

REFERENCE

A GUIDE TO MULTICS

FOR

SUBSYSTEM WRITERS

CHAPTER VII

RESOURCE SHARING AND INTERCOMMUNICATION

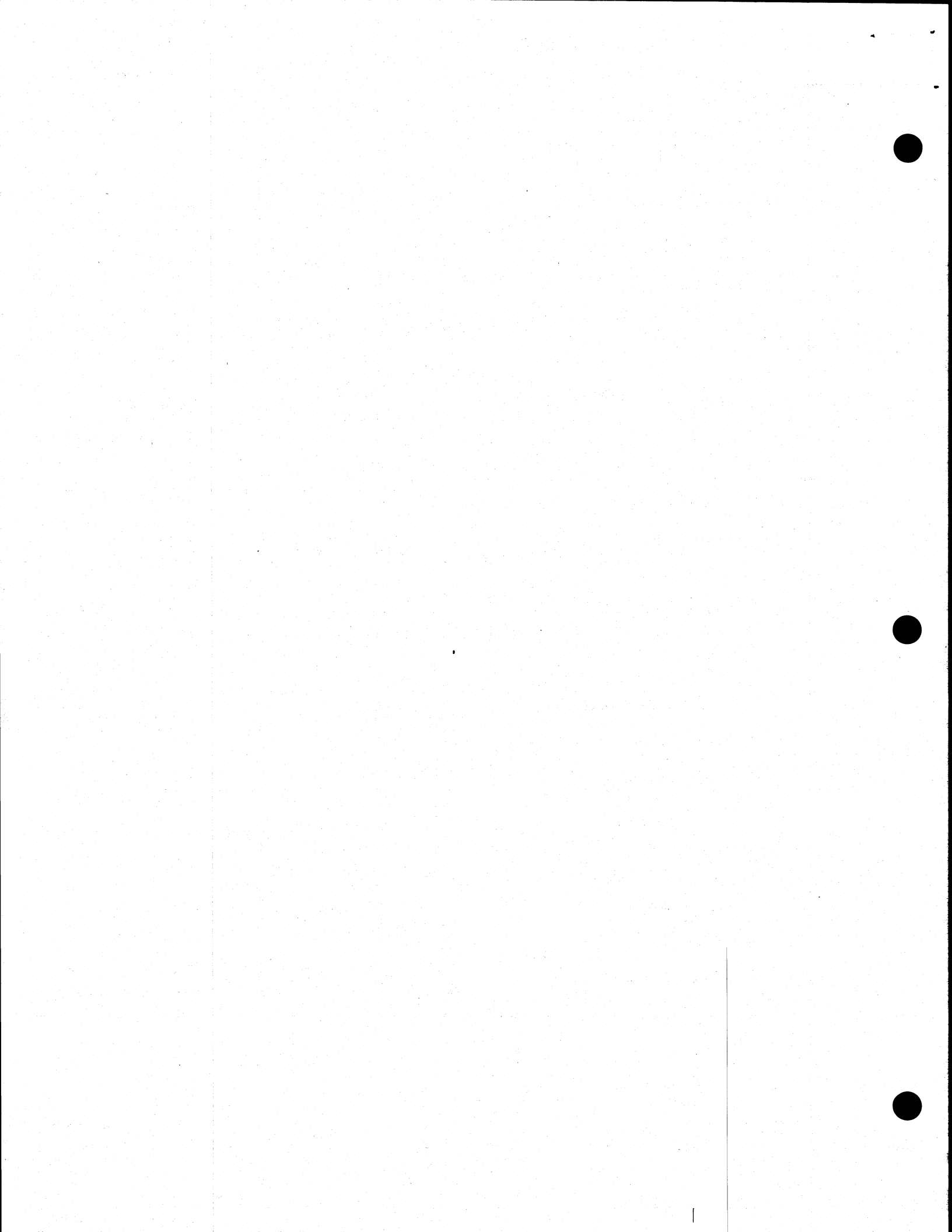
