# An Overview of the CAL Time-Sharing System

Butler W. Lampson

September 5 1969

Computer Center
University of California
Berkeley

This version was produced by OCR from the version published by the UC Berkeley Computer Center; there may be errors. The figures were scanned from the Infotech report.

Originally entitled On Reliable and Extendable Operating Systems, *Proc. 2nd NATO Conf. on Techniques in Software Engineering*, Rome, 1969. Reprinted in *The Fourth Generation*, Infotech State of the Art Report 1, 1971, pp 421-444.

## Introduction

A considerable amount of bitter experience in the design of operating systems has been accumulated in the last few years, both by the designers of systems which are currently in use and by those who have been forced to use them. As a result, many people have been led to the conclusion that some radical changes must be made, both in the way we think about the functions of operating systems and in the way they are implemented. Of course, these are not unrelated topics, but it is often convenient to organize ideas around the two axes of function and implementation.

This paper is concerned with an effort to create more flexible and more reliable operating systems built around a very powerful and general protection mechanism. The mechanism is introduced at a low level and is then used to construct the rest of the system, which thus derives the same advantages from its existence as do user programs operating with the system. The entire design is based on two central ideas. The first of these is that an operating system should be constructed in layers, each one of which creates a different and hopefully more convenient environment in which the next higher layer can function.[3] In the lower layers a bare machine provided by the hardware manufacturer is converted into a large number of *user machines* which are given access to common resources such as processor time and storage space in a controlled mariner.[6] In the higher layers these user machines are made easy for programs and users at terminals to operate. Thus as we rise through the layers we observe two trends:

a. the consequences of an error become less severe

b. the facilities provided become more elaborate

At the lower levels we wish to press the analogy with the hardware machine very strongly: *where the integrity of the entire system is concerned, the operations provided should be as primitive as possible*. This is not to say that the operations should not be complete, but that they need not be convenient. They are to be regarded in the same light as the instructions of a central processor. Each operation may in itself do very little, and we require only that the entire collection is powerful enough to permit more convenient operations to be programmed.

The main reason for this dogma is clear enough: simple operations are more likely to work than complex ones, and if failures are to occur, it is very much preferable that they should hurt only one user, rather than the entire community. We therefore admit increasing complexity in higher layers, until the user at his terminal may find himself invoking extremely elaborate procedures. The price to be paid for low-level simplicity is also clear: it is additional time to interpret many simple operations and storage to maintain multiple representations of essentially the same information. We shall return to these points below. It is important to note that users of the system other than the designers need not suffer any added inconvenience from its adherence to the dogma, since the designers can very well supply, at a higher level, programs which simulate the action of the powerful low-level operations to which users may be accustomed. They do, however, profit from the fact that a different set of operations can be programmed if the ones provided by the designer prove unsatisfactory. This point also will receive further attention.

The increased reliability which we hope to obtain from an application of the above ideas has two sources. In the first place, careful specification of an orderly hierarchy of operations will clarify what is going on in the system and make it easier to understand. This is the familiar idea of modularity. Equally important, however, is a second and less familiar point, the other pillar of our system design, which might loosely be called ‘enforced modularity’. It is this: if interactions between layers or modules can be *forced* to take place through defined paths only, then the integrity of one layer can be assured regardless of the deviations of a higher one.[4] The requirement is a strong one, that no possible action of a higher layer, whether accidental or malicious, can affect the functioning of a lower one. In general, hardware assistance will be required to achieve this goal, although in some cases the discipline imposed by a language such as Algol, together with suitable checks on the validity of subscripts, may suffice. The reward is that errors can be localized very precisely. No longer does the introduction of a new piece of code cast doubt on the functioning of the entire system, since it can only affect its own and higher layers.

## CAL-TSS

The above considerations were central to the design of the CAL Time-Sharing System. CAL-TSS is a large, general purpose time-sharing system developed for the dual-processor Control Data 6400 at the University of California, Berkeley. We present here a brief sketch of the important features of this system.

Four aspects of the hardware are important to us (see figure 1).

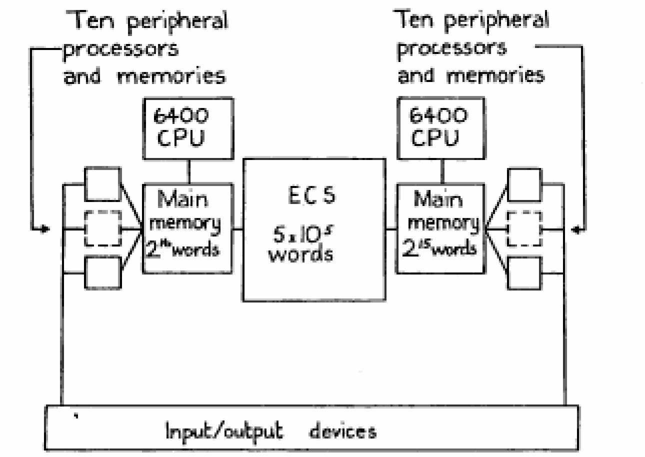

Ten peripheral processors and memories
6400 CPU
Main memory $2^{15}$ words
ECS $5 \times 10^5$ words
Ten peripheral processors and memories
6400 CPU
Main memory $2^{15}$ words
Input/output devices

**Figure 1**: Dual 6400 hardware configuration

1. The two processors have independent main core memories and communicate through the extended core storage (ECS), which has a latency of 4 microseconds and is capable of transferring ten words of 60 bits each in one microsecond. This means that a program of 10,000 words or about 30,000 instructions can be swapped into core in one millisecond. The ECS is regarded as the normal repository for active data. Only one user program at a time is held in each main core memory.

