

OPERATING SYSTEM NOTES

Charles Grant

July 27, 1973

Systems Group

Center For Research in Management Science

University of California

Berkeley

1. Introduction

The operating system consists of software which, in conjunction with micro-coded functions within the SIMPLE processor, serves to coordinate the execution of user programs. A program in execution is called a process. Memory is contained in logical units called segments which may consist of up to 64k words.

2. Unique names and objects

Segments and processes are examples of objects. Objects of other types will be discussed later. A 32 bit value is assigned to every object by the operating system. This value is assigned to no other object of the same type throughout the lifetime of the system. This value is called the unique name of the object. It is always stored with the object. The unique name is not to be confused with a symbolic name which may refer to an object.

3. Capabilities

A capability constitutes permission to access a specified object in one or more specified ways. Capabilities are stored in special segments called capability segments. Capability segments may be written only by the operating system. The format of a capability is

8	24
object type	access and type dependent data bits
unique name of object	

The ways in which a capability may be used will be described below.

4. Processes

A process consists of four segments: 2 code segments, a data segment and a capability segment. These segments are termed the closely held segments of the process. The data segment maintains certain status information about the process such as the p-counter value and stack length. A process is termed active if it has an entry in the Active Process Table.

Every process has a priority, and the microcode arranges that the highest priority ready process is running. A ready list is maintained by the microcode, and whenever a process becomes ready it is spliced into the ready list at its priority level. A process is ready if it is ON and AWAKE. If it is either OFF or BLOCKED or both, then it is not ready. The transitions among these states occur as a result of SIPU instructions. Furthermore, a process can be made AWAKE as a result of a hardware interrupt or a software trap.

When a process is ready, the operating system must guarantee that each of the closely held segments resides in core, because the SIPU makes this assumption.

An entry in the active process table is in the following format:

Process table ents

unique name of process					
csegbase			-cseg length		
p0segbase			-p0seg length		
plsegbase			-plseg length		
dsegbase			-dseg length		
RL backpointer			RL next pointer		
O	A	W	PRI	Q	PORT MASK
Total COMPUTE TIME					

O - on/off bit

A - awake/blocked bit

W - the Wake-up waiting bit

PRI - the 4-bit priority

Q - The compute time slice is to be used
for this process.

5. Segments

A process may directly read and write segments other than its data segment only by the use of capabilities. Segments for which a process has capabilities to read and/or write are called external segments of that process.

To access an external segment, a process executes an SIPU instruction specifying the index in the capability segment of the capability for the segment, the address, and the type of access (read or write) desired. If the specified capability is of type segment and permits the requested type of access, then the SIPU refers to the CORE WINDOW TABLE to attempt to locate the segment. The Core Window Table is a hash table keyed on segment unique name. An entry in the CWT corresponds to a segment or contiguous sub-section ("window") of a segment which currently resides in core. If an entry is found which corresponds to a window containing the desired address, then the access is completed, otherwise a TRAP occurs.

CWT entry

unique name of segment				
lower bound of window		length of window		
core base address		A	R	D

- A - available bit, indicating that this window is not only in core but available for accessing.
- R - referenced bit, turned on by the SIPU whenever the window is used to make an access.
- D - dirty bit, turned on by the SIPU whenever the window is used to make a write access.

6. Traps

A trap is the action taken by the SIPU whenever one of the following actions occur:

A process attempts an illegal operation.

A process attempts to access part of a segment which is not in core.

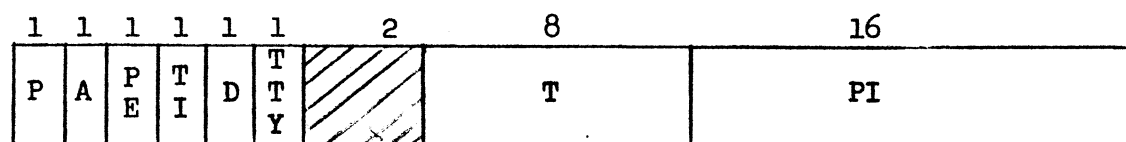
A process executes beyond its allocated time quantum.

A process executes an explicit TRAP instruction.

A trap has two results: one, information describing the reason for the trap is saved in designated cells of the data segment and in cell 0 of real Core; and, two, a special process, called the KERNEL is issued a WAKE UP. Thus if the KERNEL becomes ready, and is a higher-priority process than the one which trapped, the KERNEL immediately begins execution and is expected to resolve the issue which caused the trap.

If a trap occurs for a process with priority greater or equal to the KERNEL priority, then the SIPU will halt.