AMONG COEXISTING PROCESSES

Elliott I. Organick

August 1969

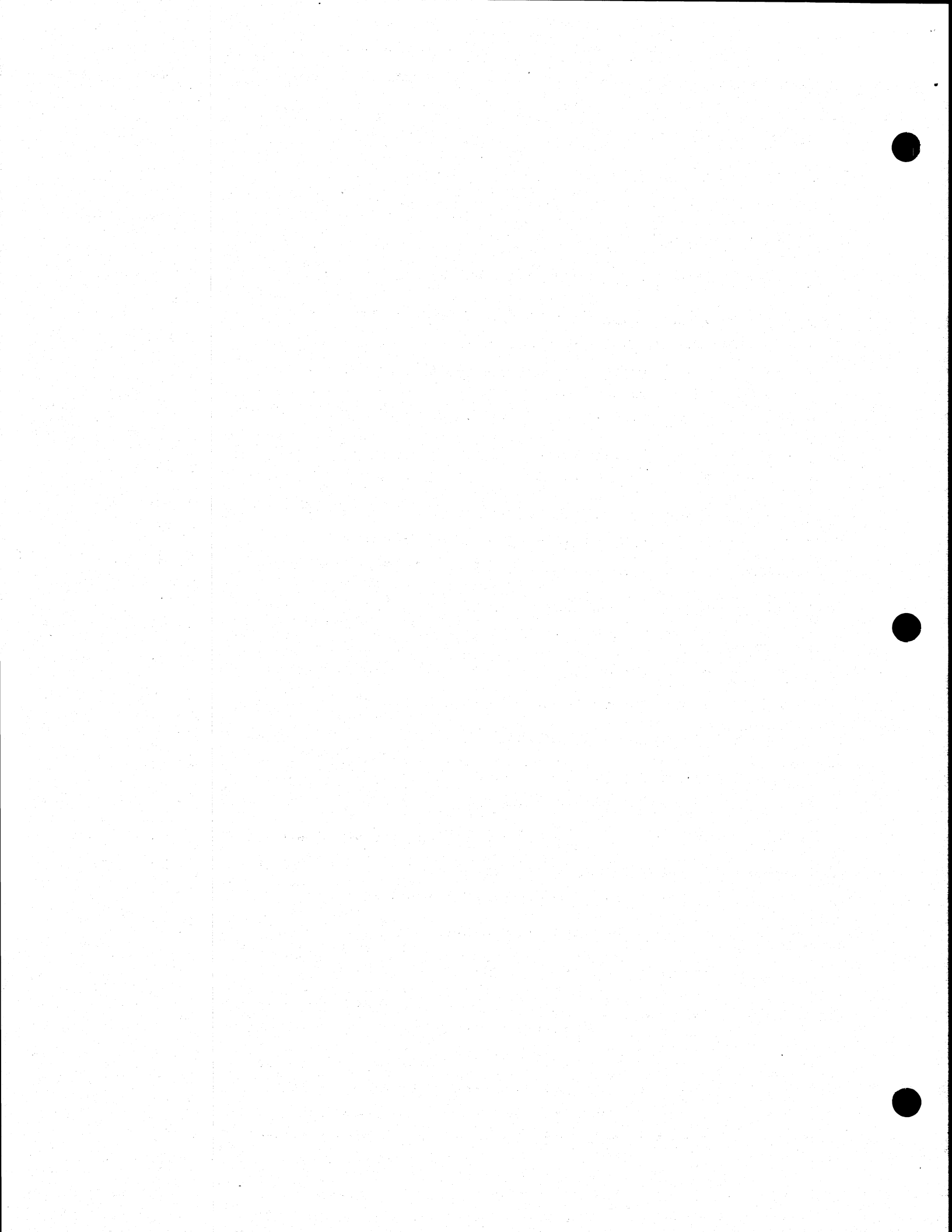
Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

