The Control Structure of an Operating System

J Gray

IBM Thomas J Watson Research Center
Yorktown Heights, New York

B Lampson

Xerox PARC Research Center
Palo Alto, California 94304

B Lindsay

Department of Computer Science
University of California
Berkeley, California 94720

H Sturgis

Xerox PARC Research Center
Palo Alto, California 94304

IBM Research Report RC3949
July 21, 1972

ABSTRACT

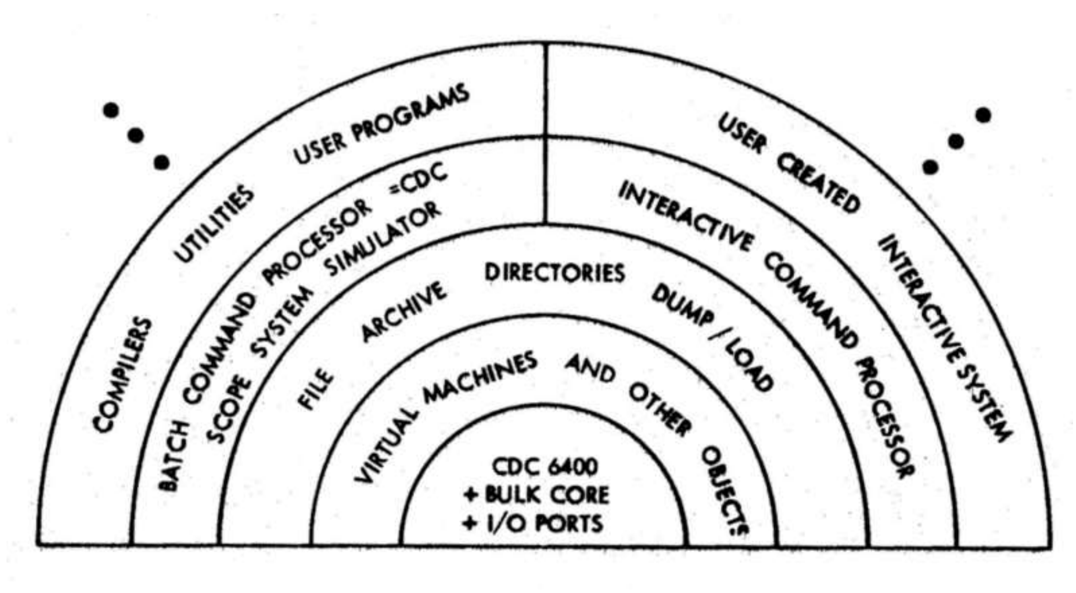
CAL is an operating system based on the concepts of capabilities and of implementation via machine extension. We first present some brief comments on our design philosophy and our experience with this approach. Extensions to the capability (descriptor) mechanism are described. The remainder of the paper concerns the control structure for (a) intra-process communication: creation and display of processes, domains and gates, various forms of domain activation, and the fielding of traps; and (b) inter-process communication: messages, events, interrupts, and locks.

1. Motherhood

The CAL operating system was begun in the fall of 1968. Within nine months the basic system had been implemented on an off-the-shelf CDC 6400 with extended core storage. Since that time it has been in daily use for further development and experimentation. Since the fall of 1970 it has been available to the Berkeley campus community. During this time we have had considerable experience with the system and are now in a position to judge its virtues and flaws. One of the richest aspects of CAL is its control structure. This paper describes those aspects of the control structure which we have found to be particularly useful. In doing this we have freely done violence to the realities of CAL. Obvious (but unimplemented) generalizations have been included. Mistakes made in the implementation are not repeated here. Concessions to the hardware are obscured.

The design of an operating system is like a Chinese puzzle spread out on a table: because of its size it is impossible to tell what it will be when assembled and whether there are too many pieces or too few. For this reason we have been careful not to depart too far from reality for fear of losing a piece or two. We have simply shaved off a few rough edges.

A reasonable way to design or model a complex system is to define the *objects* manipulated by the system and to define the *operations* which may be performed on these objects. This approach is usually called machine extension since it augments the universe of objects manipulated by the machine and the operations which can be performed. To give a trivial example, objects called stacks may be added to a machine by adding the operations `CREATE.STACK(N)`, `DELETE.STACK(NAME)`, `PUSH(NAME, ITEM)`,



COMPILERS
UTILITIES
USER PROGRAMS
BATCH COMMAND PROCESSOR ≡CDC
SCOPE SYSTEM SIMULATOR
FILE ARCHIVE
DIRECTORIES
DUMP / LOAD
VIRTUAL MACHINES AND OTHER OBJECTS
CDC 6400
+ BULK CORE
+ I/O PORTS
USER CREATED INTERACTIVE SYSTEM
INTERACTIVE COMMAND PROCESSOR

Figure 1. *The layers of CAL*

`POP(NAME)`, and the predicates `EMPTY.STACK(NAME)` and `FULL.STACK(NAME)`. This example points out two important aspects of design via machine extension:

- The design is modular and gives a functional specification for modular implementation. It ignores questions such as resource allocation which are properly implementation questions.
- The operations on an object completely define the object. No other form of access to the object is allowed. This permits great flexibility in implementation of the object (for example, the stacks above could be implemented as arrays or lists or functions). Such variations are functionally invisible.

The adoption of the extended machine approach is mitigated by two concessions to the limitations of the human brain and psyche:

- Operations must be conceptually simple and have a uniform interface to one another so that the validity of an implementation can be verified, and so that their use may be easily documented and explained.
- Each extension must be small enough to be completed in a time less than the attention span of the designers-implementers (typically one year) and less than the Ultimate Deadline set by the manager, customer, or finance company.

Extending a simple machine to a complex one may require several extension steps. This will give the implementation a layered appearance. The particular extension steps we chose are depicted in Figure 1. A bare machine was extended to a class of simple virtual machines, each with a virtual memory. These virtual machines are extended to have operations on a global file system. The file system machine is extended to accept commands, control the actions of the virtual machine, and to interpret file system naming. Most programs run ‘‘on top of’’ this machine, extending it in various ways.

Figure 1 may be misleading since it seems to suggest that higher layers are unaware of the lower layers. Not so. In an extended machine a layer may invoke operations implemented at any level below it. In particular, a layer often executes hardware instructions directly. The lower layers appear to be a single machine with some very powerful opcodes.

Except for differences in scale and generality, this is essentially the approach taken by Dijkstra and his colleagues in designing and implementing the ‘THE’ system.[1] In following this approach we have observed two phenomena which are not pointed out by Dijkstra.

Although it is indeed possible to be confident about the correctness of any particular module (operation), it is much more difficult to analyze the interaction of a group of modules. Similarly, it typically involves one man working one day to fix a bug inside a module. On the other hand, our entire group often spent weeks just discussing how to fix some flaw in the interactions among a group of modules. The most common and difficult faults we encountered were either errors or deficiencies in the design of the interfaces. It is difficult to foresee such errors since they are typically of a dynamic nature or they exploit some facility in an unforeseen way. As a corollary to this, the lower levels of the system tend to grow and change with time to accommodate these difficulties. In theory this is not necessary. All deficiencies in the lower levels may be corrected by appropriately extending the given machine. In practice a certain number of such extensions are done at a lower level.

This last point is a consequence of a second phenomenon: as the layers pile up, the cost of gate-crossing becomes significant. Operations which seem simple at a high level may unleash a flurry of activity at lower levels. This is primarily because of the rigidly enforced independence of operations, and because each operation at each level typically calls two or more operations at a lower level. One need only examine the function of 2^N for small N to see the consequences of this. Invoking a theorem which invokes all of set theory costs nothing—invoking an operation which invokes the rest of a computer system is not cheap.

We tried various validation and debugging procedures. Having a second person check all code (peer group programming) was the most effective. It had a positive effect on style, contributed to the general understanding of the system, and unearthed many bugs. Exercisers for modules proved to be rather difficult to construct and had to be maintained as the modules changed. They did serve as good tests of obscure cases but in general were probably not worth the effort. Manual constructions of formal proofs of the correctness of modules was tried only once. It was not cost effective.

Despite these caveats, misgivings, and scars we remain enthusiastic about the extended machine approach to design. Given the adoption of the extended machine approach, the important issues become:

- What are the objects that an operating system must implement and manipulate?
- What is a spanning set of operations on these objects?
- How can these objects and operations be glued together in a uniform way?

CAL proposes one answer to these questions. This paper discusses the nucleus of CAL. This layer contains the primitive objects of the system and the operations on them. It also contains a control structure and a naming structure which provide powerful extension facilities. Great care has been taken to allow for sharing and protection of objects.

An operating system may be viewed as a programming system. As such it must have a name structure, a control structure, and a syntax. We refer to Lampson[2,3,4] for a discussion of naming and of the related issues of protection and sharing. The issue of syntax seems to us to be a matter of taste and convenience. Our taste leans toward making the operating system interface a functional extension of a programming language like BCPL. The focus of this paper will be the control structure of CAL. Later papers will describe other aspects of CAL.

