DESIGN AND IMPLEMENTATION OF AN EXAMPLE OPERATING SYSTEM

by

JOHN DANIEL DeTREVILLE

S.B., University of South Carolina
(1970)

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1972

Signature Redacted

Signature of Author . . . . . .
Department of Electrical Engineering, May 17, 1972

Signature Redacted

Certified by . . . . . Thesis Supervisor

Signature Redacted

Accepted by . . . . . .
Chairman, Departmental Committee on Graduate Students

DESIGN AND IMPLEMENTATION
OF AN EXAMPLE OPERATING SYSTEM

by

John Daniel DeTreville

Submitted to the Department of Electrical Engineering on May 17, 1972, in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

In software courses dealing with the principles of design of operating systems for digital computers, it is useful to be able to give the students a case-study of a specially-designed example operating system. The design and implementation of such an example operating system is presented in this thesis, along with guidelines for its use in a classroom environment.

The body of this thesis, then, is concerned with the details of the example operating system as designed and as implemented on the IBM System/360. The system consists of two major sections: a nucleus, which performs the basic operations of process management, memory management, and device management, and a top-level supervisor, which utilizes the features provided by the nucleus. These sections are presented from various viewpoints, and then the details of the operation of their composite modules and routines are given in increasing level of detail. Effort is made in this presentation to split the functions of the system, and particularly of the nucleus, into parts along the same manner in which these functions are typically split in computer science courses.

These sections also serve as the primary documentation of the example operating system.

An appendix serves to illustrate the use of the example operating system in the classroom. General guidelines are presented, and then a number of typical homework assignments for the students, based on the workings of the example operating system, are outlined, along with information on their teaching value.

THESIS SUPERVISOR: John J. Donovan
TITLE: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENTS

Well, first off, I have to thank Professor John Donovan. He was the one who first suggested this topic to me; he has continued to offer helpful suggestions as supervisor.

Throughout the thesis work, Stuart Madnick worked closely with me, and was very understanding concerning delivery slippages for the system and for the documentation.

Fellow graduate students Jerry Johnson and Leonard Goodman were forced to sit through several expositions of the principles of the operating system and should be thanked for this.

Plus which, thanks to everyone else who's had a part in this thesis.

This thesis was edited on the Project MAC Multics time-sharing system; the final copy was produced using the RUNOFF program.

Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction of this document, in whole or in part, is permitted for any purpose of the United States government.

TABLE OF CONTENTS

CHAPTER 0

GOALS OF THE SYSTEM

The topic of this thesis is the design and implementation of an example operating system for use as a pedogogical aid in computer science courses. It is intended to use this system as an example in 6.802, a course taught at M.I.T. on the principles of operating systems. This system meets a large number of pedogogical goals.

It is a simple operating system, making it easy for the students to learn. This is in contrast to such other systems as OS/360 or Multics, whose main purpose is not pedogogical.

It is a small operating system, as operating systems go. In its present form, it occupies about 1500 cards (three-quarters of a box) of assembly-language code. It does not include language processors or utility programs, but instead implements a basic system nucleus which provides such features as multiprocessing, dynamic memory allocation, device management, etc., to which any top-level supervisor and associated programs could be easily fitted. These basic features are the ones of the most importance in a course on

operating systems.

This implementation does include a simple top-level supervisor to provide processing of the job streams. Thus, the operating system is complete, in that students could imagine it being used for some real-world application. (Such an application for this system would be controlling a large number of real-time devices, with additional background computing.)

Lastly, this operating system is explicitly designed in a modular manner, which is not often the case in real-world operating systems. In particular, the relevant sections for processor management, or memory management, or device management, can easily be separated from the rest of the system. This is important in a classroom environment since these functions are usually taught separately from each other. In real-world systems, some of this modularity is usually sacrificed for performance.

TECHNICAL ACKNOWLEDGEMENTS

Most of the features of this operating system were first introduced elsewhere, and are here taken from other papers in the field, or from other operating systems. The idea of a system implemented in various discrete levels (here, the nucleus and top-level) has been best expressed by Dijkstra (Dij 68), who also introduced the concept of a semaphore. The form of the message system for interprocess communication comes from Hansen (Han 70), who also elucidated the concept of a system nucleus. A great part of the terminology and concepts in process control were introduced by Saltzer (Sal 66) and implemented in Multics (Cor 65), from which several ideas in this area were taken. The view of the operating system as a group of resource management modules, and the specific breakdown in this respect, comes from Donovan (Don 72).

Additionally, several key points were taken from such operating systems as OS/360 (both MFT (IBM :2) and MVT (IBM :3)), CTSS (Cri 65) and IBSYS (IBM :1).

OVERVIEW OF THE SYSTEM

The example operating system is a multiprogramming system for the IBM System/360. It requires the standard instruction set, and both store and fetch protection. Any types of external devices can be supported, but currently standard system support is provided only for card readers and line printers (user programs can provide their own routines for non-standard devices, however).

The example operating system currently runs under the control of a System/360 simulator, which simulates an environment with 32K-bytes of storage, two card readers, and four line printers.

Memory allocation for user programs is in the form of variable-length partitions (although not relocatable; the System/360 hardware will not allow this). Users can specify the amount of storage required in terms of 2K-byte increments from 2K-bytes upwards. The system currently requires about 6K-bytes for its own routines; the rest of memory is available for user programs.

The system is capable of supporting multiple job streams coming in from different input devices (here,

readers), with the output being directed to different output devices (here, printers). Each job stream consists of a number of jobs, stacked in order.

A job consists of a $JOB card, followed by an object deck, followed by optional data. The format of the $JOB card is:

```
$JOB,core,name=devtype,name=devtype,...
```

as in:

```
$JOB,8K,FILEA=IN,FILEB=OUT
```

The "core" field gives the amount of core required for the job (in the example, 8K). The "name=devtype" field can be repeated any number of times (including zero). The "name" gives the name by which the user program can reference this device (these names are up to the user, and very flexible, device-independent referencing of devices is provided) while the "devtype" tells the type of device to be assigned. There are currently three possibilities for this field.

The type "IN" specifies the system input unit (here, the card reader). The type "OUT" specifies the system output unit (here, the line printer). The type EXCP indicates a non-standard device, for which the user will supply his own handler routine. Currently, the $JOB card can contain at

most one reference to IN, one reference to OUT, and one reference to EXCP.

The object deck, which immediately follows the $JOB card, has the same format as the standard OS/360 object deck. External references are not allowed, however. The object deck gives the text to be loaded into the user's partition, and specifies the starting point; relocation is performed during loading.

Following the object deck is the card input data to the user program, if there is a reference to "IN" on the $JOB card.

The user's program may specify parallel processing to take place within it, and there are features to control this, along with flexible facilities for communication between different parallel execution paths of the program. The system automatically schedules the various user programs in the system, and their parallelly-executing sections, in such a way as to tend to maximize the use of the CPU and the data channels.

MODULES OF THE SYSTEM

As described in chapter 0, the example operating system is implemented in two levels: a nucleus and a top-level supervisor. However, for clarity, we can split the system down more finely to view it as being implemented in a number of layers. One property of this layered construction is that each successive layer, from the bottom up, depends only on the existence of those layers existing below it, and does not directly depend on higher layers (this feature being of importance for pedogogical reasons).

The five layers (or "modules") of the example operating system are:

Process Management, lower module	(lowest)
Memory Management module	
Process Management, upper module	
Device Management	
Supervisor Process module	(highest)

(where, of course, the user program depends on all of these). A few features of this breakdown are of note.

First, the functions of process management have been split into a lower module and an upper module, one on either side of memory management. This is because certain parts of

process management (those in the upper module) depend on memory management routines, but memory management itself depends on certain process management routines, thus placing these in a lower module below memory management.

As shown, the supervisor process module depends on process management (both modules), memory management, and device management. The dichotomy between the nucleus and the top-level supervisor is not as clear in this breakdown as when it was presented in chapter 0; however, the modules below the supervisor process module may be viewed as the nucleus, and the supervisor process module itself as the currently-implemented supervisor.