Cell 0 of memory:



P process trap

A APL processor signal

PE parity error

TI timer

D disk command completion

TTY terminal attention signal

T type of process trap

PI index of trapped process

7. System calls

System calls are requests made to the KERNEL for some function implemented by the KERNEL. To make a system call, a process must put the arguments to the call on its stack, put an identification of which system function is desired in its accumulator, and then execute the ATTENTION opcode.

8. Interprocess communication

The IPC mechanism exists to permit processes to communicate small messages containing data or capabilities in a controlled, synchronized manner.

Associated with each process are 16 ports on which a process can receive messages. A port is an object, and a process with a capability for a port of some process may send messages to that port. A port capability has the format:

8	8	16
port type		port bits
unique name of process		

A send operation is a system call which takes three arguments: the capability index of a port capability, a descriptor for a vector containing data to be sent in the message, and a descriptor for a vector of capability indexes which correspond to capabilities to be sent in the message. The data and the actual capabilities are copied into the Message Table if

- a) the port capability refers to an active process, and
- b) this message added to the current queue of unreceived messages on the port will not cause the number of words of message to exceed the maximum for the port.

Finally, if the destination process is dismissed waiting for a message on the port, then the KERNEL will schedule that process, forcing it to re-execute the code corresponding to the receive call.

A receive is a system call which takes three arguments: a port mask, a descriptor for a vector where received data is to be copied, and a

descriptor identifying a vector containing indexes which specify where in the capability segment that received capabilities are to be placed. The bits in the port mask specify from which of the 16 ports of the process a message will be accepted. The chronologically first message sent to any one of those ports is the one which will be received. If no message is queued to one of these ports, the process will be idle until one is sent. The receive operation will fail if either of the vectors is not large enough to contain a sent message.

The KERNEL maintains a fixed size Message Table in core for buffering IPC messages. The table is a chained hash table, keyed on the unique name of the destination process. A sent message is put at the end of the chain, and receive commands scan from the head of the chain for a message that is acceptable.

An entry in the MT has the following form:

unique name of dest. process		
unique name of source process		
port #	previous pointer	next pointer
	# caps	# data words
first data word		
.		
.		
.		
last data word		
first capability		
.		
.		
.		
last capability		

The user vector which receives the data sent with the message is formatted as:

unique name of sender		
# words of data	# caps	port #
first data word		
.		
.		
.		
last data word		

9. Directories

Directories are objects which exist to associate symbolic names with capabilities for objects on a long-term basis. A directory maintains a list of pairs (<symbolic name>, <capability for an object>). With the appropriate authorization (i.e., capability) for a directory, a process may look up a given symbolic name in the directory and obtain a copy of the corresponding capability. There are actually 3 levels of authorization which may be used to look up entries in a directory, and the one which applies is specified in the capability for the directory. Associated with each entry in the directory are 3 bit masks which correspond to the 3 authorization or access levels. When a lookup is done with access level i , the i th bit mask is ANDed with the option bits from the sought capability before it is copied. The 3 levels of access will normally be thought of in the following ways:

- 1) general public access
- 2) trusted friend access
- 3) owner access.

A directory is composed of a capability segment and a data segment. The first entry in the capability segment is a capability to read and write the data segment. The data segment is used to contain the symbolic names and access masks, while the associated capabilities are contained in the capability segment.

All operations on directories are performed by a process designated the Directory Manager. Requests to access directories are made via the inter-process communication mechanism. A capability for a directory has the form:

TYPE	option bits
unique name of capability segment	

option bits:

- L1 lookup with access level 1
- L2 lookup with access level 2
- L3 lookup with access level 3
- C create entries
- D delete entries
- M modify entries

(If more than one access level bit is on, then the highest one is used.)

The directory manager is empowered by the KERNEL (by virtue of having the appropriate capability) to convert a directory capability into a capability to copy capabilities into and out of the capability segment specified by the unique name.

Directories perform another function which is to designate the ownership of objects. A directory is said to own a particular object if the "ownership" bit is on in the directory data segment entry for an object. The ownership bit is used to do accounting on long-term storage of objects. Also, the ownership bit is used to keep track of which objects in the system are being paid for. The creation of system objects (e.g., segments, directories) is handled by the KERNEL, but the directory manager is the only process which is granted permission to request this action. The directory manager guarantees that each segment or directory, when created, has an ownership entry for it in some directory. Furthermore, when an ownership entry is deleted, the directory manager will instruct the KERNEL to destroy

the actual object. Thus the directory manager must be called in order for a user process to create a segment, and this call must specify a directory and symbolic name to correspond to the created segment.