2. Objects

The nucleus of CAL implements the following objects:

1. files
2. processes
3. domains
4. gates
5. banks
6. event queues
7. capability lists

A *file* is a sequence of words of data numbered from zero to some dynamic upper bound. Operations exist to create, destroy, read, write, lengthen, and copy files. Files are variously known as segments and data sets in other systems.

Processes, domains, and gates will be discussed in greater detail below. A *gate* is an entry point into a domain. A *domain* is a sphere of protection (or a name space) within a process. A *process* is a scheduling and accounting entity. It may be thought of as the envelope containing a virtual processor.

Banks are the funding elements of the system. All resource use is charged against some bank. Banks also participate in resource allocation by limiting the resources of each category that a process may consume. When a bank is exhausted, any process charging against it is trapped. Operations exist which create and destroy banks and which transfer funds from one bank to another.

Event queues provide convenient communication and synchronization among processes as well as between processes and external devices. They are discussed at length below.

If an operating system is to implement objects, and operations on these objects, then there must be some way to name the objects. Clearly such names must be manipulated by the operating system and hence qualify as objects. This circular (recursive) reasoning has several fixed points variously called descriptors, capabilities, and control blocks. The particular fixed point one chooses depends on the issues of protection and scope.

If no special care is taken about protection, then any program may create a name and pass it to the operating system. Although such a decision has the virtue of convenience, it allows any program to name any object in the system. If the system intends to maintain critical tables, accounting information, sensitive data, or if the system intends to provide any form of protection among users, then creation of arbitrary names is not acceptable.

Hence names are made objects which only the system may manufacture. There are several possible implementations of such a scheme.[3] CAL adopts the scheme used by Burroughs[5] and by Dennis and Van Horn[6] of maintaining names in special objects called *capability lists*, ‘C-lists’ for short. These objects are variously called segment dictionaries, descriptor segments, or program reference tables in other systems. Associated with each domain of each process is a C-list. When executing in a particular domain, a process refers to objects by presenting an index into this list. Thus unprotected names (integer indices) are converted to protected names. By supplying each domain of each process with a distinct C-list, a very flexible system of protection and sharing is possible. A particular domain of a process can only refer to objects named directly or indirectly by its C-list. However, different capabilities for the same object may appear in several C-lists and so sharing of objects among domains is straightforward.

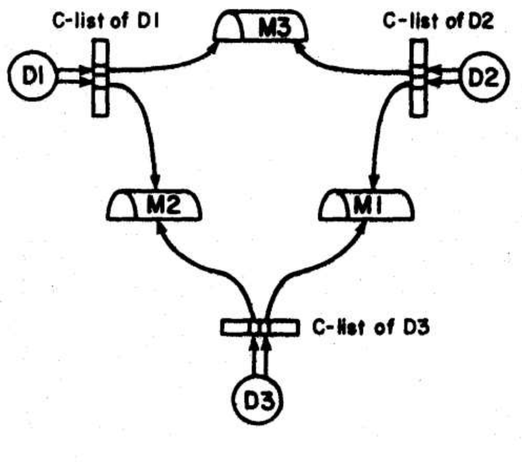


Figure 2a. *Three domains pairwise sharing two mail boxes via capabilities.*

```
begin
  procedure DOMAIN(NEIGHBOR1, NEIGHBOR2);
    mailbox NEIGHBOR1, NEIGHBOR2;
      begin
      •
      •
      •
      end;
    begin
      comment a new block to prevent free variable M1, M2 or M3 by DOMAIN;
      mailbox M1, M2, M3;
      parbegin
          D1: DOMAIN(M2, M3);
          D2: DOMAIN(M1, M3);
          D3: DOMAIN(M1, M2);
      parend;
    end;
  end;
end;
```

Figure 2b. *An 'ALGOL' implementation of Figure 2a.*

An example of this may be helpful. Suppose that each of three domains must share two of three mail boxes with its two neighbors and further that these mail boxes must be pairwise private. Figure 2a depicts a solution to this problem using capabilities. The capabilities are presumed to be allocated to those domains by some fourth domain which 'manages' the names of the mail boxes. This name manager may write directly into the C-lists of the domains D1, D2, and D3 or it may use the parameter binding mechanism of domain call. Solving this simple problem with the naming structure of most programming languages is non-trivial. One can prove that it is impossible with the static name structure of ALGOL. The issue is preventing `M(i)` from being global to `D(i)`. In ALGOL the solution is to use the parameter binding mechanism (call-by-name) and to eliminate free variable resolution and hence free variable 'capture'. Figure 2b displays this idea implemented in 'ALGOL'. A similar trick works for the Multics ring structure.[7, 8] The decision not to impose a particular naming structure on CAL derives from the experience with the B5000 stack mechanism and the Multics ring structure which make structures similar to those in Figure 2 difficult to construct.

CAL has some interesting extensions to the capability mechanisms described by Burroughs,[5] by Dennis and Van Horn,[6] and by Ackerman and Plummer.[9] The reader unfamiliar with the concept of capabilities should consult one of the above references before attempting to read the remainder of this section. In CAL a capability is represented by at least three fields:

```
TYPE
OPTIONS
OBJECT.NAME
```

and in the case of capabilities for files and for C-lists by two additional fields:

```
BASE
LENGTH
```

The generalizations of capabilities are as follows:

C.1. The concept that a capability is a protected name for the object it refers to has been generalized to allow other layers (e.g., users) to exploit this naming scheme for the new objects that they may implement. Only a few of the 2^{18} different types of capabilities are reserved by the nucleus (see the first paragraph of this section for a complete list). The remaining types of objects are made available to users as follows:

(a) The nucleus has an operation which will return a *license* to manufacture a particular type of object. This license is actually a gate to the system (a new operation) which will make new capabilities of a certain `TYPE`. The call looks like

```
GET.LICENSE()(MY.LICENSE)
```

where `GET.LICENSE` is a gate to the system. This gate returns a capability for a new gate (license) which makes capabilities of a fixed, unique type. `MY.LICENSE` is a C-list index to receive the gate returned by `GET.LICENSE`.

(b) Suppose that this call returns a gate which makes licenses of `TYPE=932`. Then we are assured by the nucleus that the license to make capabilities of this type will never again be given to a domain by the `GET.LICENSE` operation. Thus `MY.LICENSE` becomes a trademark of the process which owns `MY.LICENSE` and of any other domains that somehow have shared access to it.

(c) The `OBJECT.NAME` of a capability created by `MY.LICENSE` is specified by the caller. So for example

```
MY.LICENSE("PASSWORD")(KEY)
```

creates and returns a capability of

```
TYPE=932
OPTIONS=11...11
OBJECT.NAME="PASSWORD"
```

where:

`MY.LICENSE`	is the gate described above.
`"PASSWORD"`	is a string parameter specifying the new `UNIQUE.NAME`.
`KEY`	is an index into the caller's C-list to receive the resulting key.

(d) Instances of capabilities created by license (i.e., `TYPE` > 7) are called *keys*. Keys are like any other capability. They may be copied, displayed, passed, and returned. Hence names manufactured by users come under the protection umbrella of the system. Note that no one may modify the `OBJECT.NAME` of a key; no such operation exists. So a domain may manufacture keys and pass them out to other domains as the names of the objects implemented by the licensor. Since the licensor has exclusive rights to make keys of a certain type, it can be assured that whenever it sees a key of that type (as a parameter to some request made by some other domain), then that key was originally manufactured by a domain possessing the appropriate license. Thus, if a domain protects its license (or shares it selectively) then the domain can be assured that such keys contain valid (unmodified) information.

To give a concrete example: the disk system is licensed to make keys of `TYPE=9`. Any key of `TYPE=9` in the system is manufactured by the disk system. The `OBJECT.NAME` of such a key has a disk address in it (by a convention established within the disk system). Possession of such a key is proof of the right to access the named section of the disk subject to the constraints of the options of the key

C.2. The capability mechanism was extended to allow domains to share files and C-lists on a per-item and on a sub-file or sub-C-list basis. This is the purpose of the `BOUND` and `LENGTH` fields of these two capability types. This simple extension is transitive and allows for even tighter protection/sharing.

C.3. Indirection through C-lists (i.e., path names in the directed graph defined by C-lists) was found to reduce C-list sizes dramatically; for example, domains typically share a global pool of gates.

C.4. Almost all naming mechanisms have qualifiers which describe the type of operations which the capability allows (e.g., files are read, write, execute, ...). In CAL this has been generalized in two ways. The class of options has been expanded to allow for more diversity. This is then exploited by having the system gate keeper check the types and options of all actual parameters against the formal parameter list of a gate. The gate keeper traps the caller if the actual parameters are not consistent with the formals.

C.5. CAL allows capabilities to be passed between domains as events.

The ability to copy a capability and reduce the options of a capability is distributed freely. Only the system manipulates the other fields of a capability. In the case of a key, any domain licensed to manufacture a key may manufacture one with `OBJECT.NAME` and `OPTIONS` specified by the domain. The key `TYPE` is fixed by the license. Once created, the `OBJECT.NAME` field of a key cannot be changed, although anyone with the license can create a new key with the desired `OBJECT.NAME`.