Finally, it will be seen from the discussion that follows that the supervisor process module and the user program itself operate in very nearly the same environment. This is an unusual aspect of this operating system, and comes from the nucleus vs. top-level supervisor breakdown.

PROCESS MANAGEMENT,

LOWER MODULE

This module of process management provides processor multiplexing, and additionally provides basic primitive operations for inter-process synchronization. This module may be viewed as the "multiprogramming support" module.

This module provides the basic support for processes. A process may be viewed as the execution of a program within certain constraints (e.g., within a certain area of memory); for a more precise definition, see Saltzer (Sal 66). Multiprogramming allows these processes to run independently, in parallel with each other.

The "traffic controller" section of this module schedules and runs processes which are eligible to run (a process in the system is usually eligible to run unless it is waiting for the completion of some external event (e.g., it might be waiting for the completion of an input/output operation; alternatively, it might be in the action of synchronizing with other processes: see below for the relevant primitives). Processes are schedled by the traffic controller in a round-robin fashion (see the writeups on the traffic controller in chapter 3 for more details on this)

and run until they become temporarily ineligible to run, as described above, or until a certain amount of time passes (in this system, fifty milliseconds), called the quantum of time, after which the process is eligible to continue running later.

In either case, the traffic controller then selects another process and causes it to begins running.

This module also provides basic primitives for the synchronization of processes. These primitives are called the "P" and "V" operations (as named by Dijkstra (Dij 68)). Their operation is as follows.

Both the "P" and "V" operations act on a "semaphore", which has an associated integer value. The "P" operation has two possible effects. If the value of the semaphore is greater than zero, it causes one to besubtracted from that value. If the value of the semaphore is zero or less, the process issuing the "P" operation begins to wait for a signal from another process (thus becoming temporarily ineligible). This signalling will be caused by another process executing a "V" operation on the semaphore.

The "V" operation also follows one of two possible courses of action. If there are no processes currently waiting as a result of performing a "P" operation on this semaphore, it adds one to the value of the semaphore. If,

however, there are processes waiting, a signal is sent to one of then to stop waiting.

The "P" and "V" operations, together with semaphores, have several useful applications. For example, a semaphore with initial value one can be associated with some data base to serve as a lock on that data base. If all processes accessing that data base perform a "P" operation on the associated semaphore before acessing the data base, and perform "V" operations on that semaphore afterwards, it may be seen that only one process can ever access the data base at any one time, thus insuring the integrity of that data base.

MEMORY MANAGEMENT MODULE

This module performs the operations necessary for dynamic allocation and freeing of memory. It provides routines which will on request allocate a block of memory of a given size and of a given address-alignment (e.g., on a double-word boundary), or which will on request free a block of memory of a given size at a given location. To accomplish this, the module keeps a list of "free" storage, taking from or adding to this list as requests are processed. Communication with this module is by means of explicit calls.

The memory management module uses the process management lower module in the following respect. If there should ever come a request to allocate memory which cannot be satisfied (i.e., there is currently no such contiguous block of memory free) the allocation routine must wait until some other process frees some memory. The basic synchronization primitives provided by the process management lower module are used for this (for more detail, see the appropriate writeups in the next chapter).

PROCESS MANAGEMENT,
UPPER MODULE

This module provides routines for the control of processes (e.g., their creation and deletion within the system), and also provides a sophisticated inter-process communication routine with buffered messages. Communication with this module is in the form of explicit calls.

Firstly, this module provides routines for the control of processes. Most importantly, there are calls by which processes of a given name may be created and started to run at a given location, with certain register contents passed to it as arguments, and there are calls by which a process within the system may be stopped and destroyed. At this level, of course, it is entirely possible to ignore the existence of the traffic controller, in that we may view all processes, say, as continually running; this view, if taken by this module, would be perfectly consistent.

This module also provides a method by which messages may be sent between processes. There is a call saying to send, to a process of a certain name, the k-character message "such-and-such"; there is also a call saying to read the next message waiting to be read, which returns the text

of the message and the name of the sender. Messages may be of arbitrary length and any number of messages may be waiting to be read by a process.

As may be seen from the above, this is the level at which we introduce the concept of the "name" of a process. Names are chosen at process-creation time and are used to reference processes within this upper module. This module also introduces the concept of a group (see chapter 3) which is the set of processes associated with a job within the system.

This upper module depends directly on both the process management lower module and the memory management module. It used the "P" and "V" primitives to provide synchronization between the sender and the receiver in message processing. Also, since this module runs in the process of its caller, there is an implicit dependency on the traffic controller. The memory management module is called to allocate or free space needed to store system information concerning processes, of to provide temporary buffers for message processing.

DEVICE MANAGEMENT MODULE

This module provides the routines necessary to issue the appropriate input/output commands to external devices. Unlike those before, but like all higher levels (the supervisor process module and the user program), this module runs in a separate process (in this case, one per (group, device) pair). Communication with this process is by messages sent to it, and the status of the result is returned via messages which it sends back.

The routine, when created, is provided with sufficient information to issue input/output instructions. Its major purpose is to issue these and then, after an interrupt occurs, interpret the status information available for the construction of a return message.

This module depends directly on the process management lower module in that it uses semaphores as locks against two processes simultaneously attempting to access the same device (here, the semaphore is a lock on a block of information held about the device) and to wait for an interrupt (by performing a "P" operation on a semaphore with initial value zero, in order to wait until another routine in this module, which is entered upon interrupts, performs a

"V" operation on this semaphore).

SUPERVISOR PROCESS MODULE

This module serves as the top-level supervisor. It uses the features provided by the previous modules to create an interface for processing of user jobs. This module runs in a number of separate processes, one per job-stream (thus one per group).

Its purpose is to, for each job in sequence, allocate a partition for it to run in (the size being specified on the $JOB card) and create and start processes for interface to the device module (see chapter 3 for more details). It then loads the user's deck into the partition and creates and starts a process in that partition. At this point, the supervisor process can stop running for a moment and wait for a message signalling completion to come from the user program. When completion is signalled, the supervisor process cleans up (i.e., destroys any processes it might have created for this job, and frees the partition allocated), and goes on to the next job.

THE USER PROGRAM

The user program runs in the partition of memory allocated for it by the supervisor process. There is at first one process in which it runs, but there is the ability to create others to run in parallel. When the job is completed, there are primitives provided by which the user program can signal either successful completion.

The user program can, of course, use the facilities of any of the other modules in the system.

CHAPTER 3

DETAILS OF THE OPERATION
OF THE EXAMPLE OPERATING SYSTEM

This chapter goes into detail on the organization of the data bases of the system and the operation of the modules. First, we will examine the databases.

DATABASES

This section presents all the major data bases present in the system, explaining their structure and their use.

The descriptions here consist of a PL/I-style structure declaration, followed by details on each of the fields.

PROCESS CONTROL BLOCK

The Process Control Block (PCB) stores the information associated with a process. There is one PCB for every process in the system. The PCB is defined approximately by:

```
declare
1 pcb based (pcbptr) aligned,
2 name char (8),
2 next_pcb_this_group ptr,
2 last_pcb_this_group ptr,
2 next_pcb_all ptr,
2 last_pcb_all ptr,
2 stopped bit (1),
2 blocked bit (1),
2 interrupt_save_area like (save_area),
2 message_semaphore_common like (semaphore),
2 message_semaphore_receiver like (semaphore),
2 first_message ptr,
2 next_semaphore_waiter ptr,
2 in_smc bit(1),
2 stop_waiting bit (1),
2 stopper_semaphore like (semaphore),
2 stoppee_semaphore like (semaphore),
2 fault_save_area like (save_area),
2 memory_allocator_save_area like (save_area),
2 memory_freer_save_area like (save_area);
```

where the fields are as follows.

"name": this field contains an eight character field giving the name for this process. A name is supplied for a process by the process that creates it; these names are used to reference processes in calls to the process management upper module routines.