2. Each processor has very simple memory protection and mapping consisting of a relocation register and a bounds register.

3. The entire state of a processor (i.e. all the information not in memory which is required to define a running program) can be switched in 2 microseconds.

4. Input/output is handled by ten peripheral processors for each central processor. They all run independently, each with its own 4K x 12-bit memory. All can access through main memory (but not ECS), all can switch the state of the central processor, and all have exactly the same access to the input/output devices of the system. There are no interrupts.

The software is organized into a basic system, called the ECS system, which runs on the bare machines and is a single (lowest) layer for protection purposes, and any number of modules which may form higher layers. The ECS system implements a small number of basic *types* of *objects*, each one of which can be named and referred to independently (see figure 2).

File
Process
Event channel
Capability list (C-list)
Operation
Class code
Allocation block

**Figure 2**: Types of objects in the ECS system

Data in the system is stored in files. Each file is an ordered sequence of words which are numbered starting at 0. Operations exist to address a block of words in a file and transfer data between the block and memory.

A process is a vehicle for the execution of programs, a logical processor which shares the physical processor with other programs.[5] It consists of a machine state, some resources which it expends in doing work, and some additional structure to describe its memory and its access to objects (see figure 3). The details of this structure are the subject of later sections of this paper.

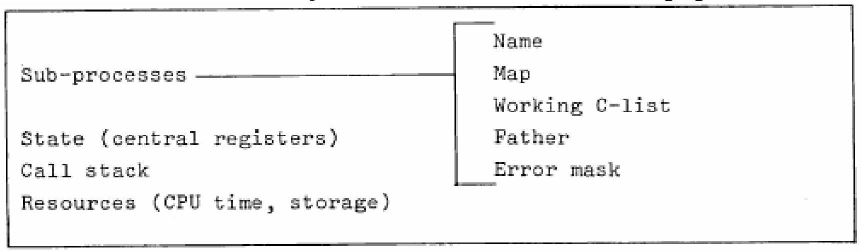


**Figure 3**: Components of a process

Processes communicate through shared files or through event channels, which are first-in first-out queues of one-word messages or *events*. A process may try to read an event from one of a list of event channels, and it will be blocked from further execution until an event is sent to one of the channels by another process. Many processes may be blocked on the same event channel, and many events may be sent to one channel before any process comes to read them.

Allocation blocks are used to control and account for the expenditure of system resources, which in the ECS system are only storage space in ECS and CPU time. Every object in the system is owned by an allocation block, which provides the resources for the space the object takes up and accumulates changes for the word-seconds of storage expended in keeping the object in existence. Allocation blocks also allow all the objects in the system to be found, since they form a tree structure rooted in a single block belonging to the system.

The remaining types of objects in the ECS system are closely related to the subject matter of this paper, and we now turn to consider them in more detail.

## Names and Access Rights

All objects in the system are named by *capabilities*, which are kept in capability lists or C-lists.[2] These lists are like the memory of a rather peculiar two-address machine, in the sense that operations exist to zero C-list entries and to copy capabilities from entry *i* of C-list A to entry *j* of C-list B. In addition, any operation which creates an object deposits a capability for it in a designated C-list entry. The function of a capability is two-fold:

1. it names an object in the system

1. it establishes a right to do certain things with the object.

At any given time a process has a single working C-list W. A capability is referenced by specifying an entry in w; the *i*-th entry will be referred to as W[*i*]. Since C-lists are objects which can themselves be named by capabilities, it is possible to specify capabilities in more complex ways; e.g., W[*i*][*j*] would be the *j*-th entry of the C-list named by the capability in the *i*-th entry of W. In the interests of simplicity, however, all capabilities passed to operations in the ECS system must be in W, and they are specified by integers which refer to entries of W.

In this rather obvious way a capability, which is the protected name of an object (i.e. it cannot be altered by a program) itself acquires an unprotected name. The integrity of the protection system is preserved because only capabilities in W can be so named. The fact that a capability is in W is thus construed as *prima facie* evidence that the process has the right to access the object which it names. From this foundation a complex directed graph of C-lists may be built up which provides a great deal of flexibility and convenience in the manipulation of access rights, although it is still limited in certain important respects which we shall explore later.

A capability is actually implemented as two 60-bit words (figure 4). The type field is an integer between 1 and 7 if the capability is for an object defined by the ECS system. Other values of the type are for user-created capabilities which are discussed below. The MOT index points to an entry in the Master Object Table, which in turn tells where to find the object, i.e. gives its address in ECS.