Operations on C-lists include creation and destruction of lists, copy a capability from one list slot to another list slot (while possibly reducing the options), delete a capability, send or get a capability via an event queue, pass a capability as a parameter or result, and receive a capability as a result. Perhaps the most interesting operation is 'display capability', which returns the bit pattern representing the capability to be displayed. Since there is such an emphasis on privacy and security, protection within the system is not based on secrecy. Rather, it is based on a tight control on who may manufacture and reference names. Privacy is obtained by limiting access, by judiciously using options on capabilities (e.g., execute-only files), and by controlling the distribution of capabilities.

A general rule we have followed is that all the system tables (with the exception of the password file) should be open to public inspection. This strategy results in some minor violations of privacy (e.g., one can find out how much computer time some other user has consumed) but not in any violations of protection.

Further details on these topics can be found in a paper by Lampson.[2]

3. Domains and Processes

A domain defines an execution environment. All non-local names generated by a process executing in a particular domain are interpreted with respect to the capability list of that domain. Local names refer to names in the current stack frame (activation record) of the process. Domains are intended to provide fine grain protection and to provide for sharing of objects. When designing a process, one design goal is to separate the processes into several domains and thus to limit the instantaneous name space of the process to the objects of immediate interest. This facilitates verification and debugging and limits error propagation. The scope rules and block structure of most programming languages have similar motivations. However an example was given above which demonstrated that exclusive sharing is difficult to obtain in most languages. In this sense, domains are a generalization of common scope rules.

The domains of a process are organized into a rooted tree called the process tree. The purpose of this tree is to direct the flow of trap (error) processing and to define a priority for interrupts. The parent of a domain will be passed any traps not accepted by the domain. Any interrupts directed to the parent domain will interrupt the execution of any of its descendants. This will be explained in more detail below.

There is no concurrency within a process. Only one domain of a process is active at any instant. Concurrency can be obtained by spawning new processes or communicating with existing ones. See Figure 3 for an example of this.

A domain consists† of a capability list which defines its name space, a trap-gate which is an entry point to accept traps, a trap-accept vector which indicates which traps are acceptable and which are to be passed to the parent, a capability for the parent domain, and an interrupt inhibit flag, an interrupt buffer, and an interrupt lockout timer.

† In fact each domain also has a swapping directive associated with it. Since the CDC-6000 machines have only a relocation and bounds register, no attempt was made to provide virtual memory for the processor. Each process must explicitly allocate its memory. In this paper we will assume a segmented memory space and a process stack in the style of the B5000[5] and thus ignore these shortcomings.

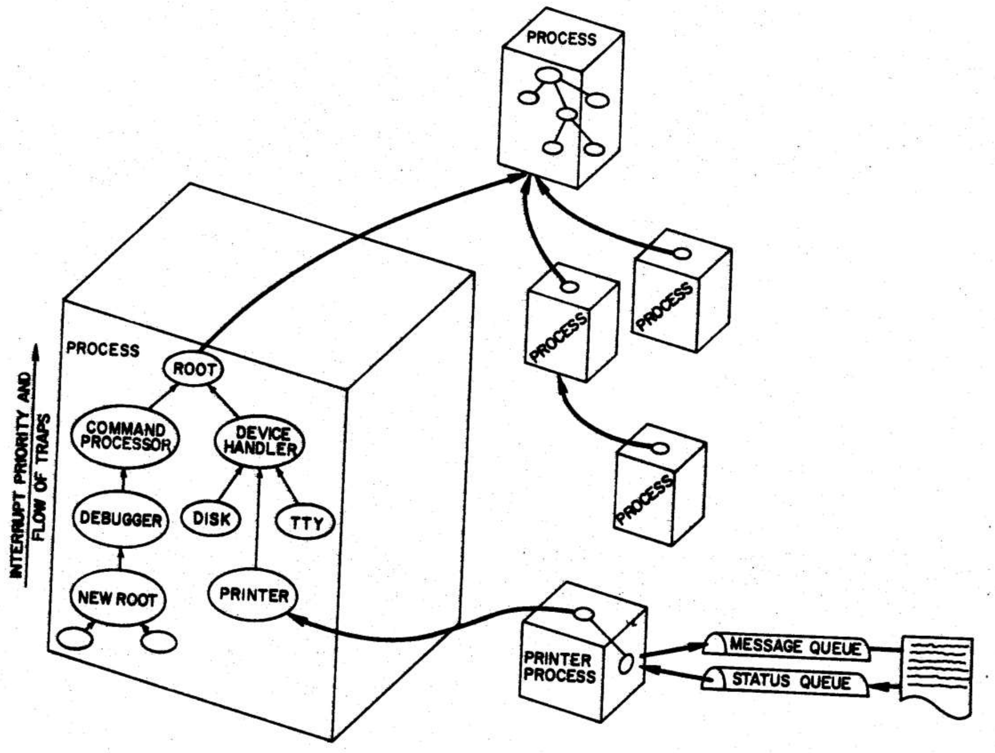


Figure 3. *A process debugging a new version of the ROOT while a spawned process concurrently prints a file.*

Assuming that the C-list for the domain has been created, the following operation creates a new domain in the process containing PARENT:† †

```
CREATE.DOMAIN(C.LIST, PARENT, TRAP, TRAPOK)(RESULT)
```

where:

`C.LIST`	is a capability for the C-list of the new domain.
`PARENT`	is a capability for the parent domain.
`TRAP`	is an index of a capability for a file in C-list and a displacement in that file (a file address). In the event of an accepted trap or interrupt, control will pass to this location.
`TRAPOK`	is a boolean vector such that `TRAPOK[I]=TRUE` indicates that the domain will accept the I'th trap.
`RESULT`	is the result of the operation. It is a capability for the newly created domain with all options allowed. The name `RESULT` refers to some C-list slot to receive this capability.

Most commonly, the capabilities needed by a domain are planted in its C-list when it is created. While executing, a domain may obtain a capability by creating an object, by receiving it as a parameter or as an event, or by receiving it as a result returned by some called domain.

† † The existence of options on capabilities means that a capability does not necessarily grant complete access to the object itdescribes. In this paper we will implicitly assume that all capabilities carry options which allow the specified access. Here for example we assume that the capability for the C-list allows it to be added to a domain, that the capability for the parent allows the addition of a descendant to the parent, and that the capability for the trap is a file capability which allows execution of the file. If any of these assumptions are violated, the gatekeeper will trap the caller.

The above operation simply creates a domain. This corresponds to declaring a block in a programming language. Below we will describe how gates are declared; they correspond to procedure entry points to domains. Then we will describe how such procedure entry points are used to construct domain activations.

Processes are extended virtual computers. When assigned to physical processors they execute instructions. CAL considers processes to be objects. Viewed in this light a process is a scheduling and accounting entity. A process is composed of a directory of its constituent domains, a stack of activation records of domains visited but not yet returned from (the call stack), the current processor state, a collection of clocks (user, system, swap), a collection of flags (active, ...), and a bank which will fund the activities of this process.

The following operation creates a process:

```
CREATE.PROCESS(C.LIST, PARENT, TRAP, TRAP.OK, BANK, START) (RESULT)
```

where:

`C.LIST`, `PARENT`, `TRAP`, `TRAP.OK`	specify the root domain of the new process as above.
`BANK`	is a capability for a bank which will fund the activities of the process.
`START`	is the initial state of the process executing in the root domain.
`RESULT`	is a C-list slot to receive a capability for the newly created result.

The domain created above is called the root domain of the process since it is the root of the process tree. The process is created in a suspended state. When activated by the `ACTIVATE(PROCESS)` operation it will begin execution with the state `START`. Any traps which the root refuses to accept are passed as an interrupt to its parent domain which is in some other process.

Process and domain destruction are somewhat simpler and are the same operation:

```
DESTROY(DOMAIN)
```

If the domain has no descendants and has no activation records buried in the process stack it is deleted, and if it is the root of some process, that process is deleted; otherwise, the caller is trapped with an error.

`DESTROY` is a generic function which, given a capability for any object, will attempt to delete it from the system. It will trap the caller if the capability does not have the destroy option enabled.

To summarize, a process embodies a virtual computer. It is a scheduling and accounting entity. Its execution is interpreted in the context of an activation of one of its constituent domains. Each of these domains provides an error and interrupt handling context as well as providing a name space: the local variables in the current stack frame plus the set of all objects pointed to directly or indirectly by the C-list of the domain.

Contrasting this to other systems, observe that:

CAL	has several domains per process,
ALGOL	has a hierarchy of domains (blocks) per process,
Multics[8]	has eight progressively smaller domains (rings) per process,
B5000[5]	has one domain (program reference table) per process, and
Dennis and Van Horn,[6] and B6500[10]	have several processes per domain (C-list or segment dictionary).

We chose to have several domains per process because each other scheme may be emulated by the first by appropriate indirection and sharing C-lists among domains and processes. Also we wanted to have several protected modules per process since the overhead of a process switch (scheduling, accounting, status, stack) is necessarily greater than that of a domain call.