"next pcb this group": is a pointer to the next PCB in this group. The set of processes in the system is divided into a number of mutually-exclusive subsets known as groups; the processes in each group are chained together in a circular list. This pointer is the forward pointer in that circular list. All names within a group are unique, and naming of processes is always relative to a group.

"last pcb this group": the backward pointer for the circular list of the PCB's in this group.

"next pcb all": in addition to their being chained together in the above-mentioned group-oriented fashion, all PCB's in the system are independently linked into a single circular list. This is for the purposes of the process management lower module (in particular, the traffic controller) which is not concerned with groups (these being

an upper-module concept within process management). This pointer is the forward pointer within this chain.

"last pcb all": the backward pointer for the chain of all PCB's.

"stopped": this bit is zero when the associated process is not stopped, and one when it is. A process is not considered runnable by the traffic controller if it is stopped. When a process is first created, it is in the stopped state, and may be started by some other process performing a "start process" primitive for it. A non-stopped process may be stopped by another process performing a "stop process" primitive for it, usually as a prelude to destroying it.

"blocked": this bit is zero when the associated process is not blocked, and one when it is. A process which is blocked is not considered runnable by the traffic controller. A process is normally not blocked, but can go blocked if it performs a "P" operation on a semaphore with non-positive value. It will be waked up (made to go non-blocked) whenever some other process performs a "V" operation on the semaphore (for a more precise statement of how this is performed, see the writeups on semaphore structure and on the functioning of the "P" and "V" primitives).

"interrupt save area": a save area. Its format is given below in the save_area section, and its purpose is described below in the routines section.

"message semaphore common": this is a semaphore with initial value of one. Whenever a routine is entered to send a message to this process or to read a message sent to this process, this semaphore is used as a lock on the message chain pointed to by first_message, during the period when the message chain is being searched or modified.

"message semaphore receiver": this semaphore is used to regulate a process receiving messages. Its initial value is 0. Whenever a message is sent to this process, the sending process performs a "V" operation on this semaphore, thus adding one to its value. Whenever an attempt is being made to read a message sent to this process, there is first a "P" operation performed on this semaphore. Thus, if there are no messages currently readable, the "read message" routine will wait until there are.

"first message": this is a pointer to the first message waiting to be read by this process. See the writeup on message format for information on the structure of the message chain. If there are no messages readable by this process (i.e., if the value of the "message semaphore receiver" is zero), the value of this pointer is

meaningless.

"next semaphore waiter": all processes currently waiting on a single semaphore are chained together in a linear list. The head of this list is pointed to by a field in the semaphore, and the PCB's in the list are linked together by this field. The length of the list (see the writeup on semaphore format) is taken to be the maximum of zero, and the negative of the value as stored in the semaphore.

"in smc": this bit is zero if the process is not in an smc section, and one if it is. The term "smc" stands for "system must complete". If a process is in an smc section, an attempt to stop that process will wait until that process leaves the smc section. There are primitives to enter and leave an smc section.

"stop waiting": this bit is zero if there is no stop request waiting for this process for when it leaves any smc section it may be in, and one if there is. This bit is set by the "stop process" primitive.

"stopper semaphore: this semaphore is used to wait on stopping a process in an smc section. The process attempting to do the stopping performs a "P" operation on this semaphore, which has an initial value of zero, whenever

it notices that the process that it is trying to stop has the "in smc" bit set on. When the process to be stopped attempts to execute a "leave smc" section primitive and notices that the "stop waiting" bit is on, it performs a "V" operation on this semaphore, and then a "P" operation on the "stoppee semaphore" (described next).

"stoppee semaphore": this semaphore, which has initial value of zero, is used to stop a process which has had a stop postponed until it left an smc section. It performs a "P" operation on this semaphore, thus blocking itself until the stopping process can execute another stop request.

"fault save area": a save area. Its format is given below in the save_area section, and its purpose is described below in the routines section.

"memory allocator save area": a save area. Its format is given below in the save_area section, and its urpose is described below in the routines section.

"memory freer save area": a save area. Its format is given below in the save_area section, and its purpose is described below in the routines section.

RUNNING

This is a pointer to the PCB of the process currently being run. It is set by the traffic controller, and is used whenever it is necessary to access the PCB for "this" (the current) process.

NEXTTRY

This is a pointer to the PCB of the process that the traffic controller will next try to schedule after it is next entered. This pointer is set by the traffic controller to be the same as the "next pcb all" pointer in the PCB pointed to be RUNNING, but may be modified by the execution of a V operation (see the writeup on the V operation for more deatils).

NEXTTRY_MODIFIED

This is a bit which is zero if NEXTTRY has not been modified since being set by the traffic controller, and one if it has.

SAVE AREA

Whenever system routines need save areas for storing the conditions appertaining when it was entered, so that they can be restored upon exit, it uses one of the four save areas in the PCB. These save areas look like:

```
declare
1 save_area based (save_area_ptr),
2 old_psw like (psw),
2 old_registers(0:15) like (register),
2 next_save_area ptr,
2 temporary(3) bit(32);
```

where the fields are used as follows:

"old psw": this is the old psw upon entry to the routine.

"old registers": these are the old general-purpose registers upon entry to the routine.

"next save area": this pointer is currently unused. See Appendix A, Minor Flaws section, as to why.

"temporary": this is a temporary storage area of three words. It is used mainly to contruct argument lists, which must be held in storage, for calling other routines. If a

routine ever needs more than three words of temporaries, it uses these three words to contruct a call to the memory allocator.

SEMAPHORES

Semaphores are used for basic, low-level synchronization of processes. They can be accessed by "P" or "V" operations, or their value field may be examined directly. Their format is as follows:

```
declare
1 semaphore based (semaphore_ptr),
2 value fixed (31,0),
2 first_waiter ptr;
```

where the fields are used as follows:

"value": is the value of the semaphore. A "P" operation always subtracts one from this value, and a "V" operation always adds one. The initial value of the semaphore must be non-negative.

"first waiter": for semaphores with negative values, this is the pointer to the first in a list of PCB's chained together by their "next semaphore waiter" pointers, where the length of the list is the magnitude of the value. A "P" operation on a semaphore with non-positive value places the

PCB for its process at the end of the list, whereas a "V" operation on a semaphore with negative value modifies this pointer to in essence take the first PCB off the list (after which it receives a wake-up): thus we have a FIFO queue.

FREE STORAGE BLOCK

All of free storage (storage which can be given to allocate requests) is chained together in free storage blocks. All free storage blocks are chained together in order of increasing size. No two free storage blocks are ever adjacent in memory: rather, these two would be collapsed together into a single larger block.

Whenever an allocation is requested (see the routine writeups for more information) a block is unlinked from the free storage list, and the remainder or remainders is linked back onto the list. When a block is freed, it is linked back onto the list, after collapsing any adjacent blocks.

The format of a free storage block (FSB) is:

```
declare
1 free_storage_block based (fsb_ptr),
2 next ptr,
2 size fixed (31,0),
2 unused (free_storage_block.size-8) char(1);
```

where:

"next": a pointer to the next FSB in the ascending-size-order chain of FSB's. For the last FSB in the chain, this field contains all zeros (null).

"size": this field contains the size of this FSB. The size is stored in bytes, but all FSB's contain an integral number of words (since all requests to the memory management module (all calls are made by the system) specify that the allocated area is to be on a doubleword boundary).

"unused": this field fills out the remainder of the block and contains nothing of importance.

FREE STORAGE BLOCK POINTER

This pointer (FSBPTR) points to the first free storage block in the ascending-order-of-size chain. If there are no blocks in the chain, this pointer contains all zeros (null).

FREE STORAGE BLOCK SEMAPHORE

This semaphore, with initial value one, controls access to the free storage list by serving as a lock. Any memory management routine, upon entering a section where it examines or modifies the free storage list, does a "P" operation on this semaphore, thus ensuring the integrity of this data base, and does a "V" after it stops using this data base.