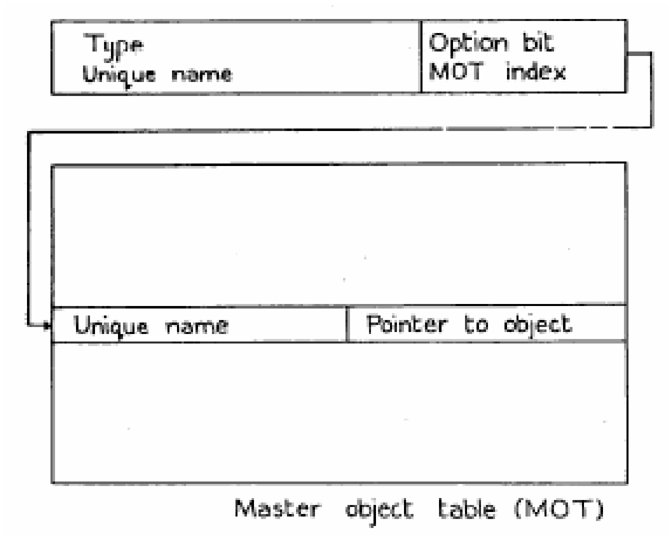


**Figure 4**: The structure of a capability

The unique name, guaranteed to be different for each object in the system, must be the same in the capability and in the MOT entry. This requirement makes it possible to destroy an object and reuse its MOT entry without finding and destroying all the capabilities for it. If we could not do this, it would be necessary to severely restrict the copying of capabilities and to keep an expensive and error-prone list of back-pointers from each object to its capabilities. Two additional benefits obtained from the MOT-unique name organization are that

1. if an object is moved in ECS, only the pointer to it in the MOT needs to be updated, since all references to the object are made by indirection through the MOT. Since the system's storage allocation strategy is a first-fit search of free blocks, followed by a compacting of free space if no block is big enough, it is essential to be able to move objects.

2. if some damage is accidentally done to a capability, it is extremely unlikely that the damaged capability can be used improperly, since the chance that the unique name will match with the one in the MOT is small.

It is worthwhile to note that a slightly modified version of this scheme, in which the MOT index is dispensed with and the object is found by association on the unique name, is also possible, although significantly more expensive to use.

The option bits field of a capability serves as an extension of the type. Each bit of the field should be thought of as authorizing some kind of operation on the object if it is set. A file capability, for example, has option bits authorizing reading, writing, deletion, and various more obscure functions. The operation which copies a capability allows any of the option bits to be turned off, so that weaker capabilities can easily be made from a given one in a systematic way without a host of special operations. The interpretation of option bits is also systematized, by the system's treatment of operations, to which we now turn.

## Operations

Operations can be thought of as the instruction set of a user machine created by the system, or as the means by which a program examines and modifies its environment. Viewing them in these ways, we want them to have the following features:

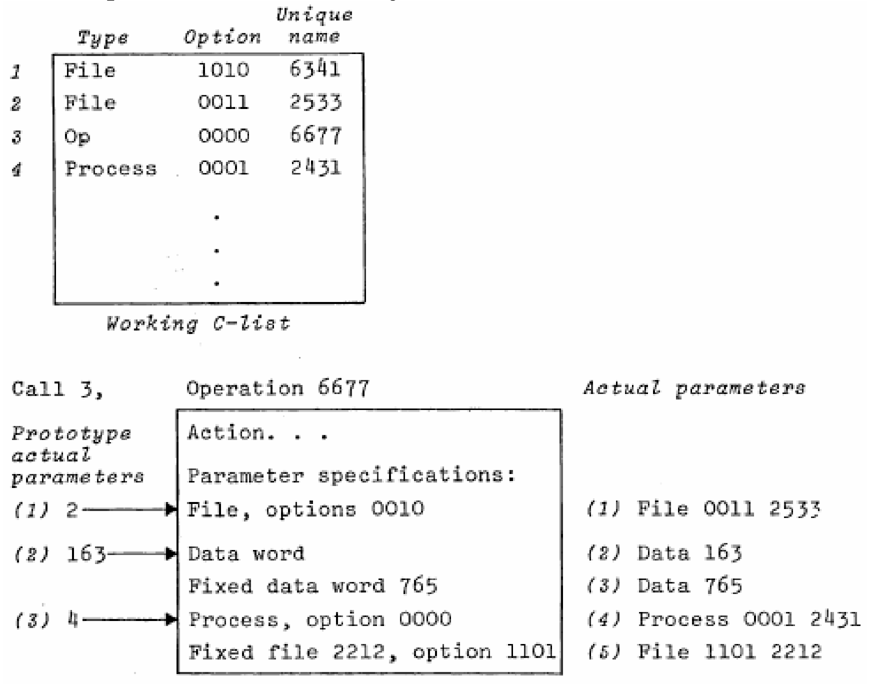
a. Operations can be handed out selectively, so that the powers exercised by a program can be controlled.
b. Mapping between the names which, a program uses to specify operations and the operations themselves can be changed, so that it is possible to run a program in an 'envelope' and alter the meaning of its references to the outside world.
c. New operations can be created by users which behave in exactly the same way as the operations originally provided by the ECS system.
d. A systematic scheme must exist to handle error conditions which may arise during the attempted execution of an operation.

The first two points are dealt with by treating operations as objects for which capabilities are required. This means that a process can only call on those operations which it finds in its working C-list W. Furthermore, since operations are identified only by indices in W, the meaning of a call on operation 3, say, can easily be changed by changing the contents of W[3]. When a program starts to execute, it expects to find at pre-arranged locations in W the operations which it needs in order to function. All of its communication with the outside world is therefore determined by the initial contents of W.