4. Gates, parameters, and results

As described above, processes are decomposed into domains. Each domain may be viewed as a mode of execution having a different memory space and a different set of operations that it can perform. For example the master-mode slave-mode dichotomy of many systems can be emulated by creating two domains, one (the master) containing a capability for the C-list of the other (the slave) and also containing some privileged files and operations.

When a process is created and activated, it begins execution in its root domain with a processor state specified by the creation operation. Clearly there must be some way for the processor to move from one domain to another. Since domains are spheres of protection, this movement must be controlled by the protection system. In the example above it should not be possible for a processor in the slave mode domain to enter the master mode domain at an arbitrary location or with an arbitrary parameter list.

These considerations motivate the introduction of objects of type gate. In its simplest form, a gate is an entry point to a domain plus a recipe for creating an activation record for the called domain. If one domain, A, has a gate to a domain B then A may call B by using this gate and once called B may return to A through this gate. As mentioned in the introduction, we are looking for the set of objects that an operating system should implement and for a spanning set of operators on these objects. It should come as no surprise that the operators are themselves objects (gates). This has several satisfying consequences. The identification of operators and gates makes it impossible for a program to distinguish between a ‘‘user created’’ operator and a ‘‘system’’ operator. This property is vital to a layered system. After each extension, all the operators in existence have the same interface independent of the layer at which they are implemented. In fact the layers are completely invisible. Another virtue of making operators objects is that they come under the protection umbrella of the system. Thus the gates to the PL/1 compiler can be public and the gates to the directory system can be protected. Since this protection is dynamic, it is possible for any domain of a process to call any other domain of the process so long as the caller has the appropriate gate capability. This is another example of the flexible scope rules allowed by capabilities.

In many cases the caller wants to specify some parameters for the callee, and the callee wants to return some results to the caller. Since domains may share files and capability lists, this sharing is a simple but sometimes inconvenient matter for reasons analogous to the difficulties of COMMON storage in FORTRAN. It is sometimes desirable to be able to pass and return objects as parameters and results and make such binding dynamic. This, however, involves tinkering with the C-list of the caller and callee. A capability for a gate to a domain is not a capability for the domain: they are different objects. Hence the system gatekeeper must transfer the parameters and results between the domains when a gate is invoked.

In order to do this, a formal parameter and result list is associated with each gate when the gate is created. This list constrains the allowed types of each parameter and the required options for each type as follows:

- If the item must be a capability then the gate may constrain the allowed types of capabilities and for each type it may require certain options to be enabled.

- If the item must be data then the gate can specify the maximum amount of data to be passed or returned.

When a call is made to a domain, the caller specifies the actual parameters to be passed to the callee and gives the destinations of the results to be returned by the callee. The gate-keeper checks the types of the actual parameters against the formal parameter list. If they do not agree, then the caller is given a trap. On the other hand, if the parameters are consistent, then the capabilities are transferred to the low order slots of the callee’s C-list and the data are stacked in the callee’s local name space. The caller is suspended and the callee is activated at the entry point (file address) specified by the gate. All parameters and results are passed by value. The extension of the capability mechanism described earlier (see the discussion of objects, [C.2]) allows contiguous blocks of data and capabilities to be passed by reference. The callee is now assured that the number and type of parameters he requested were passed.

Conversely, when the callee returns some results the gate keeper checks them against the formal result list of the gate and traps the callee if they are not consistent. Otherwise the results are distributed in the caller’s name space as specified by the actual result list of the call.

To give an example, to create a gate into a domain:

```
CREATE.GATE(DOMAIN,ENTRY,FORMALS,RESULT,BANK)(RESULT)
```

where:

`DOMAIN`	is a capability for the domain to be gated. The gate option on this capability must be enabled.
`ENTRY`	is a file address interpreted with respect to the gated domain's C-list. This address will be the entry point for the gate. The file capability must have the execute option enabled.
`FORMALS`	is a list of the required parameter types and their required options.
`RESULTS`	is a list of the required result types and their required options.
`BANK`	is a capability for a bank to fund the existence of the gate.
`RESULT`	is a slot in the C-list of the caller to receive the capability for the new gate.

The operation `CREATE.GATE` is in fact a gate to the system. It has four parameters and one result. The types and options on these parameters and results are constrained as indicated above.

The function of `CREATE.GATE` is declarative. It corresponds to a procedure definition in ALGOL, or more closely to the `DEFINE` function of SNOBOL4. It constructs and returns an object of type gate which contains all the information needed to construct and bind a new activation of the domain to be called. It also specifies constraints on the actual parameters and results of each invocation of the gate.

It would be possible to merge the concepts of gate and domain by allowing a domain to have exactly one gate. There are few advantages to this and it makes the handling of constructs like multiple entry points in FORTRAN and PL/1 difficult. Not uncommonly all routines which work on a particular name space are grouped together in one domain, each with a separate entry point. For example, the directory system routines `LOOKUP`, `ENTER`, and `DELETE` coexist in one domain.

Gates are protected entry points into domains. They declare an interface definition and constraint which is interpreted by the system gate keeper. System gates and user gates are indistinguishable. This provides an elegant machine extension facility. Since gates are objects, they come under the protection/sharing umbrella of the system. The mechanisms for sharing gates are the same as those available for sharing files, C-lists, and other objects. Passage of the execution of a process from one of its constituent domains to another is always via a gate. This passage is carefully regulated by the system gatekeeper.

We conclude this section with two examples of how the return result mechanism can be replaced by appropriately passed input parameters. Suppose a domain wants the file `"FOO"` from the file system. The most direct way of obtaining it is by executing:

```
FIND("FOO",KEY)(FOO.SLOT)
```

where:

`"FOO"`	is the file name.
`KEY`	is the access key which the domain presents to identify itself.
`FOO.SLOT`	is a C-list slot for the returned capability.

A second strategy would be to pass a C-list slot as a parameter. Then the callee can fill it and no items need be returned. One creates the gate `FIND.1` which may be called by:

```
FIND.1("FOO",KEY,MY.CLIST.FOO.SLOT)
```

where:

`MY.CLIST.FOO.SLOT`	is a capability for the subsegment of the C-list of the calling domain which will receive `FOO`. This subsegment is one entry long.

The above technique is flawed by the fact that the caller cannot insure that a file will be placed in `FOO.SLOT`. That is, the type and option bit checking afforded by the gate keeper has been lost. A partial solution to this is to pass a gate to the caller which writes `FOO.SLOT` rather than passing the slot itself. This gate can test the type of object before placing it in the C-list.

First a gate called `WRITE.FOO.SLOT` is created:

```
CREATE.GATE(CALLER,WRITE.FOO,('FILE','MOVE'),,BANK)(WRITE.FOO.SLOT)
```

where:

`CALLER`	is a capability for the calling domain.
`WRITE.FOO`	is a file address which contains the code:

```
WRITE.FOO: MOVE(0,MY.CLIST.FOO.SLOT)
           RETURN
```

`'FILE','MOVE'`	constrain the parameters to the gate `WRITE.FOO.SLOT` to be files which can be moved around in C-lists.

Given the existence of the `WRITE.FOO.SLOT` operator, the `FIND` command may be redone as:

```
FIND.2('FOO',KEY,WRITE.FOO.SLOT).
```

This example generalizes to more complex situations. If the caller has an intricate data structure and the constraints on it are very subtle, he may pass operations to read and write it rather than pass the structure itself. Extremely tight protection is possible using this mechanism.

5. Domain activation and binding

The previous section described how objects of type gate are constructed and gave some simple examples of their use. This section explains the set of operations that may be performed on gates.

It is possible to view gates as procedure entry points a la PL/1 or ALGOL-68. All parameters and results are passed by value. The value of a capability parameter is a copy of the capability. This provides call-by-reference and is fairly convenient when combined with the sub-file and sub-C-list mechanism described in (C.2) of the ‘Objects’ section above.

This abstraction of gates will satisfactorily explain almost all uses of the gate mechanism and is all the naive user need know about the subject. However, to explain the operations `JUMP.CALL`, `JUMP.RETURN`, and `TRAP.RETURN` as well as the process `DISPLAY` operators it is necessary to introduce the concept of a *domain activation record*.

As described above, invoking a gate is a request to switch the execution of the process from one domain to another, and returning is a request to resume the execution of the caller. The system has a rather different view of this situation. Domain `CALL` and `RETURN` are simply operations on the process stack. Calls construct domain activation records from a gate, a parameter list, and a result list and place this activation on top of the process stack. Given an actual result list, the `RETURN` operation interprets the topmost domain activation to bind the actual results to the formal result names in the caller’s domain. Then the returning domain activation is erased from the stack.

A domain activation record contains all the information necessary to resume the execution of the domain on some processor. In particular it contains:

A pointer (perhaps implicit) to the activation of the caller.

The formal result list specified by the gate and the bindings on the actual results as specified by the caller.

The processor state (registers).

An interrupt disable flag.

The actual parameters.

Some local storage.

Figure 4 gives a more complete picture of the relationships among gates, domain activations, domains and processes. It also displays the structure of each object.