MEMORY SEMAPHORE

This semaphore, with initial value zero, controls waiting for memory. If a process attempts to allocate memory but is unable, it performs a "P" operation on this semaphore, thus blocking itself. When it reawakens, it reattempts the allocation, again possibly blocking itself, until the request can be satisfied.

A sequence of "V" operations is performed on this semaphore whenever a block of memory is freed. The number of "V" operations performed is equal to the number of processes waiting on the semaphore at that time.

ROUTINES

We now discuss the details of the composite routines of the system. As Chapter 2 divided the realm of the example operating system into the modules of process management (with a lower module and an upper module), memory management, device management, the top-level supervisor, and user programs, so we can here divide the set of system routines. Here, though, the distinction will be on how they gain control. The following are brief expositions of this division. A detailed explanation of the operation of each of these groups follows in subsequent sections.

The traffic controller, which forms part of the lower module of process management, constitutes the first case. It gains control whenever a timer runout trap is detected, whenever a "P" operation is performed on a semaphore with non-positive value, or whenever control is specifically relinquished to it (this last is a special case: see the following documentation for its use).

The synchronization primitives "P" and "V", which form the remainder of the lower module of process management, together with all of memory management and all of the upper module of process management, form the second case. ? hese

routines gain control by virtue of a specific call (here, by means of the SVC instruction). They are passed an argument list and perform requested functions.

Device management is split between two methods of gaining control. The device-handling routines themselves reside in a special process, one per (group, device) pair, originally created by the top-level supervisor, and communication with them is by means of the message primitives provided by the upper module of process control.

One function, though, performed by process management is the fielding and handling of input/output interrupts. The routines for doing this are invoked whenever such an interrupt occurs, run for a very short time in the process of whoever was running at the time of the interrupt, just long enough to store status information and wake up the process waiting for that interrupt, before returning to the processing of the interrupted process.

The supervisor process module runs in a special set of processes, one for each job stream. It is initially created and started at IPL (Initial Program Load) time, and passed arguments by the IPL routine concerning its operation.

Finally, the user's program runs in a process or processes of its own, being started by the top-level supervisor.

DETAILS OF THE TRAFFIC CONTROLLER

The traffic controller serves the purpose of creating a process-oriented environment. It runs "between processes", as it were.

The traffic controller performs the basic task of processor multiplexing, sharing the machine's CPU among the various processes runnable at any one time. A process is said to be runnable if both the "stopped" and "blocked" bits of the PCB are off. When a process is entered, it runs until its quantum of time is consumed (a timer runout trap is set to occur 50 ms. after a process is entered), it needs to wait for synchronization with another process (a "P" operation is performed on a semaphore with non-positive value), or it specifically relinquishes control to the traffic controller via a special call (this is an SVC call to routine XPER, for which see below; this entry is only used by the IPL routine, to start the system going, or by the input/output interrupt handler, in the case of needing to schedule a special process quickly for real-time response. See the appropriate sections for each of these).

All entries to the traffic controller transfer immediately to routine GETWORK. This routine runs with all

interrupts off, in key 0 (thus allowing it to access any part of memory). The old registers and PSW of the process just left are stored in the PCB's "interrupt save area".

Beginning with the process pointed to by cell NEXTTRY, it searches through all PCB's in the system, going through the pcb.next_pcb_all chain, finding the first PCB which is neither stopped or blocked. If there does exist such a one, the traffic controller resets cell RUNNING to point to that PCB, resets cell NEXTTRY to point to the next PCB in the "all" chain, sets the timer to interrupt in 50 milliseconds, and then proceeds to call a standard system service routine to reload the registers and PSW from the interrupt_save_area in the PCB pointed to by the new RUNNING, thus beginning to run this new process.

If all PCB's in the "all" chain are stopped or blocked, the traffic controller loads a PSW with location set to GETWORK, but with the wait state set on. As the section on input-output interrupts explains, when such an interrupt occurs, the old PSW's wait bit is set off before it is reloaded. Thus, the traffic controller hereby waits until an input-output interrupt occurs before it has another try at scheduling (since there can be, in this situation, no processes can be runnable until such an interrupt occurs,

although there is always one runnable immediately afterwards). The traffic controller utilizes the wait state instead of scheduling, say, an ever-present, ever-runnable process with an infinite loop in it, since (as chapter 1 relates) this system is really run under the control of a System/360 simulator; this simulator is vastly more efficient on simulating the wait state than it is for infinite loops running, for at total, with a simulated time of several seconds for even a small job stream.

REQUEST-DRIVEN ROUTINES

PROVIDING SYSTEM PRIMITIVES

This section includes those routines which are entered due to a request for a system primitive function. These are invoked by the SVC instruction which, on the System/360, provides for a one-byte operand; this byte is used to store an mnemonic for the operation to be performed. A list of the allowable mnemonics, their function, the routine which processes them, and the save area which they use (I stands for the interrupt save area, F stands for the fault save area, MA stands for the memory allocator save area, and MF stands for the memory freeer save area; note that these names as used here are archaic and no longer really relate to the function performed by the save area: Appendix A (Fixing Minor Flaws section) has further information on this).

(in process control, lower module)

P	the "P" synchronization primitive	rout. XP	(I)
V	the "V" synchronization primitive	rout. XV	(I)

(in memory management)

A	allocate a block of memory	rout. XA (MA)
F	free a block of memory	rout. XF (MA)
B	link a block onto the FSB chain	rout. XB (MF)

(in process management, upper module)

C	create process	rout. XC (F)
D	destroy process	rout. XD (F)
H	halt job and signal supervisor	rout. XH (F)
I	insert a PCB into the chain	rout. XI (I)
J	remove a PCB from the chain	rout. XJ (I)
N	find a PCB given its name	rout. XN (MF)
R	read message	rout. XR (F)
S	send message	rout. XS (F)
Y	start process	rout. XY (MF)
Z	stop process	rout. XZ (I)
.	enter traffic controller	rout. XPER(I)
!	enter smc section	rout. XEXC(I)
,	leave smc section	rout. XCOM(I)
?	abnormally terminate this job	rout. XQUE(I)

The save areas are so assigned that, in the course of modules calling on one another, no save area is ever overlaid. Note also that the calls which might cause the

traffic controller to be entered (namely XP and XPER) both use the I save area, the same as the traffic controller itself.

All of these routines run in key 0, allowing arbitrary memory references, overriding the protection mechanism. Routines XP, XV, XB, XI, XJ, XN, XZ, XPER, XEXC, XCOM, and XQUE operate with interrupts off, while the others operate with interrupts on. The advantage to having interrupts off is the assurance that no other processes can run while that routine is in operation.

All routines which operate with interrupts on (namely XA, XF, XB, XC, XD, XR, XS, XY, and XZ) call the XEXC routine upon entry and the XCOM routine upon exit. This is to assure that a process will not be destroyed when it has some important system lock set.

Only some of these routines are callable directly by the user. These are the XC, XD, XH, XN, XR, XS, XY, XZ, and XQUE routines; the rest are usable only by the system routines (there is a check for this on entry, done by testing if the key under which the caller was operating was 0, indicating a system procedure, or non-zero, indicating a user procedure. Only these routines listed, which are primitives in the process management upper module, are safe

for the user to call; allowing a direct call to the others could allow a breach of security.

When an SVC instruction is executed, there is an automatic transfer to a special SVC-handling routine. This routine looks up the mnemonic for the operation, stores the registers and old PSW in the appropriate save area, then transfers to the appropriate routine (the SVC handler operates under key 0). If the called routine needs to call another service routine, it again uses the SVC-style call.

Arguments are passed in an area pointed to by register 2. Any returned values are placed in this argument list. A return is effected by reloading the old registers and old PSW.

Following is a description of the purpose and functioning of the above-mentioned routines.

ROUTINE XP

This routine implements the "P" primitive. Upon entry, register 2 is assumed to point to a semaphore. The routine subtracts one from the value, and then returns if the result is non-negative.