We now proceed to consider the internal structure of an operation in more detail. An operation is a sequence of *orders* which are numbered starting at 1. Each order consists of an *action* and a parameter specification list, which is a sequence of *parameter specifications* (PS). The action tells what to do; it is either an ECS system action or a user-defined action (discussed below). The PS list describes the parameters which are expected. Each parameter may be;

a. a data word, which is simply a 60-bit number
b. a capability, in which case the PS specifies the type and the option bits which must be on in the actual parameter.

When the operation is called (see figure 5) a list of prototype actual parameters must be supplied. Each one is a number. If the corresponding PS for order 1 calls for a data word, the number itself becomes the actual parameter; if the PS calls for a capability, the number is interpreted as an index in W and the capability W[*i*] becomes the actual parameter, provided that the type is correct and all the option bits demanded by the PS are set in W[*i*].



|  | Type | Option | Unique name |
|---|---|---|---|
| 1 | File | 1010 | 6341 |
| 2 | File | 0011 | 2533 |
| 3 | Op | 0000 | 6677 |
| 4 | Process | 0001 | 2431 |
|  |  | . . . |  |

**Figure 5**: Calling an operation: constructing the actual parameter list

Given an operation, it is possible to create from it a new operation in which some of the PS are converted into *fixed parameters*, i.e. the actual parameters for the action are built into the operation and are no longer supplied in the prototype actual parameter list. In this way a general operation may be specialized in various directions without the addition of any significant overhead to a call.

An action may *return* values in central registers after it has completed successfully. Of course, the caller can pass it files and C-lists in which it can return either data or capabilities of arbitrary complexity. It may also *fail* if it meets with some circumstance beyond its competence. In this case it has two options: it may return with an *error*, which is handled by mechanisms described later. Alternatively, it may take a *failure return*. The subsequent action of the system depends on the order structure of the operation. When the call on the operation is made, the PS list of order 1 is used to interpret the arguments and the action of order 1 is executed. The *order level i* of the call is set to 1. If the action takes a failure return, the system re-examines the operation to see if it has order *i*+l. If not, a failure return is made to the caller. If so, *i* is increased by 1 and order *i* of the operation is called.

The rationale behind this rather elaborate mechanism is to allow relatively simple operations to be supported by more complex ones. Suppose that A is a simple and cheap operation on a file which fails under certain, hopefully rare, circumstances. For example A might read from the file and might fail if no data are present. Now operation 3 may be devised to recover from the failure of A; it might attempt to obtain the missing data from disk storage. From A and B we make the two-order operation C = (A,B). A call of C now costs no more than a call of A if the data is present in the file. If it is not, B must be called at considerable extra cost, but this is hopefully an infrequent occurrence. The alternative approach is to call B and have it in turn call A. This is quite unsatisfactory if we are thinking in terms of a system which may eventually have many layers, since it requires passage through every layer to reach the most basic bottom layer. Such a design is acceptable if each layer expands the power of the operation, so that a great deal more work is normally done by a call on layer 2 than by a call on layer 1; not so when the higher layers are present to deal with unusual conditions, however, and normally add nothing to the work accomplished.

## User-defined Operations and Protection

The last section has suggested two ways to look at an operation:

1. As a machine instruction in an extended or user-machine.
2. As a means of communicating with the world outside a program.

A third analogy which is inevitably suggested is with an ordinary subroutine call. The crucial difference between a call on an operation and a subroutine call is that the former involves a change in the capabilities accessible to the process, i.e. in the working C-list. This is obviously the case when the operation is one defined by the ECS system, since the code which implements the operation is then running entirely outside of the system's protection structure. For a user-defined operation a more formal mechanism must exist within the overall protection structure for specifying how the capabilities of the process change when the operation is called.

To this end some additional structure is defined for a process (see figure 3). In particular, a new entity called a *subprocess* is introduced. Associated with each subprocess is a working C-list and a map which defines the memory of the subprocess. At any given instant the process is executing in one *active* subprocess (or in the ECS system itself) and consequently has the working C-list and memory of that subprocess. When the active subprocess changes, the memory which the process addresses and the working C-list also change, and the process consequently finds itself in a different environment.

Because a change in the active subprocess (i.e. a transfer of control from one subprocess to another) implies a change in capabilities, there must be some means for controlling the ways in which such transfers are allowed to take place. This is done as follows. A subprocess can only be called by calling on an operation which has that subprocess as its action; the means for constructing such operations are discussed below. The call proceeds as follows:

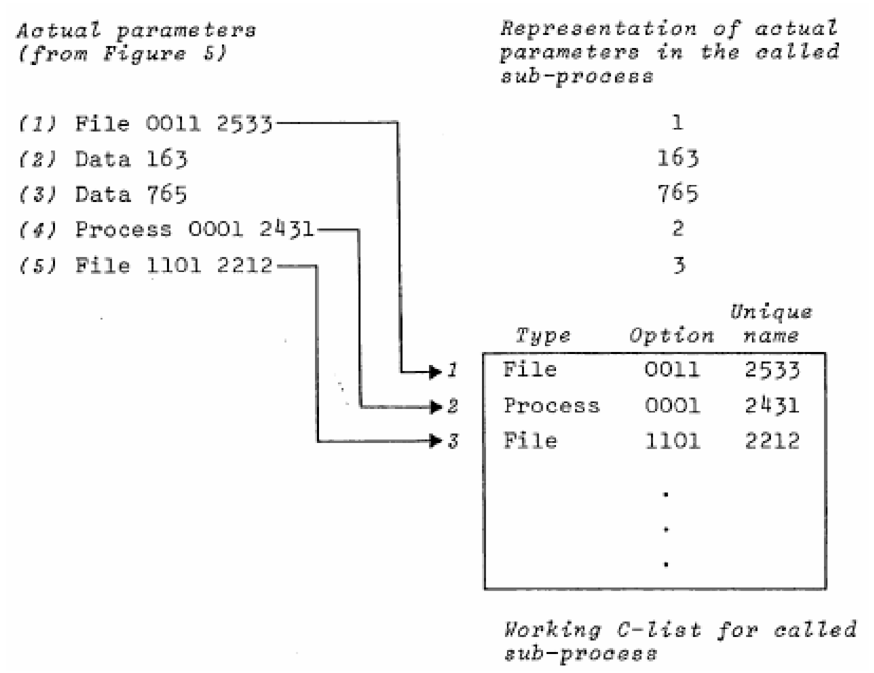


**Figure 6**: Calling an operation: copying the actual parameter list

1. Compute the actual parameter list (APL)

2. Copy a representation of the APL into a fixed place in the memory of the subprocess S being called; see figure 6. A data AP is represented by its value. A capability AP is copied into W, the working C-list of S and is represented by its index in W. Note that the representation of the APL could be used as the prototype APL for another call on the operation.

3. Start executing in S at a fixed location called the entry point.

This calling mechanism has been designed with some care to have the following features:

1. It is possible to control who can call a subprocess by controlling creation and distribution of operations with the necessary action.

2. Calls can enter the subprocess only at a single point.

3. The called subprocess need not have either fewer or more capabilities than the caller. Any capabilities needed by the called subprocess are passed as parameters. If capabilities are to be returned, the caller can pass a C-list into which the called subprocess deposits the new capabilities.

4. An operation which calls a subprocess is used exactly like one which calls the ECS system or any other subprocess. It is therefore very easy to run a program and intercept some or all of its calls on the ECS system or on the subprocesses.

A call operation implies a return. Since we do not want any unnecessary restrictions on the depth to which calls can take place or on recursion calls, a stack is used to keep track of return points. It is referred to as the call stack and is maintained by the ECS system.

## Access to Subprocesses

The careful reader may have noticed that we have refrained from saying that a subprocess is an object in the system. The reasons for this are twofold. First, a subprocess does not have existence independent of the process which contains it. Second, and more important, we wish to have a means for referring to 'similar' subprocesses with the same name in different processes, so that the same user-defined operations can be shared by a number of processes. To achieve this goal we introduce the last type of ECS system object, which is called a *class code* and consists simply of a 60-bit number. This number is divided into two equal parts, called the *permanent* and *temporary* parts. There are two operations on class codes:

1. Create a class code with a new permanent part, never seen before, and zero temporary part.

2. Set the temporary part of a class code. This operation requires one of the option bits to be set. It is therefore possible to fix the temporary part also, simply by turning off this bit.

A class code can be thought of as a rather flexible means for authorizing various things to be done without requiring the possession of a large number of Individual capabilities. A subprocess, for example, is named by a class code. This means that anyone with a capability for the class code can create an operation which calls on the subprocess with that name. When the operation is created, a capability for it is returned and can then be passed to other processes like any capability. If the class code is passed along with it, subprocesses with the same name can be created in several processes and called with the same operation. Since the permanent part of a class code is unique, there is no chance that independently created operations will name the same subprocess. Furthermore, it is easy to create a subprocess which cannot be called by any operation, simply by creating a new class code and then using it to name the subprocess.

The most useful application of the scheme is in connection with subsystems consisting of a number of operations, various files containing code and data, and perhaps several subprocesses. All the necessary information can be gathered together into one C-list and used to create copies of the subsystem (subprocess) in any number of processes.

Another application for class codes is built into the ECS system: they are used to authorize the creation of user-defined types of capabilities. Thus there is an operation which takes a class code and a data word and creates a capability (see figure 4) with type given by part of the class code and second word given by the data word. Such capabilities will always produce errors if given to ECS system operations, but they may be passed successfully to user-defined operations. In this way users can create their own kinds of objects and take advantage of the ECS system facilities for controlling access to them.

Class codes also provide a mechanism for identifying classes of users to the directory system. Rather than give a user capabilities for all files he owns, the log on procedure merely gives his process class codes which identify him to the directory system.

## Errors

A running program can cause errors in a variety of ways: by violating the memory protection, executing undefined operation codes, making improper calls on operations. Some orderly method is required for translating these errors into calls on other programs which will take responsibility for dealing with them. In order to do this, it is necessary to attach to each subprocess S another one to which errors should be passed if S is unwilling to handle them. We call this ‘next of kin’ the *father* of S and insist that the father relation define a tree structure on the subprocesses of a process, so that an error condition can be passed from one subprocess to another along a path which eventually terminates.