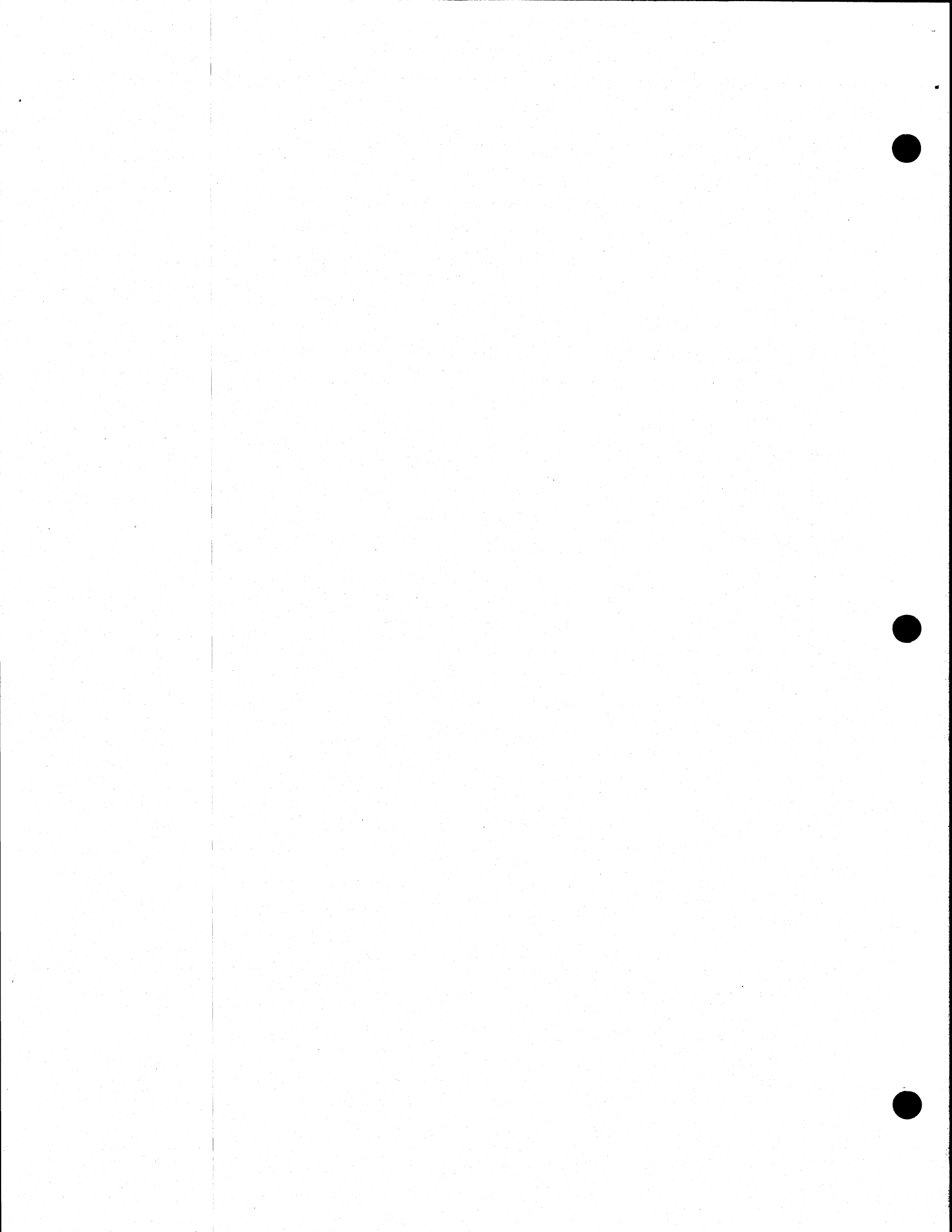


# TABLE OF CONTENTS

| <u>Section</u>                                                                                    | <u>Page</u> |
|---------------------------------------------------------------------------------------------------|-------------|
| LIST OF ILLUSTRATIONS                                                                             | iv          |
| LIST OF TABLES                                                                                    | v           |
| <br>                                                                                              |             |
| 7.1 INTRODUCTION                                                                                  | 7-1         |
| 7.2 MULTIPLEXING PROCESSORS                                                                       | 7-6         |
| 7.2.1 A Simple Mechanism for Multiprogramming                                                     | 7-7         |
| 7.2.2 The TC Used for Time Sharing                                                                | 7-9         |
| 7.2.3 Block, Wakeup Functions for Use in I/O Control<br>and in General Interprocess Communication | 7-11        |
| 7.2.4 Other Control Functions of the Traffic Controller                                           | 7-20        |
| 7.3 CORE RESOURCES EMPLOYED AND MANAGED<br>BY AN ACTIVE PROCESS                                   | 7-24        |
| 7.3.1 Minimum Core Requirements of an Active Process                                              | 7-28        |
| 7.3.2 The System Segment Table and Shared Segments                                                | 7-33        |
| 7.3.3 Handling of Segment and Page Faults                                                         | 7-35        |
| 7.3.3.1 Segment faults                                                                            | 7-35        |
| 7.3.3.2 Page faults                                                                               | 7-36        |
| 7.3.4 Ways to Reduce Segment Faults                                                               | 7-37        |
| 7.3.5 Ways to Reduce Page Faults                                                                  | 7-39        |