If the result was positive, however, then the PCB for this process is inserted at the end of the list of processes waiting on this semaphore. See the writeups on the "first waiter" field in the semaphore, and the "next semaphore waiter" field in the PCB for more information. It then sets the "blocked" bit in the PCB to one, keeping the process from being scheduled to be run until a wakeup is sent to it by the XV routine (see below). Control then transfers directly to the traffic controller.

ROUTINE XV

This routine implements the "V" primitive. Upon entry, register 2 is assumed to point to a semaphore. The routine adds one to the value of that semaphore. If the value is now greater than zero, it returns.

If the value is zero or less, that means that there were processes waiting on the semaphore. The first is taken

from the list (see the references referenced in the XP writeup) and a wakeup is performed for that process, setting its "blocked" bit to a zero, allowing the process to be scheduled once again.

To allow the target process to be scheduled soon, the XV routine typically sets the NEXTTRY pointer used by the traffic controller to point to the target PCB, and sets the "nexttry modified" bit to a one. This is not done, however, when that bit is already a one. Thus, if one process wakes up a large number of others in the course of its execution, the first to be awakened is the first to be run after the current one.

The XV routine then returns.

ROUTINE XA

This routine serves to allocate a block of memory. On entry, register 2 is assumed to point to an argument list, which contains the size of the block desired, expressed in bytes, and supplied by the user, and then a power of two, indicating what type of boundary the allocation is desired on (for example, an argument of eight would indicate that doubleword alignment was desired), and then a fullword space

into which the routine returns the address of the allocated area.

The routine first does a "P" operation on the "fsb semaphore". It then goes through the free storage list until it finds a block large enough to contain the requested area. The routine calculates which parts of that block need to be relinked onto the free storage list (i.e., those which overlap on either side of the allocated area: the allocated area is selected as close to the beginning of the block as possible with the specified alignment, in order to minimize breakage) and performs this relinking using the "B" primitive. The address of the allocated area is inserted into its place in the argument list and the routine performs a "V" operation on the "fsb semaphore", and then returns.

If, however, there is no way to satisfy the request (i.e., the routine reaches the end of the free storage list without finding a suitable block) then the routine does a "V" operation on the "fsb semaphore", and then does a "P" operation on the "memory semaphore". As explained in the writeup for this semaphore, this has the effect of waiting until someone frees some memory. The routine, after waiting, returns to the beginning and re-attempts the allocation.

ROUTINE XB

This routine performs the function of linking a block of storage onto the free storage list, with the assumption that compacting of the new area with existing FSB's is not possible (thus XB can be called directly from routine XA). On entry, it assumes that register 2 points to an argument list, which contains the size of the area to be freed, measured in bytes, and the address of the area.

Since this routine is called only by those routines (XA and XF) that have already done a "P" on the "FSB semaphore", this routine does not need to worry about database interference from other processes.

The routine searches through the free storage list to find the point at which the new block should be inserted, and then performs the relinking, formatting the new block to look like a FSB. It then returns.

ROUTINE XF

This routine performs freeing of areas of storage, whenever compacting of the new area with current FSB's might be necessary. On entry, it assumes that register 2 points to an argument list, which contains the size of the area to be freed, measured in bytes, and the address of the area.

The routine first does a "P" operation on the "fsb semaphore", to lock the free storage list. It then searches through the free storage list to see if these are any FSB's defining a region of memory contiguous with the one being freed. If there are, they are removed from the free storage list, and the address and size of the block being freed are recomputed. When the end of the list is reached, the routine does a "B" operation, calling routine XB, to link the block onto the free storage list.

The routine then examines the value of the "memory semaphore" to determine how many processes are waiting for memory to be freed, and then performs that number of "V" operations on the "memory" semaphore (obviously, since the XB routine is called only the XA and XF routines, performing these wakeups here is sufficient). The routine then returns.

ROUTINE XC

This routine performs the "create process" function. On entry, register 2 is assumed to point to an argument list consisting solely of a name for the process to be created.

The routine first checks that the name is not already used in this group. If it is, an error routine is entered (calling routine XQUE). If not, however, the routine calls routine XA to allocate an area for the new PCB. The new PCB has the name stored into it, the stopped bit turned on, the blocked bit turned off, the semaphores set to their initial values (as noted in their writeups), and the "in smc" bit turned off. It then calls the XI routine to link this PCB into the two chains. The routine then returns.

ROUTINE XD

This routine performs the "destroy process" function. On entry, register 2 is assumed to point to an argument list consisting solely of a name for the process to be destroyed. The process to be destroyed must have been previously stopped using the "stop process" primitive (routine XZ).

The routine uses the XN routine to determine the address of the PCB for the process to be destroyed (if there is no such process, an error routine is entered, calling

routine XQUE). This process must be in the stopped state. A "P" operation is performed on the "message semaphore common", thus keeping other processes from sending messages. Then all messages waiting to be read by this process are gone through, and their storage freed. Finally, routine XJ is called to unlink the PCB from the two chains, and the storage for the PCB is freed (all freeing here is done by the XF routine), and this routine returns.

ROUTINE XH

This routine performs the "halt this job" function. It takes no argument list.

The sole processing performed by this routine is to send a message (via routine XS) to process "*IBSUP" (see the writeup on the supervisor process module) stating that the job should be halted now, and that normal conditions pervade, and then perform a "P" operation on a standard semaphore, used just for this purpose, that no "V" operation is ever performed on (this being done after the value of that semaphore is set to zero). Thus, the XH routine never returns.

ROUTINE XI

This routine performs the function of chaining a PCB into the two PCB chains. On entry, register 2 is assumed to point to a PCB.

The PCB is chained onto the all-PCB chain immediately following the PCB for the running process, and onto the this-group chain immediately following the PCB for the running process. The routine then returns.

ROUTINE XJ

This routine performs the function of removing a PCB from the two PCB chains. On entry, register 2 is assumed to contain a pointer to the PCB.

The PCB is removed from the two chains by modifying the links. This routine does not free the storage; that is left up to the caller. It then returns.

ROUTINE XN

This routine serves the function of finding a PCB for a process, given the name of that process. On entry, register 2 points to an argument list consisting of a name and an area into which it returns a pointer to the PCB.

The routine looks along the "next pcb this group" chain, starting with the PCB for the running process, until it finds a PCB containing the desired name, upon which case it stores the pointer in the argument list and returns, or until it finds that there is none such, in which case it stores zeros in the argument list and returns.

ROUTINE XR

This routine performs the "read message" function. On entry, register 2 points to an argument list containing an area for the name of the sender, a number giving the size of the area supplied to receive the text, followed by that area.

The routine first performs a "P" operation on its "message semaphore receiver" semaphore, which serves to wait until a message has arrived. The routine then does a "P" operation on the "message semaphore common" semaphore, to lock the message chain (both of these semaphores are the ones in the current process's PCB, of course). Then, the first message is taken off the message list, the name of the sending process is found and placed into the argument list, and then the text is moved into the receiving area, being truncated or padded with blanks as necessary. Then the size field in the argument list is modified to contain the number of signifigant characters transmitted. Finally, the storage for the message in the message list is freed, and the routine does a "V" on "message common semaphore" and returns..

ROUTINE XS

This routine performs the "send message" function. On entry, register 2 points to an argument list containing the name of the destination process, a count for the number of characters in the text, followed by the text itself.

The routine first finds the PCB for the process by the given name: if there is none such, it enters an error

routine (by calling routine XQUE). If there is, the routine d s a "P" on the "message semaphore common" semaphore in that PCB, after allocating a region of memory large enough to hold the message block. Then that message block is moved down onto the end of the message list (the length of the message list being determined by the value of that PCB's "message semaphore receiver" semaphore), and filled with the sender's name, the count of the text, and the text itself. The routine then does a "V" operation on the "message semaphore common" semaphore, then a "V" on the "message semaphore receiver" semaphore, then returns.