When an error occurs, it is identified by two integers called the *error class* and the *error number*. Every subprocess has a bit string called its *error selection mask* (ESM), and is said to *accept* an error if the bit of its ESM selected by the class is on. When an error occurs, a search is made for a subprocess A which accepts it, starting with the subprocess in which it occurs and proceeding along the path defined by the father pointers. The root of the tree structure is assumed to accept all errors. When A is found, it is called with the error class and error number as arguments. The entire current state of things is thus preserved for A to examine. If it wants to patch things up and continue execution, it can just return. If it decides to abort the computation, it can force a return to some subprocess farther up the call stack.

## Memory and Maps

It is fairly obvious that the right to access the memory addressed by a program is similar in nature to the right to access a file. Logically, it should therefore be controlled by a capability which should be mentioned every time an access is made. On a segmented machine this is a very natural and satisfactory point of view.[1] Lacking segmentation hardware, however, we must adopt a variety of compromises. Further complications are introduced by the fact that many machines, including the 6400, do not have address spaces large enough to allow all subprocesses to talk about each other’s memory, much less satisfactory controls on access to it. This section is concerned with the rather ad hoc schemes adopted in CAL-TSS to deal with this problem. More generally, it is of interest for two reasons:

1. To point out the salient problems to be faced by any system
2. To show how unpleasant are the expedients to which unsuitable hardware may force us.

No attempt is made to discuss the problem in the abstract or with any generality, and no argument is offered in favor of the devices described except expediency on the available hardware.

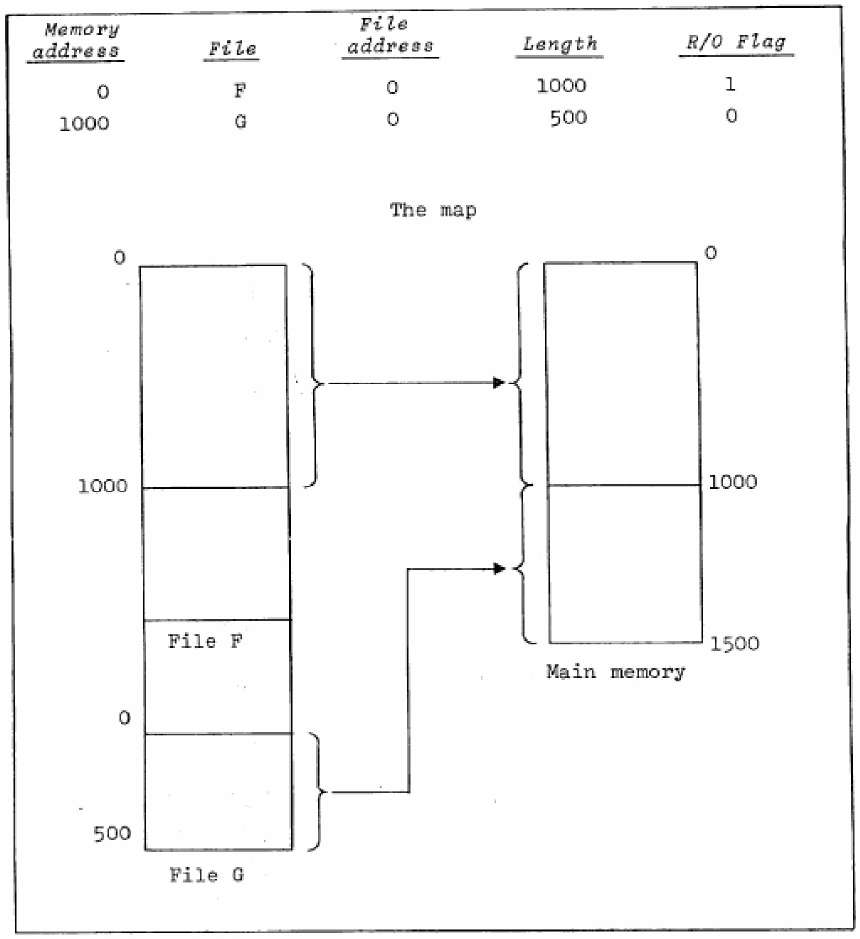
The first problem is to find a representation of a process' memory when the process is not running and therefore is not in main memory. In order to avoid introducing a new kind of object, and to facilitate the sharing of read-only code, a scheme suggested by hardware mapping mechanisms has been adopted. The memory of a subprocess is defined by a *map*, which consists of a list of entries each of the form:

memory address *m*,
file *f*,
file address *a*,
length *l*,
read-only flag *r*.

The strict meaning of such an entry is:

1. when the process is chosen to run next, *l* words starting with word *a* in file *f* are transferred into main memory starting at word *m* (relative to the hardware relocation register)

2. when the process stops running, the transfer is reversed if r=0. Otherwise, nothing is done.



| Memory address | File | File address | Length | R/O Flag |
|---|---|---|---|---|
| 0 | F | 0 | 1000 | 1 |
| 1000 | G | 0 | 500 | 0 |

**Figure 7**: Operation of the map

We usually think of this in the terms suggested by figure 7: a section of the file is made to correspond to a section of the addressable storage of the subprocess. The analogy breaks down if any modification is made to the file while the main memory copy produced by the map entry exists; this is extremely unfortunate if writeable data bases are to be shared, but unavoidable. The scheme works very well for sharing read-only programs and data, however. Multiple copies need exist only in central memory, which is not a precious resource since only one process is allowed to reside there at a time because of the very high swap rate.