| 7.4 ASSIGNMENT OF PROCESSOR RESOURCES                                                             | 7-39        |
| 7.4.1 Time-Sharing Philosophy                                                                     | 7-40        |
| 7.4.2 Multics Design Details                                                                      | 7-42        |
| 7.4.2.1 Insertions in the ready list (scheduling)                                                 | 7-43        |
| 7.4.2.2 Eligibility Management - design details                                                   | 7-45        |
| 7.4.2.3 Pre-emption of Eligibility                                                                | 7-47        |
| 7.4.3 Expected System Response                                                                    | 7-48        |
| 7.5 INTERPROCESS COMMUNICATION                                                                    | 7-50        |
| 7.5.1 The Nature of Processes and the Nature of<br>Their Intercommunication                       | 7-50        |
| 7.5.2 Communication Mechanisms                                                                    | 7-51        |
| 7.5.2.1 Messages, Mailboxes (event channels)<br>and Transmission                                  | 7-53        |
| 7.5.2.2 Setup for Interprocess Communication                                                      | 7-59        |



|         |                                                                   |      |
|---------|-------------------------------------------------------------------|------|
| 7.5.3   | Programming of a Multi-purpose Process                            | 7-64 |
| 7.5.3.1 | Event call channels                                               | 7-65 |
| 7.5.3.2 | Concept of the Wait Coordinator                                   | 7-67 |
| 7.5.3.3 | Call-Wait polling order                                           | 7-68 |
| 7.5.3.4 | Invoking an associated procedure and controlling its repeated use | 7-69 |
| 7.5.3.5 | Other channel management functions                                | 7-71 |
| 7.5.4   | Limitations of Multi-purpose Processes                            | 7-72 |
| 7.5.4.1 | Two case studies using timing diagrams                            | 7-72 |
| 7.5.4.2 | Ways to prevent sluggish event wait response of event call tasks  | 7-76 |