ROUTINE XY

This routine serves to start a process that is in the "stopped" state. On entry, register 2 contains a pointer to an argument list consisting of the name of the process to be started, an address in memory at which the process should start running, and a pointer to a block of 16 words specifying the registers which should be loaded for that process in the beginning (this is the method of passing arguments to newly-created processes).

The routine first gets a pointer to the PCB of the process of the given name (if there is none such, an error

routine is entered, calling routine XQUE). It then stores in that PCB's "interrupt save area" the registers specified, and a PSW with the address specified (and with the remainder of the PSW identical to the old PSW existing when this routine was called). The "stopped" bit in the PCB is then turned off, and the routine returns.

ROUTINE XZ

This routine performs the "stop process" operation. On entry, register 2 points to an argument list, consisting solely of the name of the process to be stopped.

The routine first gets a pointer to the PCB of the process of the given name (if there is none such, an error routine is entered, calling routine XQUE). If that PCB's "in smc" bit is off, the routine simple turns the "stopped" bit on and returns.

If, however, the "in smc" bit is on, then the "stop waiting" bit is turned on, and a "P" operation is performed on the "stopper semaphore". When the process next begins to run (see the writeups on these data bases), the process to be stopped has left the smc section, and the stop can be performed normally.

There is one exception to the above. It is not allowed for a process whose name does not begin with an asterisk (this lack of an asterisk being used by the supervisor process module to indicate user processes) to stop processes whose names do begin with asterisks (the presence of an asterisk similarly being used to indicate system processes).

ROUTINE XPER

This routine serves to transfer to the traffic controller. It simply causes a transfer to routine GETWORK.

ROUTINE XEXC

This routine serves to notify that an smc section is about to be entered. The routine sets the "in smc" bit in its PCB on, and then returns.

ROUTINE XCOM

This routine serves to notify that an smc section is being left. It turns the "in smc" bit to zero. Then, if the

"stop waiting"bit in the PCB is off, it returns.

If, however, the "stop waiting" bit is one, then it is reset to zero, a "V" operation is performed on the "stopper semaphore", and a "P" operation is performed on the "stoppee semaphore" (see the writeups of these data bases for more information).

ROUTINE XQUE

This routine serves to abnormally terminate a job. It operates the same way that the XH routine operates (see above) but with a message that an error has occurred.

DEVICE-HANDLING PROCESSES

To provide for the control of external devices, there is a special process for each (job, device) pair. This process is created by the supervisor process, and passed arguments (through registers, when it is started) giving the address of the external device which it controls. It is started running in a routine for handling its type of device, and receives requests in the form of messages; it sends messages back when the requested input/output operation has been completed, these messages containing information on the result of the request. There are three such routines: one for reader, printer, and user-supplied handler-interface.

READER HANDLER

This routine controls card readers. When first entered, in its process, it has the unit control block address in register 3, and the protection key to be used held in register 4. The routine then goes into a loop reading messages. If the messages are not of the form "READxxxx", where xxxx represents a four-byte binary storage address, the message is rejected and another message is read.

If, however, the message is correct, the routine performs a "P" operation on the "user semaphore" in the UCB, to keep other processes from using the device for the iterim. The routine then constructs a channel control block (see the channel command block writeup for its special format). The operation specified in the CCW is a "read" request, with the address as specified in the message received. The count is eighty characters, and all flags controlling chaining, special interrupts, etc. set to zero. Then a CAW is constructed, with the address of the CCW, and the key being the key passed originally. Then a "P" operation is done on the "CAW semaphore", the machine's location for the CAW is loaded, the "start i/o" instruction for the appropriate reader is issued, and a "V" operation is done on the "caw semaphore".

Then a "P" operation is done on "wait semaphore" on the UCB, thus waiting until an interrupt occurs. The status as stored by the interrupt routine is then examined. If it indicates successful or unsuccessful completion of the request, the process examines the card, as described below. If, however, the operation has not completed, the process goes back to wait some more.

The process now examines the process name of its invoker. If the name starts with an asterisk, it is assumed to be part of the supervisor process module (see the section

on the supervisor process module for naming conventions) and a message reading either "YES" or "NO", depending on the success of the request, is returned. However, if the name of the invoking process does not start with an asterisk, it is assumed to be a user process, and the situation is more complicated.

First, if the card just read in was a $JOB card, this indicates the end of the data for this program. Thus, a "NO" message is sent back to the caller, after the card is saved in a special area and the user's copy is zero'd out. Then the process enters a loop wherein it answers all user requests with a "NO" until it gets a system request, at which point it stores the card in the specified area and returns the message "YES".

Otherwise, the message "YES" or "NO" is returned as for a ssystem request.

PRINTER HANDLER

This routine handles output for printers. Its operation is directly analogous to that of the reader handler. Messages are of the form "PRINxxxx", where xxxx represents the address to be printed from.

There is no data checking based on process name.

EXCP HANDLER

This routine handles the interface for those devices where the user wishes to provide his own input/output routine. The method of initialization and operation is similar, but the messages sent to it are of the form "EXCPxxxxcccccccc", where xxxx represents the unit address and cccccccc represents the channel command word to be used. There is a UCB for EXCP devices (one per group) but it does not contain a unit address.

When an interrupt occurs for an EXCP device, the EXCP interface returns a message of the form "ssssssss", where ssssssss is the status returned by the device and the channel. (Thus, the routine having called the EXCP process will be the next to be scheduled, since this is the only "V"-causing operation performed here. This fact, coupled with the fact that EXCP UCB's have the "fast processing required" bit on, provides for very fast processing of EXCP interrupts.)

INPUT/OUTPUT INTERRUPTS

The io_interrupt routine performs handling of the input/output interrupts occurring after an input/output request. The routine operates (essentially) in whatever process was running when the interrupt occurred. The routine performs a small number of operations, and then returns to the normal processing of that process.

First, it finds the UCB for the device causing the interrupt (see the writeup on Channel Command Blocks for how this is done). It then OR's the new status information with the status information as stored in that UCB (the structure of the System/360 is such that this is meaningful) and then performs a "V" operation on the "wait semaphore" in that UCB. Normally, the routine returns at this point by reloading the old registers and old PSW (the old PSW has been modified by this point to turn off the wait bit if it might be on: for the signifigance of this, see the Traffic Controller writeup).

If, however, the "fast processing required" bit in the UCB is on, the routine used the XPER routine to transfer into the traffic controller to run a new process (see the writeups for the "wait semaphore" and the "fast processing required" bit, along with the writeup on the XV routine, for

more information). Since the NEXTTRY field has usually been modified to point to the PCB for the device handler, this does indeed provide fast processing.

DETAILS OF THE SUPERVISOR PROCESS

The supervisor process, which serves in this system in the general capacity of the top-level supervisor, serves to provide processing of the job stream, giving structure to the system as the interface between the user jobs and the nucleus. There is one supervisor process per job-stream, created by the IPL routine (see below). Since these are created in separate groups, there is no communication between them; they are invisible to each other.

The supervisor processes, as was mentioned above, are created by the IPL routine. This is the routine that is entered upon an IPL (Initial Program Load). The IPL routine runs free of the rest of the system; most particularly, there is no point in the IPL routine, until the very end, at which the traffic controller can be logically entered. (Thus, there is a flag set such that if the traffic controller does get entered during this period, before the IPL routine makes an explicit transfer to it, the system will stop cold. Such a transfer could be caused by too little core, for example.)

The IPL routine first sets up a PCB for itself, in a permanently-allocated area, and sets pointers RUNNING and

NEXTTRY to point to that PCB, as well as all the chains within the PCB. Next, all available memory (not used by the system) is placed on the free list, and the protection keys associated with all of memory are set to zero.