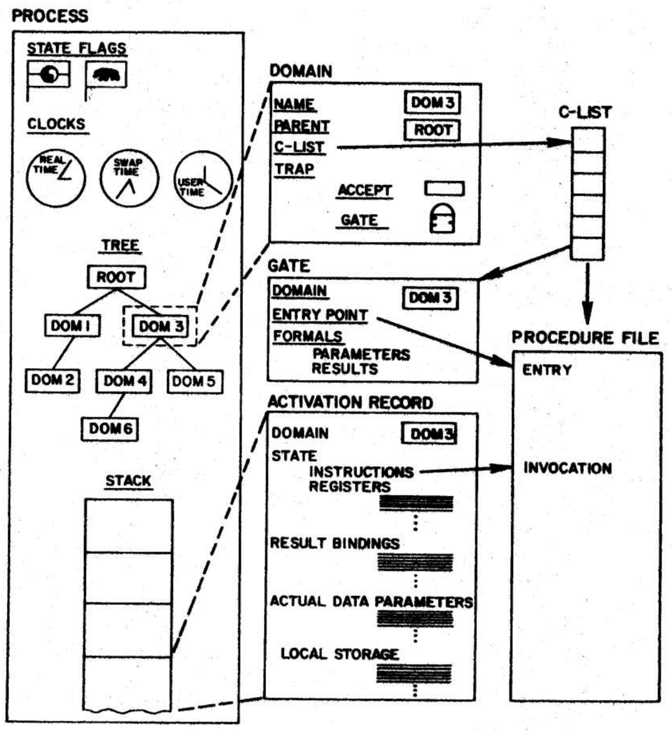


Figure 4. *The structure and interrelationships among processes, domains, gates and C-lists.*

When a domain executes a `CALL` operation such as:

gate(*actual parameter list*)(*result binding list*)

the system gate-keeper first checks the number and types of the actual parameters and the actual results against the number and types specified by the gate. If they do not agree, the caller is trapped. Otherwise, using the gate and the parameter and result lists, the gatekeeper constructs a new activation record for the called domain and pushes it on to the process stack, thus making it the active domain. Each of the actual data parameters is then copied onto the process stack and each of the capability parameters is copied into the called domain's C-list starting at index zero. (Note: this precludes domains calling one another recursively with capability parameters, a flaw in the system design.) Then the processor state is set to begin execution at the entry point specified by the gate.

In order to implement cascade routines a form of `JUMP` is introduced:

`JUMP.CALL` *gate*(*actual parameter list*)

The effect of a `JUMP.CALL` operation is exactly the same as that of a `CALL` operation except that the activation of the callee *replaces* the activation of the caller in the process stack rather than being pushed onto the process stack. The activation of the caller is completely lost. This mechanism can be used to construct coroutines and other baroque control structures. Since the caller's activation is lost it cannot be returned to, so the callee's activation inherits the result bindings and constraints imposed on the caller (i.e. these constraints are copied over to the new activation).

The simplest form of return is to invoke the operation:

`RETURN`(*actual result list*)

The `RETURN` operation first checks the number and type of the actual results against the formal result list saved in the activation record. If there is an error the returning routine is trapped. Otherwise the gate keeper binds the actual results to the caller's domain as specified in the call (and saved in the activation record of the returning domain). Returned data is copied into the process stack; returned capabilities are copied into the designated C-list slots. Then the activation of the returning domain is erased from the top of the stack. This has the effect of making the returned-to-activation the topmost activation and hence it is assigned the processor. Note that if jumps are used then the returned-to-activation is not necessarily the activation of the caller of the returning domain. However, as described earlier, the jump mechanism preserves the result bindings of the returned-to-domain activation. The actual result list of the returning domain must satisfy these constraints.

It is sometimes convenient to return a trap rather than a result (see the section on 'Traps'). The operation

```
TRAP.RETURN(TRAP.NAME)
```

erases the activation of the domain which invokes it and generates the designated trap in the returned-to domain activation.

The return mechanism is also generalized. It is common to want to flush the stack, that is to delete some number of domain activations from it. The `JUMP.RETURN` mechanism allows this if the returning domain can present the appropriate capabilities for each domain to be deactivated:

`JUMP.RETURN(`*result list*,`C.LIST,N)`

where:

`C.LIST`	is a capability for a list of capabilities for all domains to be flushed from the stack.
`N`	is a number giving the depth of stack to be flushed.

A similar operation exists to return a trap to a domain activation:

```
JUMP.TRAP.RETURN(C.LIST,N,TYPE)
```

where:

`C.LIST,N`	are as above.
`TYPE`	is the type of trap to be returned.

These operations first check to make sure that the `C.LIST` spans the $N-1$ domains which will have activations erased from the stack. If not, the returning domain activation is trapped. They then examine the returned result constraints for the activation $N-1$ deep in the stack; if they are not consistent with the actual returned results, the returning domain is trapped. Otherwise, the gate-keeper erases $N-1$ activations from the stack and then behaves as though it were an ordinary return or trap return operating from the activation $N-1$ activations into the stack.

6. Traps

The hardware and software continually make tests on the validity of the operations of each process in an attempt to detect errors as soon as possible and thus to prevent their propagation. Arithmetic faults, address faults, invalid system requests, and bad parameter lists are the most common examples of the general phenomena called *traps*. When a trap occurs there must be a mechanism which allows the process to recover and continue execution. Such a mechanism has strong ties with the protection structure of a system, as is pointed out by Lampson.[4]

When a trap occurs, the execution of the trapped domain is interrupted and a new domain is newly activated at its trap-entry-point. The central issue is the selection of this new domain. One must specify an ordering on the set of domains of each process which will direct the flow of trap processing.

The most obvious choice of an ordering is the process call stack. However, it often happens that the caller is less privileged than the callee or that the two are completely unrelated (e.g., written by different authors). Hence the caller cannot in general be expected to correct the problems of the callee. The caller merely has a capability for a gate to the called domain, not a capability for the domain itself. This rules out the use of the call stack as an ordering. (Note that PL/1 ON conditions do use the stack and so violate this simple logic.)

The ordering clearly must take the form of increasing responsibility. If one domain refuses to accept a trap, then the trap is passed on to a more responsible domain. The process tree is constructed with exactly this relationship in mind. Each node is considered to be responsible for its descendants and, conversely, interrupts to a parent are considered to apply to the descendants as well. If one thinks of each domain as an ALGOL block and of the process as an ALGOL program, then the process tree can be viewed as the static block nesting of the program. The choice of propagating traps up the process tree is analogous to the decision to use the static rather than the dynamic block structure to solve the free variable problem. Lest the reader carry this analogy too far we point out that, unlike the ALGOL name structure, the capability mechanism is orthogonal to the process tree and that the meaning does not have any name scope rules associated with it. In particular, it is possible for any domain to know about any other domain and to share objects with it.

Having decided on an order for trap processing we now define the trap operations. Conceptually each trap is given a name and each domain may execute a command:

```
    ON(TRAP.NAME, ENTER)
```

which will cause the domain to accept the trap named `TRAP.NAME` and to be entered at `ENTER` with information in its process stack about the trapped domain. If a particular domain traps then the trapped domain's activation is saved and the system searches up the process tree, starting with the trapped domain, until a domain is found which is willing to accept the trap. The accepting domain is newly activated at its trap-gate entry point with parameters describing the trap. To make loops less likely, this trap-accept is turned off. The domain must explicitly reset the trap-accept condition in order to catch a re-occurrence of that trap.

Several trap names were left undefined to allow users to exploit the trap mechanism within their subsystems.

To disarm a trap for a domain and thus default it to the parent domain invoke the operation:

```
    OFF(TRAP.NAME)
```

In the course of debugging and testing it was found to be useful to be able to generate traps. This is especially important in testing time-dependent code. Hence the operation:

```
    TRAP(TRAP.NAME)
```

Also, some cases require the ability to report traps back to the caller. Thus an operation exists which returns a particular trap to the caller. This erases the callee from the stack and traps the caller at the location from which he made the call:

```
    TRAP.RETURN(TRAP.NAME)
```

Lampson[2] gives a detailed example of the use of these facilities.

Two anomalies remain. First, it is possible that the ancestor domain has a bug and that the trap processing will loop. Needham[7] proposes a scheme which solves this problem: never allow an error to repeat and put a tight time limit on each domain responding to a trap. Within this context it is difficult to write a fault tolerant monitor. For example, it is now common to use the hardware to detect arithmetic errors and also to allow a program a quota of errors greater than one. Needham's scheme would forcefully exit any program which produced an overflow twice. Such a mechanism should not be welded into an operating system, although such a strategy could be implemented within CAL by suitably coding the root and by clearing the trap-accept-vector of any other domain. In fairness to Needham, it should be mentioned that he proposed a more general scheme that does allow a fault tolerant monitor, but it does not solve the loop problem.

CAL treats the loop problem by faith in the trap processors, by disarming the trap accept flag of a domain when it is used to catch a trap, by strict accounting and, in the interactive case, by user console interrupts.