An entry in a directory data segment contains the following information: The symbolic name of the object, the 3 access level option masks, the ownership bit and specification of which capability in the directory capability segment the entry corresponds with.

Since directory entries may be objects which are themselves directories, and since every created directory must be entered in some directory, all the directories in the system are part of one data structure with one original or "root" directory. Since the creation of a new directory and the creation of an ownership directory happen simultaneously, there are no loops in this data structure (when tracing ownership directories) and it is in fact a tree. The directory manager will accept "path names" to locate an entry, where a path name is of the form NAME 1. NAME 2...NAME n. Such a name means: Search a specified directory for an entry with name NAME 1. This entry should be a directory itself, and will be searched for name NAME 2, etc. The item in directory NAME n-1 with name NAME n is the sought object. Obviously, if any of the directories have no entry with the specified name, then the directory system must report failure.

When a call is made to the directory manager to look up a path name, a capability for the directory to be used to begin the search must be passed as part of the call. In fact, a variable number of directory capabilities may be passed, and if the first name of the path name is not found in the first directory specified, then the second directory will be searched, etc. This ability to specify a sequence of directories (referred to as a "scan

list") to use to try and find an entry permits, among other things, the notion of a "current" directory whose full path name need not be given in order to look up its entries. By default, when no path name is given then the first directory in the scan list will specify the directory. By convention, it is expected that calls on the directory manager will be routed through an 'EXEC' process which maintains the 'scan list' of directories for all user process for a given user.

The functions which will be provided by the directory manager are indicated below. Each of these calls is requested with an IPC call in a format to be specified later.

```
examine\entry (<name>, <scan list>) → <cap>, <access mask>, <owner list>
lookup (<name>, <scan list>) → <cap>
delete\entry (<name>, <scan list>) →
create\entry (<name>, <cap>, <access mask>, <scan list>) →
create\segemnt (<name>, <length>, <access mask>, <scan list>) → <cap>
create\directory (<name>, <access mask>, <scan list>) → <cap>
change\name (<name>, <simple name>, <scan list>) →
change\am (<name>, <am>, <am #>, <scan list>)
scan\directory (<name>, <entry #>, <scan list>) → <name>, <cap>, <am>, <ok>
create\cap\seg (<name>, <length>, <am>, <scan list>) → <cap>
```

10. IPC runtime routines

The interprocess communication system calls (section 8) provide a very low level form of process communication. This section describes a set of functions to be included as part of the standard SIMPLE Runtime Package to be used to interface to the IPC system calls. These routines provide a mechanism for assembling and then disassembling a sequence of "items" which comprise a message. An item which can be transmitted as part of a message may be a scalar, descriptor, vector, string, or capability.

The functions associated with sending messages are:

BEGIN\MESSAGE ()

Calling this function informs the runtime that a new message is about to be assembled and sent.

SEND\SCALAR (<scalar>)

This function appends the argument scalar to the message.

SEND\DESCRIPTOR (<descriptor>)

This function appends the two-word argument descriptor to the message.

SEND\VECTOR (<vector>)

This function appends the entire argument vector to the message.

SEND\STRING (<string>)

This function appends the contents of the string (as specified by the selector) to the message.

SEND\CAPABILITY (<index>)

This function causes the capability which is contained at the specified index (at the time the SEND\MESSAGE call is made) to be appended to the

message. Capabilities are appended in a separate queue from other types of items.

SEND\MESSAGE (<port capability>)

This function causes the assembled message to be actually sent to the specified destination port.

The following functions may be used to receive a message which was assembled with the above functions:

AWAIT\MESSAGE (<port mask>)

This function awaits the first message which is sent to any of the ports specified in the mask. Furthermore, it returns as value a reference to a two-word "header" for the message which is formatted as:

unique name of sender		
port #	caps	data

where port # specifies which port the message was received on; "caps" specifies how many capabilities came in the message and have not been removed from the message; "data" specifies how many words of data came with the message and have not been removed yet. These last two fields are dynamically maintained as items are removed from the message.

RECEIVE\SCALAR ()

This function removes a scalar from the message and returns it as value.

RECEIVE\DESCRIPTOR ()

This function removes a two-word descriptor from the message and returns it as value.

RECEIVE\VECTOR ()

This function interprets the next item on the message as a vector and returns a vector descriptor which locates the vector. The data located by the vector descriptor is valid up to the point that a new message is received.

RECEIVE\STRING (<string>)

The next item on the message is interpreted as a string, and the argument string is set up to locate the sent string. This data is also valid only until the next message is received.