Other PCB's are now created, one for each supervisor process, and each in a group by itself (thus the "create process" primitive cannot be directly used here); each process is named "*IBSUP". Then there is a call to "start process" for each of these processes, starting them in the Job Stream Processor routine, with register 3 pointing to an argument list containing a pointer to the UCB for the reader and a pointer to the UCB for the printer for this job stream (all UCB's are stored in a permanently-assigned part of memory), and register 4 containing the protection key which should be used for user programs in this job stream. Then the IPL routine modifies its PCB to read "stopped", and only then does it transfer (via routine XPER) to the traffic controller, to begin running the supervisor processes.

Upon entry, the supervisor processes first create and start processes called "*IN" and "*OUT", passing them a pointer to their UCB's in register 3, and their protection key in register 4. Then it begins to sequentially process each job in turn.

Upon reading a $JOB card, the first card of each job, the supervisor process first sends it to the printer. It then determines the amount of memory required and then allocates that amount, lying on a 2K boundary (because of the protection hardware), and sets the key associated with that area to the one to be assigned to the user program (as sent by the IPL routine). It then creates a process called USERPROG, which will become the process that the user program first begins to run in.

The process then scans the device assignment fields, these fields being of the form name=devtype. (Note that "name" cannot begin with an asterisk ("*"): see below.) For each of these, it creates a process called "name" and starts that process in an interface routine for that "devtype". For "devtype" either IN or OUT, the routine used is one that reads messages from the user process, sends them unchanged to process "*IN" or "*OUT", as appropriate, waits for a reply from that process, and then sends that reply to the original calling process. For "devtype" being EXCP, the interface routine is the one described earlier in this chapter under the Device Handler heading.

The supervisor module then begins to read the user's object deck into his partition of core. Relocation is performed as necessary. When this is completed, the process

USERPROG's PCB has its blocked bit temporarily turned on. Then the supervisor process starts the user process at the specified location, and then modifies the PSW so that it will run in user mode under the specified key. Then the blocked bit is turned off and the user process can start to run.

At this point, the supervisor process starts to wait for a message to be returned. This can be either a "success" or a "failure" message. In either case, the content of the message is printed on the printer, and the supervisor process destroys all of the processes created for or by the user job, frees the partition of memory allocated, and goes to the next job.

Whenever there are no more jobs to be run, the supervisor process enters an infinite loop reading messages (which it will never get).

DETAILS OF USER PROGRAMS

User programs run in a process or process of their own. They start out, as was mentioned above, in one process called USERPROG. However, more processes may be created; free creation of subsidiary processes is encouraged. User processes cannot have names starting with an asterisk ("*"), nor can they destroy a process whose name does begin with an asterisk. Thus, the supervisor is protected against the user.

There are two routines, XH and XQUE, which signal success and failure, respectively. XH is usually called by the program, wheras XQUE is usually called by a system routine invoked by the user, upon detection of an error condition (XQUE can take an argument in the form of an English-language message). Both of these routines send a message to the supervisor process and then block themselves.

APPENDIX A

USE OF THE EXAMPLE OPERATING SYSTEM IN THE CLASSROOM ENVIRONMENT

This operating system was designed for its use in a course on the principles of design of operating systems. There are two categories of such use.

First, it can be presented, either whole or piecemeal, to the students for their examination as a case study. When learning about the generalized aspects of operating systems, it is useful be able to tie each of these down by seeing how a real operating system has handled such problems.

The second category of use, however, is the more important. After the students have studied the example operating system and become familiar with its workings, they can be assigned problems concerned with adding new features not included in the original version.

Thus, one assignment may be to add a facility for the detection and prevention of memory-allocation deadlocks (such a facility not currently being provided). It is much easier to have students add such features to an existing

operating system than to have them program, as it were, in a vacuum, since their additions can easily be checked out by running test cases which would cause, say, memory-allocation deadlocks, and noticing whether the desired actions are taken. Without an example operating system available, the analogue is to construct a special grading program which calls the students' programs as subroutines, providing them with a pseudo-environment similar to (but not identical to) the environment these programs would have embedded within a real system. Provision of this type of facility becomes quite difficult when there is of necessity a dependence on timing (in device management, for example) or on the behavior of the programs running on the system at the time (with respect to some aspects of memory management and other modules).

There has yet been little direct experience with this system in the classroom in this regard, so there is little information yet available on how best to do this. Some knowledge may be gotten from (Cor 63), a report on the CAP classroom-oriented assembler used in 6.251 at M.I.T. in the early sixties; certain aspects of this text are analogous with respect to assignments given to the students.

These assignments may be broken down into a few categories.

FIXING MINOR OPERATIONAL FLAWS PRESENT IN THE EXISTING SYSTEM

This category includes assignments dealing with having the students correct a minor flaw in the system, which is not really critical, but which possibly has tricky aspects.

Currently, after using the "stop process" primitive (routine XZ), the stopped process cannot be restarted by the "start process" primitive (routine XY), since the stopped process might be in the middle of the "leave smc section" primitive (routine XCOM) stopped waiting on a semaphore which no other process will ever access (at to why, see the appropriate documentation). An assignment to test for such a condition in the "start process" routine would test the students' basic understanding of the system and its data bases.

Currently, semaphores are not usable directly by user programs (although messages can be utilized to provide the same operational capability). This is because certain system information, a pointer to the PCB of the first process waiting on the semaphore, is stored in the semaphore itself. Therefore, if a semaphore could be stored in the user's partition, this pointer's integrity could not be guaranteed.

Thus, the system currently allows only system routines to call the "P" and "V" routines, and these calling routines pass only system semaphores, stored outside of any user's partition and accessible only by system routines, as arguments. The students could be assigned to provide a facility usable by the user's programs for semaphore-type synchronization. Basically, it would have to have an entry point for creating semaphores in a system area , at the user's request, and passing back a name for the created semaphore to be used in future pseudo-P and pseudo-V requests. The basic thing the students would get from this, apart from gaining a familiarity with P and V, would be an awareness of the problems of memory allocation, as implemented within this system, since full records will have to be kept of what name is associated with what semaphore, and also of what semaphores were allocated by the user's programs, so that these can be freed at the end of the job. This bookkeeping serves to get the student down into the details very quickly.

The system as now implemented performs no accounting. One assignment is for the students to modify the system to expect another field on the $JOB card, of the form USER:username, where username is the name of the user to which this work is to be charged. At the end of the job, an

accounting tailsheet is printed containing the username, the resources used, and the total cost. As an adjunct to this problem, the students could be given basic information on what a 370/155 and its peripheral devices cost per month, what paper supplies cost, what an operator costs per hour and what types of functions he must perform for each job, and then asked to develop various pricing schemes implementing various goals for the installation at which this pricing scheme would be implemented. The paper by Nielson (Nie 70) is a helpful collateral reading in this regard.

And then there are small bugs noticed in the system during its development but left in for possible problems for the student. These serve probably best in the warm-up category. As an example, it is possible that a timer interrupt might be pending when a process goes to the traffic controller because of, say, performing a "P" operation on a semaphore with value 0. When the next process is scheduled and begins running, it will get the timer runout trap right off, even though it was meant for the last process. This is perhaps not crucial but it is certainly inelegant: it tends to undermine the notion of what a process is. A simple test will correct this one.

Another inelegancy in the system is caused by the large

number (4) of save areas in the PCB; these save areas together take up the vast majority of the PCB's space. (A bit of history is perhaps in order here: during the development of the system it was decided that perhaps certain modules might neeed to call each other recursively, or at least in very complicated patterns. Thus, a facility whereby any system routine could allocate a block of automatic storage, in the PL/I style, was added. However, the memory allocator would be necessary to do this, thus it needed space for its save area in the PCB. The memory allocator called the memory freeer, which thus needed its own space in the PCB, since it couldn't allocate its own, being unable to call back on the allocator. The timer runout routine needed space, since a timer runout could occur at any time, even in the middle of memory management. And the routines themselves needed a save area, for use before they could allocate an automatic area. After the allocation of the automatic region, they would move their old save area into that region, freeing that part of the PCB up for other calls; this explains the "next save area" pointer in the save area, which is currently unused. The problem here is that this facility was no longer needed, but the scheme was so deeply imbedded that not one of the save areas could be removed from the PCB.) The student could possibly, with a certain amount of rethinking, devise a new method for save

areas which does not use up quirte so much memory.