The second anomaly concerns the case in which the root is unwilling to accept a trap. In this situation, the process is suspended and an interrupt (of the appropriate type) is sent to the parent of the root domain. This parent is in a different process and presumably is in a better position to handle the trap. This gives an example of how convenient it is to identify traps with interrupts.

By far the most common example of a free trap (i.e. not accepted by the root) occurs when the bank of a process is exhausted so that the root domain cannot execute because it cannot pay for any resources. In this case the bankrupt process is suspended and an interrupt (of the appropriate type) is sent to the parent of the root domain. In order to extricate the destitute process, this parent must replenish its bank and reactivate the process. That will allow the root to run. Of course, the parent may opt to destroy the descendant process.

The discussion above has been idealized. In fact, CAL is not prepared to handle symbolic names and multiple trap entry points. Each trap is catalogued and given an index. A trap-accept-vector and a trap entry point is associated with each domain. When trap I occurs in process P the following algorithm is invoked:

```
begin
  TRAPPER := CURRENT;
  repeat
    begin
      if TRAP.ACCEPT(TRAPPER)[I] = true
        then begin
          TRAP.ACCEPT(TRAPPER)[I] := false;
          JUMP.CALL TRAP.ENTRY(TRAPPER)[I];
        end;
      if TRAPPER=ROOT
        then begin
          INTERRUPT(PARENT(ROOT),'TRAP' | P | I);
        end;
      TRAPPER := PARENT(TRAPPER);
    end;
  end;
end
```

To summarize, the operating system provides an error handling mechanism which is both flexible and yet consistent with the production philosophy of the system. Errors propagate up the process tree until a responsible domain is encountered.

7. Display and manipulation of processes

The handling of traps requires that one domain be able to examine the state of another. On the other hand protection requires that such examination be regulated by the capability mechanism.

The C-list of a domain may be recovered by the operation:

```
DISPLAY.DOMAIN.CLIST(DOMAIN) (RESULT)
```

where

`DOMAIN`	is a capability for the domain of interest with the display option enabled on the capability.
`RESULT`	is a returned capability for the domain C-list (with all options allowed).

Examining the activation records of a domain is somewhat more complex. To display the names of the domains active in the stack invoke the operation:

```
DISPLAY.STACK.SKELETON(PROCESS) (RESULT)
```

where:

`PROCESS`	is a capability for the process owning the stack.
`RESULT`	is a data area to hold the `UNIQUE.NAME`s of the domains associated with the successive activations in the process stack.

To display a particular activation of a domain in the stack invoke the operation:

```
DISPLAY.ACTIVATION(PROCESS,DOMAIN,I) (RESULT)
```

where:

`PROCESS`	is a capability for the process owning the stack.
`DOMAIN`	is a capability for the domain associated with the activation record.
`I`	is the index in the stack of the activation record.
`RESULT`	is a data area (stack) to receive the returned activation record.

To write into an activation record:

```
WRITE.ACTIVATION(PROCESS,DOMAIN,I,TARGET,VALUE)
```

where:

`PROCESS,DOMAIN,I`	are as above.
`TARGET`	is an index into the activation record.
`VALUE`	is a (block of) value(s) to be written into the activation record.

Only the processor state and local storage may be changed in this way; all other information is protected by the nucleus.

Since capabilities are used to regulate these operations, the protection is not violated and yet one domain may exercise complete surveillance and control over another.

Since a domain may not be destroyed until all its descendants are destroyed, it is important to be able to reconstruct the process tree. The operations to do this are:

```
ROOT(PROCESSS) (RESULT)
```

where:

`PROCESS`	is a capability for the process of interest.
`RESULT`	is a capability for the root domain.

and

```
SON(DOMAIN,I) (RESULT)
```

where:

`DOMAIN`	is a capability for the domain of interest.
`I`	is an index of the desired son.
`RESULT`	is a capability for the `I`'th immediate descendant of the domain if such a descendant exists.

Lastly there are operations to suspend the execution of a process and to activate a process. When created, a process is in a suspended state. Processes waiting on an event queue are also suspended. `ACTIVATE` enters a suspended process into the scheduler's queue and `SUSPEND` removes the process from the scheduler's queue:

```
SUSPEND(PROCESS)
ACTIVATE(PROCESS)
```

8. Interprocess Communication

CAL dichotomizes communication between processes as either synchronous or asynchronous. Synchronous messages are called events and asynchronous messages are called interrupts. Although events may be sent at any time, they arrive only when requested by the receiving domain. They are thus synchronous with the execution of the receiver. Interrupts may strike the receiver at (almost) any time and therefore appear asynchronous with its execution.

8.1. Event Queues

Event queues are objects implemented by the nucleus. When created, a queue is designated as containing either capabilities or segments of data. A capability queue may only handle capabilities, and a data queue may only handle segments of data.† Associated with each event in a queue is the name of the domain which sent the event. Any process willing to supply a bank which will fund a queue may create a queue. The creation is done as follows:

```
    CREATE(('C-QUEUE'|'D-QUEUE'), BANK, CAPACITY) (RESULT)
```

where:

`C-QUEUE`	says make a capability queue.
`D-QUEUE`	says make a data queue.
`BANK`	is a capability for a bank that will pay for space occupied by the queue.
`CAPACITY`	is the maximum number of words that the messages in the queue may occupy.
`RESULT`	is a capability-list slot to receive the capability for the created queue.

The operations on queues are extremely simple. Operations exist to add an event to a queue and to remove an event from a queue or a set of queues. If the queue or set of queues is empty and a process tries to get an event, the process is suspended and its stack is chained to each such queue until an event arrives at one of them. As the name ‘queue’ suggests, a queue may hold more than one event, in which case the events are queued in a first-in first-out sequence. If the queue is full then the put operation suspends the process and chains its stack to the queue pending more space in the event queue. If several processes are queued waiting for an event, only the first process receives the event. The other processes continue to wait. (The definition of ‘first’ may be complicated by keyed events. See below.) The suspense action may be modified by a wait time; if the process is queued for more than this time, it will be de-queued and given a trap return. One other modifier is possible: a key may be associated with a message by the sender. In this case a process must present the appropriate key to get this message. If the key is incorrect, the message will be invisible. As mentioned in (C.1) above, keys are an extension of the capability mechanism; they provide protected names.

The format of messages is:

`UNIQUE.NAME`	of sender
`KEY`	optional key
`MESSAGE`	capability or block of data

The operations on queues are:

```
    PUTC(QUEUE, MESSAGE, WAITTIME, KEY)
    PUTD(QUEUE, MESSAGE, WAITTIME, KEY)
```

and

```
    GETC(QUEUE, WAITTIME, KEY) (MESSAGE)
    GETD(QUEUE, WAITTIME, KEY) (MESSAGE)
```

where

`C`	implies the message is a capability.
`D`	implies the message is a data segment.
`QUEUE`	is a capability or list of capabilities for the relevant queue or queues.
`MESSAGE`	is either a data segment descriptor or a capability to be sent or obtained.

† This is a restriction implied by the gate type constraints.

`WAITTIME`	is the maximum real time, in microseconds, that the process will wait in a queue.
`KEY`	is a key (optional to the sender) which is associated with the message. If present, the receiving process must present the key to obtain the message.

The queue mechanism is designed to facilitate convenient communication among producers and consumers of data and capabilities. For example, all external devices (terminals, tapes, card readers, printers, ...) are either producers or consumers or both. In CAL each of them communicate with internal processes via event queues.

The teletype communication facility exploits most of the features of queues. There is a process which listens to all the teletypes (it is actually run on a peripheral processor). The listener sends and receives characters from these external devices. For full duplex devices it echoes many of the characters that are sent.

The listener communicates with internal processes by using two D-queues, one for input and one for output. Whenever a special character arrives or whenever the teletype has sent a complete line, the listener puts the message into the input queue along with a key identifying the teletype. These keys allow all teletypes to share the same buffer without fear of one process intercepting another's input or output. This sharing reduces the space overhead since almost all buffers are almost always empty.

Whenever a process wants an input message from its teletype it executes

```
GETD(INPUT,10E7,KEY)(MESSAGE)
```

which will get a message, if there is one, from the teletype specified by `KEY`. The process will wait at most 10 seconds for such a message and if none arrives it will trap return. At this point the process may assume the user is in 'think mode' and cut back its working set in the swapping medium. On the other hand, if a message is returned, then the process will probably respond with a message to the output queue which specifies the echo for the message and some control information for the listener. Note that a message sender need not have the key as a protected name but that the receiver must possess the appropriate access key to get a keyed message.

This example shows how queues provide protected-pooled-buffered stream and block communication among processes. It is in sharp contrast with the buffer womping and flag setting interfaces presented by most operating systems as the I/O interface. Queues provide a complete interface to all external devices (e.g. terminals, disks, ...) and among the active processes.

It should be clear that keyed messages make a multiplicity of event queues almost unnecessary. It would be possible to have one queue global to the system and to use the keys as proxies for event queue capabilities. The central problem is that some malicious or errant process could clog or fill the queue and thus lock-up the system. The choice of having many disjoint queues stems from our ideas on accounting and on the isolation of one process from another. There is nothing to prevent an implementation from using only a single storage pool for the event buffers. Certainly at the user level this can be done. In fact each class of input-output driver has a pair of queues for communicating with all instances of that device.