The second problem is how one subprocess can access the memory of another one. A straightforward solution is to confine data sharing to files, but cursory examination reveals that this is extremely inconvenient, since the relationship between memory and files established by the map is quite indirect. On the other hand, appending the address space of the caller to the called program is also unacceptable, both because such a sledgehammer approach negates the selectivity of the rest of the protection system, and because the sum of the lengths of the two address spaces may easily exceed the available central memory. A restricted solution is to allow a subprocess to append to its address space the spaces of all its descendants along a single path to a leaf of the tree (see figure 8). The map for this extended address space is called a *full map* and is obtained by concatenating the maps of the subprocesses involved. It is constructed automatically whenever a subprocess calls one of its ancestors, since the path to be used is then uniquely defined.

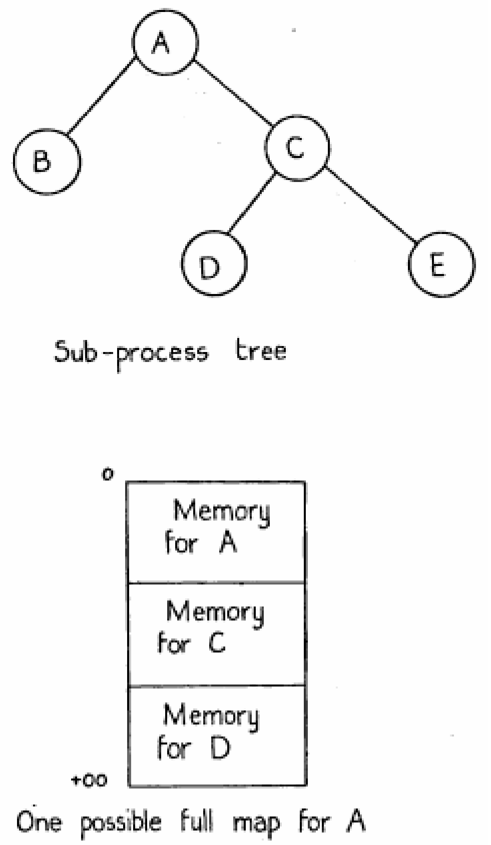


**Figure 8**: Full maps

With this scheme we have achieved:

1. Convenient addressing across subprocess boundaries for all the subprocesses on a path held together by father pointers,

2. A restriction of the total address space size for all the subprocesses on any such path to the total available main memory.

Note that when constructing a full map, it is never necessary to move anything, since when D is swapped in, say, the space required by A and C is known whether or not they are swapped in also.

## System Extendability

In the two concluding sections we present some thoughts on the general properties which we should expect an operating system to possess if it is to be a firm foundation for the construction and operation of software (including itself), and on methods for realizing these properties.

If a system is to evolve to meet changing requirements, and if it is to be flexible enough to permit modularization without serious losses of efficiency, it must have a basic structure which allows extensions not only from a basic system but also from some complex configuration which has been reached by several prior stages of evolution. In other words, the extension process must not exhaust the facilities required for further extensions. The system must be completely open-ended, so that additional machinery can be attached at any point.

Secondly, the relations between a module and its environment must be freely re-definable. This means that any module, provided its references to the outside world are documented, can be run inside an envelope which intercepts all of these references and re-interprets them at will. In order to ensure that external references are documented, it must be normal, and indeed compulsory practice to specify each module's environment precisely and independently of the module's internal structure. This requirement is satisfied by the use of C-lists to define the outside world to a subprocess.

Thirdly, it must be possible to introduce several layers of re-interpretation around a module *economically* without forcing all of its external references to suffer re-interpretation by each layer. In other words, a capability for extension by exception is required. Furthermore, the system's basic mechanism for naming and for calls must leave room for a number of higher-level subsystems to make their mark, rather than forcing each new subsystem to create and maintain its own inventory of objects.

To summarize, a usefully extendable system must be open-ended, must allow a subsystem to be isolated in an envelope, and must encourage economical reuse of existing constructs. Such a system has some chance of providing a satisfactory toolkit for others to use.

## System Reliability

Even more important than a useful system is a functioning one. Since we do not know how to guarantee the correctness of a large collection of interacting components, we must be able to break our systems up into units in such a way that