RECEIVE\CAPABILITY (<index>)

The next unreceived capability which was sent with the message is copied to the location in the process capability segment specified by the argument index.

These routines are intended to be used to assemble and disassemble messages sent between cooperating processes. Very little verification is done on the received items to guarantee that they are of the type expected. The routines will fail, for example, if there are no remaining data or capabilities on the message, or if a supposed vector item size is not within the size of the message.

The format of the actual message for the implementation of these routines is shown on the following page. The action of RECEIVE\VECTOR (), for example, is to take the next word of the message, set its SEGMENT bits to be of type GLOBAL, and adjust its base address to agree with its actual base address. The size field is used to update a pointer in the message to where the next item begins.

Format of actual message

un of sender		
port #	nwdata	ncaps
items of data		

scalar

actual scalar

vector or string

first word of vector desc
elements of the vector

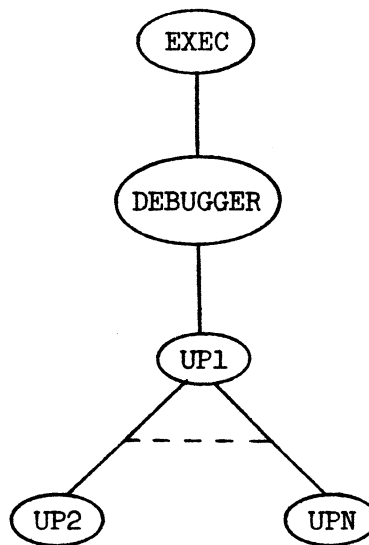
descriptor

actual descriptor

For the purpose of transmitting capabilities, the runtime is allocated a block of the process' capability segment. Capabilities received in messages are initially allocated there and then copied with the RECEIVE\CAPABILITY function.

11. Jobs

A job will be defined as an instance of a logged-in user. Thus, a job is initiated when a user logs in, and is terminated when he logs out. When a user logs in, his terminal will be controlled by a process created by the KERNEL. This process will execute code called the Executive (EXEC). Any computations performed by the user will either be performed by his EXEC process or by processes created by his EXEC. Thus, for example, the process structure of a job at a given point in time may appear as:



The responsibilities of the EXEC include the command processor, maintaining the job process structure, and providing a wide range of service functions. Furthermore, the EXEC will be responsible for managing the accounting information, some of which will be provided by the KERNEL. Later sections of these notes will deal with these issues in more detail.

12. Synchronization of access to shared objects

It is desirable that the operating system provide a mechanism for synchronizing the use of shared objects. For example, this is commonly helpful to coordinate access to files by different processes. Due to the complexity of implementation of a mechanism which enforces cooperative synchronization, only a voluntary mechanism is supplied. Thus, access to a shared object will be synchronized only if all processes accessing a shared file execute appropriate synchronization operations. To accommodate those situations where processes terminate abnormally (e.g., result of errors), an operation will be supplied which will release all synchronization requests for any objects for that process. If the EXEC executes such an operation for every destroyed process, then synchronization requests will not indefinitely cause shared objects to be reserved.

There is no reason why synchronization of objects must be performed by the KERNEL. Thus, it will either be performed by the EXEC (i.e., by the EXEC processes cooperatively) or by a separate process called the Synchronizer. In either case, the specific operations performed would include:

- request\shared \access (<object capability>)
- request\exclusive\access (<object capability>)
- cancel\all\requests (<process capability>)

Possible other operations would permit reading the status of objects and/or processes, and also the possibility of specifying for a shared request which other types of shared requests are compatible, on the basis of capability option bits.

For the purpose of encouraging the use of synchronization of accesses to files, the runtime primitives for directory lookups will include options for specifying shared or exclusive synchronization request.

13. Segment data structures

All segments will reside on the disk. A hash table which will also reside on the disk will have an entry for every segment. The entries will be hashed by the segment unique name. Each entry will contain the segment unique name, the disk address where the segment begins, and the segment length. This table is called the Disk Segment Table (DST).

The Core Window Table (CWT) was described in section 5. An intermediate table called the Active Segment Table (AST) will reside in core. This table will contain copies of entries from DST for segments considered "active." Entries will be copied into AST whenever it has been necessary to access that entry on DST. Entries in AST, a fixed sized table, will be deleted on a heuristic basis.

The AST will be used to determine the disk address for a desired window when that window is not in the CWT. Use of the AST will reduce the number of disk accesses to the DST.

For detailed information on allocation of segments on disk, allocation of windows in core, and algorithms to determine when to read in or write out windows, see later sections.