These small "warm-up" problems are not terribly important as a whole, but have their pedogogical uses in getting the students used to working on the system.

MAJOR MODIFICATIONS TO EXISTING MODULES

This category includes those changes to the system which are only modifications to the now-present structure, but which have major impact to the system, and are important pedogogically.

Currently, if the processes running in the system all start to request memory to the extent of surpassing the ability of the memory allocator to provide it, and there is no process freeing memory, a deadlock will be reached. All processes in the system might well be waiting on other processes, either waiting for other processes to free memory, or waiting for processes which are themselves waiting for memory. This can seriously degrade system performance, if it does not stop the system cold. A student could be assigned one of two types of projects. One would be to detect that all processes in the system are currently blocked, and that no-one is runnable or will ever be runnable (because there are no input-output operations currently being performed). This test, which would be performed by the traffic controller as an extension to a similar test which it currently performs, would cause it to throw a randomly-selected job off the system. Here, the detection of a deadlock situation is easy, but the

modifications to the system will cause problems. Currently, the method to throw a job off the system has as part of it sending a message to the supervisor process in its group. The problem here is that sending a message requires memory to be allocated. The amount required here is just a few words, but would be very likely to be impossible to satisfy if we were ever to find ourselves in this position. A pedogogically more important problem is that we would have the traffic controller, the most basic part of the system, calling on the supervisor process, a section of the system very far removed from it. This is a definite problem from the viewpoint of modularity, and its resolution is non-obvious.

Another possible student assignment associated with memory-allocation deadlocks would be to detect partial deadlocks. It might be that with five job streams into the system, four of them are involved in a deadlock while the fifth has a long compute-bound program running, which has no current use for programs which use the memory allocator. When it completes its compute-bound section, it might free enough memory to allow the others to run, but it more likely will get caught in the same deadlock. An important, but intellectually difficult assignment would be to extend the traffic controller to detect this type of deadlock coming

on, and deal with it as in the first deadlock problem mentioned above (of course, all of the problems noted in reference to that problem still apply). The specifics of such detection would, of course, depend on the instructor; there is certainly no easy, general solution.

Another modification of similar complexity, and also dealing with deadlocks, would be to improve the device-allocation procedure. Currently, each job can request one reader and one printer, where which device is to be assigned is determined before the request is made. This is certainly not the most efficient manner of proceeding (particularly since there are twice as many printers present on the system on which we run as there are readers!) but is simple moreover completely avoids the problem of deadlocks. An assignment to the students to improve the situation would be a good one to test their knowledge of deadlocks. Moreover, a real improvement in system performance might be gained if, say, a reader used by a job were to be released whenever there were no more cards to be read for that job, thus allowing a then-created supervisor process to begin to process the next job (this would change the status of the supervisor process).

MAJOR ADDITIONS TO THE EXISTING SYSTEM

There is finally the category of major additions of new features to the system. These can, of course, take on any form. A few representative ones are considered below.

One assignment would be to add routines to the system to handle other types of devices, such as disk. This assignment in itself might be of small teaching value, but it lays the way for further assignments. The modifications would come in the Job Stream Processor and in the device management routines.

If there were disk management, there could then come an assignment to, let us say, optimize the order of requests coming in to the disk by a modification to the disk management module. Here can be clearly seen the advantage of having a real operating system on which to base modifications: there is no need to write a clumsy disk-environment simulator.

Once there is disk management, furthermore, there is the possibility of adding a file system. Specifics of this might vary, but it might be noticed that, from the user's viewpoint, the fields on the $JOB card make easy provision

for this.

With disk management and a simple file system (perhaps even none at all, for simple versions) we can perform input and output spooling. And with spooling, addition of various job scheduling algorithms becomes a practical assignment. Exact details here, of course, are best left to the instructor.

APPENDIX B

THE IBM SYSTEM/360

This appendix explains the structure of the IBM System/360 to a degree suitable to allow a reader who is unfamiliar with the System/360 to understand the terminology used in this thesis.

The System/360 is a general-purpose, stored-program digital computer. Its words are 32 bits long and its bytes are 8 bits long. Internal character coding is in EBCDIC, a variant of BCD.

Addressing of data fields is by a 24-bit address giving the beginning byte's address. Each 2K-byte block of memory can have a protection key from 0-15 associated with it, used as described below.

The main processor state register is the Program Status Word (PSW). It stores the following.

The instruction counter, which is the address of the byte following the current instruction.

The program/supervisor mode switch. This bit is used to distinguish between program mode, where certain priviledged instructions may not be executed, and supervisor mode, where

all instructions may be executed. The instruction to reload the complete PSW is a priviledged instruction, as are instructions to reload sensitive sections of the PSW, or to reset storage keys, or to perform input/output.

The program protection key. This key is used in conjunction with the storage protection keys mentioned above. If it is zero, the program may access any part of attached memory; if it is non-zero, he can access only those sections of memory with a matching key.

A running/waiting flag. This bit indicates whether the program is in running state (normal) or in wait state. A program in wait state is not in execution, but is rather waiting for the occurrence of whatever interrupts are enabled.

Interrupt flags. These flage are used to allow, inhibit, or delay interrupts, depending on the setting of the flags, and the nature of the device.

Other fields, not important here.

Input/output is performed by means of the "start i/o" instruction, which specifies the device address to be used. This instruction sends to the channel the Channel Address Word (CAW), a word stored in a permanently-assigned address of memory. The CAW contains the protection key to be used by the operation, and contains the address of the first

Channel Command Word (CCW) the channel fetches the CCW's, which contain the operations to be performed, along with flags to control such options as command chaining or special interrupts.

Input/output interrupts occur when the appropriate bit in the PSW allows them, or wait until it does. They store a Channel Status Word (CSW) which indicates the status of the request. Then, as on other interrupts, the old PSW is stored, and a new PSW, stored in a permanently-assigned location, is loaded (thus effecting a transfer).

REFERENCES

(Cor 63) Corbató, F.J., Poduska, J.W., and Saltzer, J.H., Advanced Computer Programming: A Case Study of a Classroom Assembly Program, M.I.T. Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1963.

(Cor 65) Corbató, F.J., et al., "A New Remote Accessed Man-Machine System," AFIPS Conf. Proc. 27 (1965 FJCC), pp. 185-247.

(Cri 65) Crisman, P.A., ed., The Compatible Time-Sharing System: A Programmer's Guide, (second edition), M.I.T. Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1965.

(Dij 68) Dijkstra, E.W., "The Structure of the 'THE' Multiprogramming System," Comm. ACM 11, 5 (May, 1968), pp. 341-346.

(Don 72) Donovan, J.J., Systems Programming, McGraw-Hill Computer Science Series, 1972.

(Han 70) Hansen, P.B., "The Nucleus of a Multiprogramming System," Comm. ACM 14, 4 (April 1970), pp. 238-241.

(IBM :1) IBM System Reference Library, IBM 7090-7094 IBSYS Operating System, Version 13, System Monitor (IBSYS), File Number 7090-36, Form Number c28-6248-7, 1966.

(IBM :2) IBM System Reference Library, IBM System/360 Operating System, Planning for Multiprogramming With a Fixed Number of Tasks, Version 11 (M.F.T. 11), File Number S360-36, Form c27-6939-0.

(IBM :3) IBM System Reference Library, IBM System/360 M.V.T. Control Program Logic Summary, File Number S360-36, Form Y28-6658-0.

(Nie 70) Nielson, N.R., "The Allocation of Computer Resources -- Is Pricing the Answer?", Comm. ACM 13, 8 (August, 1970), pp. 467-474.

(Sal 66) Saltzer, J.H., "Traffic Control in a Multiplexed Computer System," M.I.T. Ph.D. Thesis, June, 1966. (Available as Project MAC Technical Report TR-30.)