The example above presumes that the processes cooperate to the extent that they collect their messages so that the teletype queue does not become cluttered. In reality this assumption is violated and so when a process is aborted or when a terminal is reallocated, the terminal allocator selectively flushes the buffer by using the following operations:

```
DISPLAY.EVENTS(QUEUE) (LIST)
DISPLAY.GET.PROCESSES(QUEUE) (LIST)
DISPLAY.PUT.PROCESSES(QUEUE) (LIST)
```

where

`QUEUE`	is a capability for an event queue.
`LIST`	is a data area to receive a list of events waiting, processes waiting for an event or processes waiting for more room in the queue.

These display operations produce a skeleton of the queue. If the list describes one of the waiting process chains then it gives the `UNIQUE.NAME` of each process (these are used as identifiers below). When displaying the contents of the queue, the list contains an entry for each event. This entry has an identifier for the event, the name of the sending process, the key on the event if there is one, and the first word of the event.

Using these identifiers, the queue may be selectively flushed by erasing unwanted events, and trapping waiting processes:

```
PURGE.EVENT(QUEUE, ID)
PURGE.GET.PROCESS(QUEUE, ID)
PURGE.PUT.PROCESS(QUEUE, ID)
```

where

`QUEUE`	is the queue to be affected.
`ID`	is the identifier of the message or process to be removed from the queue.

Of course the privilege to manipulate a queue with such operations is controlled by the options of the capability for the queue and is not passed out to the general public. For example, only the terminal allocator has a capability for the low speed I/O queues with all options enabled. All other capabilities for those queues allow only the operations `GETD` and `PUTD`.

The producer-consumer relationship is not universal. Processes are often related by mutual exclusion. A common constraint is that at most one of a community of processes may be in a certain mode at a certain time (critical section); however, there may be arbitrarily complex constraints on the concurrent execution of a community of processes. Semaphores are often proposed as a solution to these problems. Binary semaphores may be simulated by creating a C-queue of one element. If `C` is any C-list slot and `Q` is a capability for the queue, then `SET(Q)` is equivalent to `GETC(Q)(C)` and `CLEAR(C)` is equivalent to `PUTC(Q,C)`. *N*-ary semaphores are obtained by a simple generalization of this.

Unlike requests for conventional semaphores, which are granted in an arbitrary way (for arbitrarily large epsilon, there exists a non-zero probability delta such that a process will have to wait for at least epsilon changes in state of the semaphore before the request of the process is granted), semaphores simulated by queues allocate the lock on a first-come first-served basis (there exists an epsilon such that delta is zero: namely any epsilon greater than the number of processes waiting for the lock when the request is made).

If the constraints on concurrent execution are actually arbitrarily complex, then the interlocking is best done by a lock scheduler.[11] This affords simpler logic, protection, interlock avoidance and recovery.

In the simpler case, one may replace the semaphore by a protected lock on the object. Rather than let the capability `C` above be a dummy capability, let `C` be the only capability for the object. Then `GETC(Q)(C)` gets exclusive access to the object and `PUTC(Q,C);FORGET(C)` relinquishes this access. Similarly a pool of scratch storage for a community of domains may be allocated on a first-come first-served basis by placing a capability for each scratch area in `Q`.

In the interest of completeness the operations:

```
LOCK(OBJECT)
UNLOCK(OBJECT)
```

could be included by associating a semaphore with each object.

Yet a third way of looking at queues is that they provide an alternative form of inter-domain call. Sending an event can be viewed as calling some domain with one parameter (forking), or returning a result to some domain (quitting). Requesting an event is analogous to requesting a result from a function (joining), or being called with that event as a parameter. It is sometimes the case that this organization of having one process perform actions for a community of processes is either simpler or more economic than having many copies of the active process embedded in each member of the community. Its possible advantages are concurrency, lower overhead due to domain creation and private memory, immunity from interrupts, and simpler access to data.

For example, it may be more efficient to have a phantom process which drivers the printers and which is driven by a queue of file capabilities than to have distinct copies of such a driver as a domain within each process. The printer driver when it needs a new job can absorb all pending events, add them to its schedule in some priority order, and then execute the most important one. No locks are required because only one process is manipulating the data.

This incidentally provides a convenient example of the need to send structured messages. We need to send the printer driver a message which contains a capability for the file to be printed, a capability for a bank to fund the printing, the priority of the print job, the format of the file, and the format and destination of the output (e.g. font, type of paper, ...).

As with the parameter passing mechanism described earlier, it is clearly desirable to be able to pass complex data structures to events. CAL has neither the hardware nor the language support to specify such data structures. We view this as a flaw in the system. It is desirable to be able to transmit arbitrary PL/1 structures via queues.

Wirth[12] gives another example in which it is logically much simpler to have a floating process driving the operator's console than to have a domain (routine) private to each process which performs this function. The listener described above is a third example of this.

One may generalize these observations and characterize the control structures of systems as being primarily:

P-driven	most actions are implemented as procedure calls synchronous with the execution of the process.
Q-driven	most actions are implemented as queued requests to floating processes.
T-driven	control information resides in tables and processes 'execute' these tables (shared files).
I-driven	the flow of control is directed by external interrupts.

Multics and CAL are considered to be primarily P-driven although many modules of Multics and CAL are Q-driven. In particular the I/O handlers, schedulers, and the phantoms feed on queues. The cores of most operating systems are I-driven. Transaction oriented systems such as TSS/360[13] and CICS[14] are primarily Q-driven. Syntax directed compilers and decision table languages are examples of T-driven systems.

The possible advantages of a Q-structure over a P-structure are:

(q.a) Concurrency is exploited.

(q.b) The overhead of a single floating process may be less than that of many private domains.

(q.c) The floating process is protected from user errors and interrupts.

(q.d) Access to data may be simplified, thus easing the interlock problem.

On the other hand:

(p.a) In order to exploit (q.a) fully it may be necessary to have several servers. This may vitiate (q.b) and (q.d).

(p.b) In all known systems which do accounting and priority scheduling, the cost of a domain switch is much less than the cost of a process switch.

(p.c) More importantly, the space overhead for a single process is considerably higher than the space overhead for a single domain because a process has a stack, and accounting and scheduling information.

(p.d) If one considers the possibility of errors, the message discipline between processes may have to be very complex. The trap mechanisms described previously do not extend to a Q-driven system in an obvious way. If the process must wait for a response to each request then (q.a) is vitiated; if not, then complex post-analysis may be required vitiating (q.d).

The virtues of the various control structures are catalogued in Figure 5. Ideally one would like to merge the concepts of procedure call, send event, and send interrupt. We have not been able to do this.

To summarize, event queues are provided to allow interprocess communication. Most communication with other processes and with 'the outside world' is through message queues (shared files, locks, and interrupts provide more primitive communication). The queues can pass either blocks of data or they can pass entire

	P-driven	Q-driven	T-driven	I-driven
Concurrency		*	*	*
Modularity	*	*		*
Quick response	*		*	*
Tight control	*		*	*
Error handling	*	?		*
Clean interface	*	*		
Conceptually simple	*	*		

Figure 5. *The Virtues of Four Control Structures.*

objects by passing a capability. The system will suspend a process until the message arrives (or is sent) unless the process overrides this suspense. Keys can be attached to messages so that they can be directed to a particular receiver.

8.2. Interrupts

At least one author has suggested the abolition of interrupts because they create so much grief.[12] Most theoretical models of control structure seem to lack the concept of interrupt for this reason. However, real systems are not at liberty to ignore the issue of what can be done in case the event queueing mechanism breaks down or in case continuous polling of an event queue is too expensive. The obvious answer is that it must be possible to interrupt the execution of a domain in some process externally and to cause the invocation of some new domain of the process.

Any domain may *interrupt* any other domain (including itself) so long as it has the appropriate capability for the interrupted domain. An interrupt is addressed to a particular domain of a particular process. If that domain or one of its descendants is active then the interrupt strikes, causing a new activation of the domain to which the interrupt was directed. Otherwise the interrupt is arrested until its target domain is active or is a parent of the active domain of the process (see Figure 3). The interrupting domain may specify a datum describing the interrupt. This datum along with the `UNIQUE.NAME` of the interrupter is placed in the call stack of the interrupted process. As mentioned earlier, interrupts are much like traps. The interrupted domain is activated as though a trap has occurred (i.e. at the trap gate entry point). Interrupts and events may interact with each other in a bad way: if the interrupted domain is waiting for an event, it is de-queued and its process state is modified so that when the interrupted domain is resumed it will immediately re-execute the call to get an event (i.e., the instruction counter is ‘backed up’).

The interrupt operators are:

```
INTERRUPT(DOMAIN, DATUM)
DISABLE.INTERRUPTS(DOMAIN)
ENABLE.INTERRUPTS(DOMAIN)
```

where `DOMAIN` is a capability for the domain to be interrupted, and `DATUM` is any datum.

