sept. 1972), 179–196.

[HOR73] HORNING, J. J.; AND RANDELL, B. "Process structuring," *Computing Surveys* **5**, 1 (March 1973), 5–30.

[INF74] INFOTECH INTERNATIONAL LTD. *Report 20: computer systems reliability*, C. Bunyan, Ed., 1974 Infotech International Ltd, Maidenhead, Berkshire, England.

[LAM69] LAMPSON, B. W. "Dynamic protection structures, in *Proc. AFIPS 1969 Fall Jt. Computer Conf.*, Vol. 35, AFIPS Press, Montvale, N.J., 1969, pp. 27–38.

[LAM71] LAMPSON, B. W. "Protection," in *Proc. 5th Princeton Conf. on Information Sciences and Systems*, 1971, pp. 437–443. Also in ACM SIGOPS *Operating Systems Review* **8**, 1 (Jan. 1974), 18–24.

[LAM73] LAMPSON, B. W. "A note on the confinement problem," *Comm. ACM* **16**, 10 (Oct. 1973), 613–615.

[LAS76] LAMPSON, B. W.; AND STURGIS, H. "Reflections on an operating system design," *Comm. ACM* **19**, 5 (May 1976), 251–265.

[LIP75] LIPNER, S. E. "A comment on the confinement problem," in *Proc. 5th ACM Symp. on Operating System Principles*, 1975, ACM, New York, 1975, pp. 192–196.

[LIS72] LISKOV, B. H. "The design of the Venus operating system," *Comm. ACM* **15**, 3 (March 1972), 144–149.

[MAD74] MADNICK, S. E.; AND DONOVAN, J. J. *Operating systems*, McGraw-Hill Inc., New York, 1974.

[MEL75] MELLIAR-SMITH, P. M. *A project to investigate data-base reliability*, Report, Computing Lab., Univ. Newcastle-upon-Tyne, England, 1975.

[MES70] MEYER, R. A.; AND SEAWRIGHT, L. H. "A virtual machine time-sharing system," *IBM Systems J.* **9**, 3 (1970), 199–218.

[NEE72] NEEDHAM, R. M. "Protection systems and protection implications," in *Proc. AFIPS 1972 Fall Jt. Computer Conf.*, Vol. 41, AFIPS Press, Montvale, N.J., 1972, pp. 572–578.

[NEE74] NEEDHAM, R. M., quoted by C. Bunyan in *Infotech state of the art report 20: computer systems reliability*, 1974, Infotech International Ltd., Maidenhead, Berkshire, England, pp. 104–105.

[NEH75] NEHMER, J. "Dispatcher primitives for the construction of operating system kernel," *Acta Informatica* **5**, 4 (1975), 237–256.

[NEU75] NEUMANN, P. G.; ROBINSON, L.; LEVITT, K. N.; BOYER, R. S.; AND SAXENA, A. R. "A provably secure operating system," in *Project 2581 final report*, Stanford Research Inst., Menlo Park, Calif., 1975.

[ONE67] O'NEILL, R. W. "Experience using a time-sharing multiprogramming system with dynamic address relocation hardware," in *Proc. AFIPS 1967 Spring Jt. Computer Conf.*, Vol. 30, Thompson Book Co., Washington, D.C., 1967, pp. 611–621.

[ORG72] ORGANICK, E. I. *The multics system: an examination of its structure*, MIT Press, Cambridge, Mass., 1972.

[ORG73] ORGANICK, E. I. *Computer system organization: the B5700/B6700 series*, Academic Press, New York, 1973.

[PA72a] PARNAS, D. L. "A technique for software module specification with examples," *Comm. ACM* **15**, 5 (May 1972), 330–336.

[PA72b] PARNAS, D. L. "On the criteria to be used decomposing systems into modules," *Comm. ACM* **15**, 12 (Dec. 1972), 1053–1058.

[PAR75] PARNAS, D. L. "The influence of software structure on reliability," in *Proc. ACM Internatl. Conf. Reliable Software, SIGPLAN Notices* **10**, 6 (June 1975), 358–362, ACM, New York, 1975.

[POG74] POPEK, G. J., AND GOLDBERG, R. P. "Formal requirements for virtualizable third generation architectures," *Comm. ACM* **17**, 7 (July 1974), 412–421.

[PK75a] POPEK, G. J.; AND KLINE C. S. "A verifiable protection system," in *Proc. ACM Internatl. Conf. Reliable Software*, 1975, ACM, New York, 1975, pp. 294–304.

[PK75b] POPEK, G. J.; AND KLINE, C. S. "The PDP-11 virtual machine architecture: a case study," in *Proc. 5th ACM Symp. on Operating System Principles*, 1975, ACM, New York, 1975, pp. 97–105.

[RAN75] RANDELL, B. "System structure for software fault tolerance" *IEEE Trans on Software Engineering*, **SE-1**, 2 (June 1975), 220–232. [Also in Op. Cit., 437–449.]

[RUB75] RUSSELL, D. L.; AND BREDT, T. H. "Error resynchronization in producer-consumer systems." in *Proc. 5th ACM*

*Symp. on Operating System Principles, ACM SIGOPS Operating Systems Review* **9**, 5 (Nov. 1975), 106–113.

[SaS75] Saltzer, J. H.; and Schroeder, M. D. "The protection of information in computer systems," in *Proc. IEEE* **63**, 9 (Sept. 1975), 1278–1308.

[ScS72] Schroeder, M. D.; and Saltzer, J. H. "A hardware architecture for implementing protection rings," *Comm. ACM* **15**, 3 (March 1972), 157–170.

[Sha74] Shaw, A. *The logical design of operating systems*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

[Tho70] Thornton, J. E. *Design of a computer: the Control Data 6600*, Scott, Foresman, and Co., Glenview, Ill., 1970, pp. 117–120.

[Wal73] Walker, R. "The structure of a well protected computer," PhD Thesis, Computing Laboratory, Cambridge Univ., England, 1973.

[Wil75] Wilkes, M. V. *Time sharing computer systems*, American Elsevier Publ. Co., New York, 1st ed., 1968; 2nd ed., 1972; 3rd ed., 1975.

[Wul74] Wulf, W. A. et al. "Hydra: the kernel of a multiprocessor system," *Comm ACM* **17**, 6 (June 1974), 337–345.

[Wul75] Wulf, W. A. "Reliable hardware/software architecture," *IEEE* **TSE-2** (June 1975), 233–240.

[You74] Young, J. W. "A first order approximation to the optimum checkpoint interval," *Comm. ACM* **17**, 9 (Sept. 1974), 530–531.