a. each unit is simple enough to be fully debugged

b. each unit Interacts with only a few other units

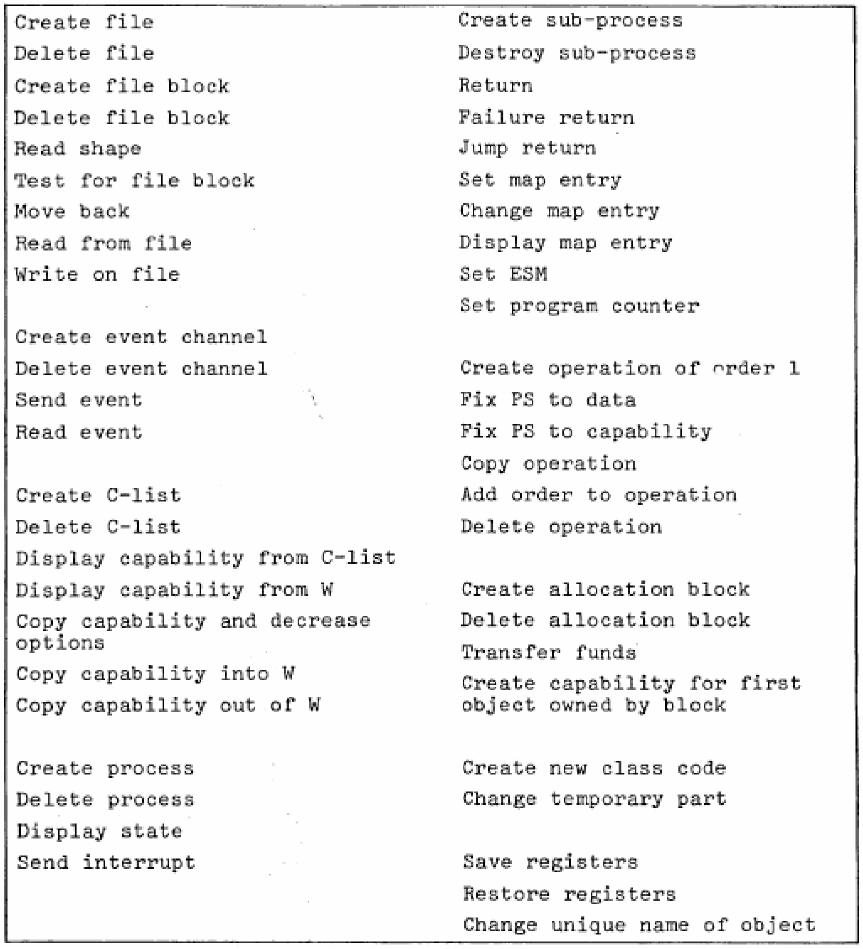
If this division is to inspire confidence, it must be enforced. It is not possible to depend on every contributor for good will and a full understanding of the rules for intercourse with others. Hence a complete and precise *protection* system is needed.

A great deal of *flexibility* is required in the manipulation of access rights. Otherwise the protection facilities will prove so cumbersome to use that they will quickly be abandoned in favor of large, monolithic designs. It must become a pleasure to write programs which safeguard themselves against the inroads of others, or at the very least it must be automatic, almost as unavoidable as the use of a higher level language.

Thirdly, the implementation must be *fail-fast*: it should detect a potential malfunction as early as possible so that corrective action can be taken. The use of unique names with pointers, redundant data structures, parity bits and checksums are all valuable devices for warning of impending disaster. On the other hand, elaborate and fragile pointer structures, or allocation tables which cannot be checked or reconstructed from the devices being allocated, are likely to cost more than they are worth.

More important, perhaps, is a general acceptance of the fact that a flexible and reliable system will exact its price. Under ideal circumstances, the price will be paid in careful design and modest amounts of special hardware to facilitate the basic operations of the system. More likely, though, are sizable amounts of software overhead to make up for basic deficiencies in the machine. Beyond a certain point (admittedly not often reached) there is little that can be done about this overhead. It can be minimized by keeping the goals of a system within reasonable bounds, but tends to be increased by the final consideration in reliability.

This, of course, is simplicity. Figure 9 lists the operations of the 6400 ECS system. There are about 50 of them, and few are implemented by more then a couple of hundred instructions, most by fewer. To convert the system they define into one suitable for a general user will take several times as much code at higher levels, hut it rests on a secure foundation.



Create file
Delete file
Create file block
Delete file block
Read shape
Test for file block
Move back
Read from file
Write on file

Create event channel
Delete event channel
Send event
Read event

Create C-list
Delete C-list
Display capability from C-list
Display capability from W
Copy capability and decrease options
Copy capability into W
Copy capability out of W

Create process
Delete process
Display state
Send interrupt

Create sub-process
Destroy sub-process
Return
Failure return
Jump return
Set map entry
Change map entry
Display map entry
Set ESM
Set program counter

Create operation of order 1
Fix PS to data
Fix PS to capability
Copy operation
Add order to operation
Delete operation

Create allocation block
Delete allocation block
Transfer funds
Create capability for first object owned by block

Create new class code
Change temporary part

Save registers
Restore registers
Change unique name of object

**Figure 9**: ECS system operations

## Acknowledgements

The system design described in this paper is mainly the work of the author and Howard Sturgis. Valuable criticism and most of the implementation were provided by Bruce Lindsay, Charles Simonyi, Paul McJones and Karl Malbrain.

## References

1. Dennis, JB., "Segmentation and the Design of Multiprogrammed Computer Systems", *J. ACM* **12**. October 1965.
2. Dennis, J.B., and E.C. van Horn, "Programming Semantics for Multi-programmed Computation", *Comm. ACM* **8**, 3, March 1966.
3. Dijkstra, E.D .., "The Design of the THE Multiprogramming System", *Comm. ACM* **11**, 5, May 1968.
4. Graham, R.M., "Protection in an Information Processing Utility", *Comm. ACM* **11**, 5, May 1968.
5. Lampson, B.W., "A Scheduling Philosophy for Multiprocessing Systems", *Comm. ACM* **11**, 5, May 1968.
6. Lampson, B.W., et al., "A User Machine in a Time-Sharing System", *Proc. IEEE* **54**, 12, December 1966.