The motive for using the process tree to moderate interrupt handling stems from the observation that interrupts are very much like traps. A domain should only be expected to observe interrupts directed to its ancestors. This corresponds to a priority interrupt system except that it puts a partial order rather than a linear order on the interrupt structure. This generalization was found to be both inexpensive and valuable.

The newly activated interrupt domain may perform any operations it desires, subject to the limitations of its C-list. In particular, it will examine the interrupt datum and the process state (by displaying the process stack) and it may interrogate the user if he is on-line.

Consider the example of a user sending an interrupt to his command processor domain from a terminal. The command processor will ask the user what action is desired. ‘DEBUG’, ‘PURGE’, and ‘RETURN’ are common responses. DEBUG causes the command processor to jump-call the debugger. The RETURN request directs the command processor to return to the interrupted domain as though nothing had happened.

The PURGE command is most interesting. Presuming that the command interpreter is powerful enough to have capabilities for all domains below its first stack activation, the command interpreter may simply jump-return to its first activation. This would destroy all the intervening domain activations. It is often the case that intervening domains would like to make a more graceful exit. They may want to close their files and write a suicide note to the user. Hence the PURGE command directs the command interpreter to `RETURN.TRAP(INTERRUPT)` to the interrupted domain. This initiates the trap processing mechanism mentioned earlier and allows the active domains to flush themselves out of the stack in an orderly manner.

If this fails, the user may interrupt the errant process and type `PURGE` N for any integer N. This will erase the top N activations off the stack and `RETURN.TRAP(INTERRUPT)` to the domain $N+1$ activations deep in the stack. (‘`PURGE`’ is an abbreviation for `PURGE 0`.)

There are times when one must abolish (disable) interrupts temporarily. Certain critical sections of code manipulating shared data must be executed without interruption. If such modifications are interrupted in mid-flight and another computation is scheduled, then the shared data will remain locked for a prohibitively long time. If the data is shared with the interrupter, this raises the specter of a deadly embrace. These problems occur at both the system and at the user level.

Two solutions are possible. One is to place the non-interruptible complex very high in the process tree so that only very high priority interrupts can strike. The second is to disable interrupts. The first is illusory. For pragmatic reasons it must be possible to interrupt any domain of any process. The fact that the interrupt is powerful will be seen to be irrelevant.

Only the second solution is tenable. Since it must be possible to invoke any other domain without fear of losing control, the scope of interrupt disable is global to the process.

The solution above has a flaw: it is now possible to construct a process which can never be interrupted. Hence interrupt disable has a real time limit. Associated with each domain (not each activation of that domain) is a timer. Disabling interrupts in a domain sets its timer to some quantum Q. At each instant that interrupts are disabled, some domain’s timer is running down. It is now possible for each domain to be assured of at least Q units of real time execution. Only the timer of the topmost domain in the stack with interrupts disabled is decremented. If this timer lapses before interrupts are enabled then the active domain is given a trap, interrupts for the entire process are enabled, and the highest priority interrupt strikes.

If a domain tries to return while it still has its interrupts disabled the caller will get an interrupt time-out trap. To prevent this misdirected trap, a returning process is trapped if the interrupt disable of the returning domain is set.

We want to be able to say that the maximum time for a process of N domains to respond to an interrupt is $N \times Q$. To insure this, we must add the constraints that a domain cannot reset its timer while the process continues in interrupt disable mode (i.e. all clocks clear at once) and that no new domains can be created while interrupts are disabled (otherwise N gives no bound). If these constraints are enforced then one may show that the maximum interrupt response time of a process is $N \times Q$.

This treatment of interrupts differs from that described by Lampson[2] in two ways. We decrement only one timer at any instant. When a domain calls another domain it really has no idea what may transpire. For example, a request to get a line from a teletype may actually be a request for a line from a file or a program. This dynamic linking means that a domain is only aware of its own execution and the interfaces and shared data it sees. This requires that the called domain be able to extend the interrupt disable quantum and that it will be able to do this without subtracting from the quantum of some other domain. The second difference is probably pedantic. We fix the number of domains and associate timers with domains rather than with their activations. This prevents a process from getting into a loop of a domain calling itself and disabling interrupts or a loop of creating a domain, calling it, and disabling interrupts. Process stack overflow is the only limit on such a situation in the BCC system.[2]

9. Conclusion

The implementation of CAL was undertaken because it was felt that the manufacturer-supplied operating systems did not allow the functions that were needed by a university computing community. After bitter experiences with other operating systems, we set generality, extensibility, rationality, and reliability as our design goals. The basic system which we have described was designed in two man years, implemented in three man years, and required four man years of polishing and redesign. It (the lowest level) consists of seven kinds of objects and of about seventy-five operations on these objects. The entire system (i.e., all levels of Figure 1) has about twenty five man years invested in it.

The control structure consists of domain call and return, trap processing, interrupt send, event get and put, and process create and destroy. As it turns out, six of the seventy-five operations account for 90% of the calls on the lowest level. The six most frequent operations are file read-write, domain call-return, and event get-put. Thus it is seen that the gate keeper and the call-return and the get-put operations are the most heavily used aspects of the system.

The system has been in operation for three years and currently averages one crash per forty hours of operation. Nine out of ten such crashes are due to unrecoverable hardware errors. The software is an order of magnitude more reliable than the hardware.

In a sense, CAL is a textbook system. It is easy to explain to a class. It is also fairly easy to use. For example, the SCOPE operating system was implemented on top of CAL by writing a domain which simulated the CDC SCOPE operating system using the operations of CAL quite heavily. This required about 5000 machine language instructions. The converse, running CAL on SCOPE, would be much more difficult.

The reader is probably convinced at this point that CAL is indeed general, extensible, rational, and even reliable. But how much does it cost? The answer of course depends. We shall simply compare the CPU time of CAL and SCOPE. SCOPE is a reasonably efficient system. On a computation-bound batch job, SCOPE and CAL deliver approximately the same fraction of the CPU to the user. However, a student batch job run on CAL requires six times the CPU required by SCOPE. Clearly one must pay something for generality and extensibility (reliability and rationality should come for free [but they seldom do]). Whether a factor of three or six is an acceptable price remains to be seen.

Acknowledgements

The implementation of a system exposes many flaws in the design and in fact the implementers redesign the system many times as the implementation proceeds. Hence the adage ‘‘designers=implementers’’. Many of the ideas in this paper are attributable to the people who worked with us in implementing CAL: Karl Malbrain, Gene McDaniels, Jim Morris, Paul McJones, Dave Radell, Charles Symoni, Keith Standiford, and Vance Vaughn.

Maarten van Emden and Alex Birman made several suggestions which have improved the presentation.

David Fischer, Laura Gould, Vincent Kruskal, Barry Rosen, Leo Rotenberg, and Steve Zilles pointed out several flaws in a draft of the paper.

Discussions with Prakash Hebalkar led to the presentation of P, Q, T and I driven systems.

References

1. E W Dijkstra, ‘‘The Structure of THE Multiprogramming System,’’ *Communications of the ACM* **11**(5) (May 1968).
2. B W Lampson, ‘‘Dynamic Protection Structures,’’ *Proceedings of the Spring Joint Computer Conference 1970*, New York, Spartan Books (1970).
3. B W Lampson, ‘‘On Reliable Extensible Operating Systems (A preview of CAL),’’ *The Fourth Generation*, Greenwich, Connecticut, Datamation (1972).
4. B W Lampson, ‘‘Protection,’’ *Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems*, Princeton, New Jersey, Princeton University (September 1971).
5. *The Descriptor – A Definition of the B5000 Information Processing System*, Burroughs Corporation, Detroit, Michigan (1961).
6. J B Dennis, E C van Horn, ‘‘Programming Semantics for Multiprogrammed Computations,’’ *Communications of the ACM* **9**(3) (March 1966).
7. R M Needham, ‘‘Handling Difficult Faults in Operating Systems,’’ *Third ACM Symposium on Operating System Principles*, ACM (1971).

8. M D Schroder, J H Saltzer, ‘‘A Hardware Architecture for Implementing Protection Rings,’’ *Communications of the ACM* **15**(3) (March 1972).

9. W B Ackerman, W W Plummer, ‘‘An Implementation of a Multiprocessing Computer System,’’ in *First ACM Symposium on Operating System Principles*, ACM (1967).

10. E A Hauck, B A Dent, ‘‘Burroughs B6500/B7500 Stack Mechanisms,’’ in *Spring Joint Computer Conference 1968*, Spartan Books, New York (1968).

11. J N Gray, ‘‘Locking,’’ in *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM (1970).

12. N Wirth, ‘‘On Multiprogramming, Machine Coding and Compiler Organization,’’ *Communications of the ACM* **12**(9) (September 1969).

13. *IBM System/360 Time Sharing System Concepts and Facilities*, International Business Machines Corporation (Form GC28-2003), Armonk, New York (1967).

14. *Customer Information Control System (CICS) General Information Manual*, International Business Machines Corporation (Form GH20-1028), Armonk, New York (1971).