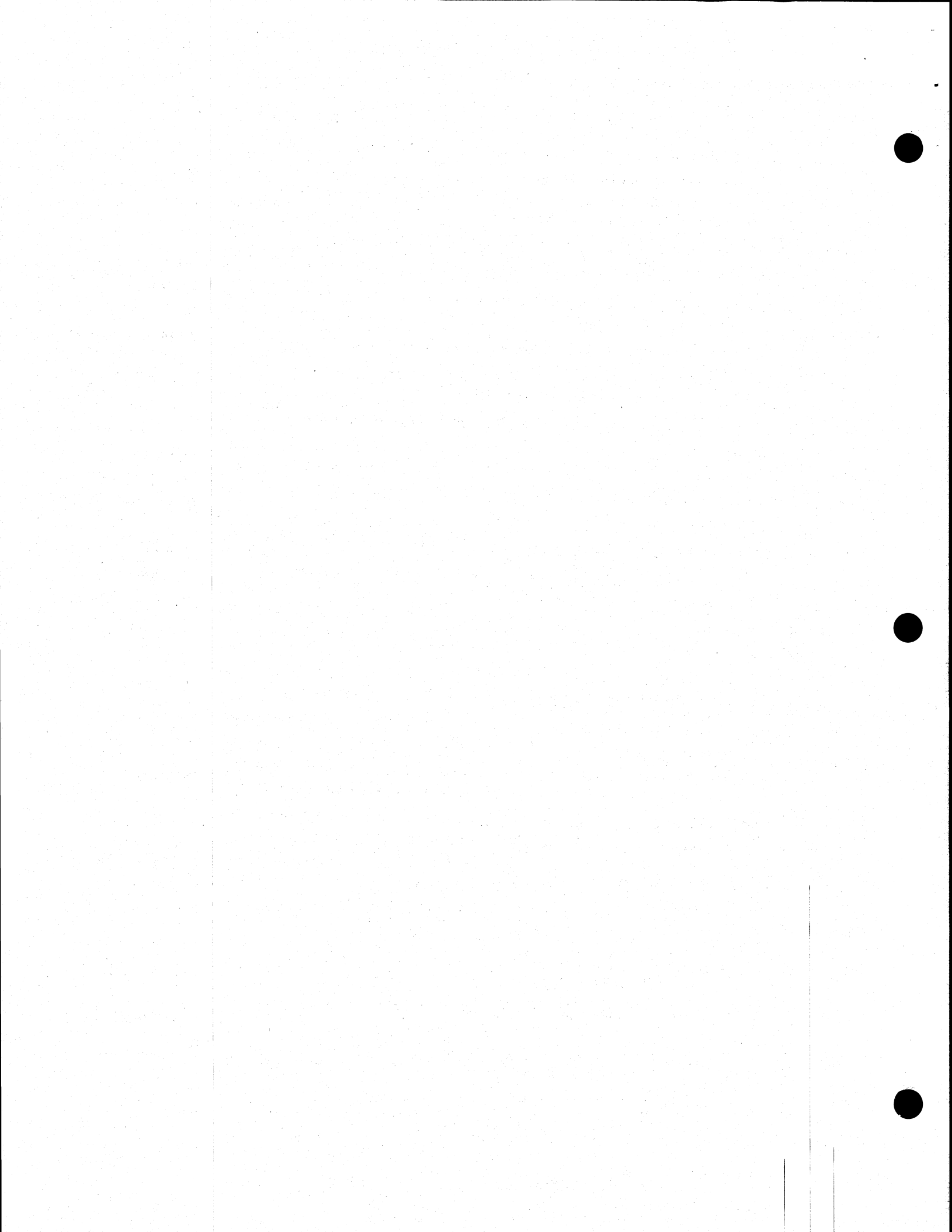


## LIST OF ILLUSTRATIONS

| <u>Figure</u> |                                                                                                                            | <u>Page</u> |
|---------------|----------------------------------------------------------------------------------------------------------------------------|-------------|
| 7-1           | Invoking and synchronizing a parallel action                                                                               | 7-4         |
| 7-2           | Schematic showing the synchronizing of a user process with the tty manager process for a write operation (simplified view) | 7-14        |
| 7-3           | More detail for box 2                                                                                                      | 7-16        |
| 7-4           | More detail of the initiation of an I/O operation                                                                          | 7-17        |
| 7-5           | Illustrating the conversion of system interrupts to process wakeups                                                        | 7-19        |
| 7-6           | The n priority level queues of the ready list                                                                              | 7-44        |
| 7-7           | Variation of R (the ratio elapsed time to virtual time) as a function of virtual time                                      | 7-49        |
| 7-8           | Time-line characteristics of sequential processes                                                                          | 7-51        |
| 7-9           | Some details of the Traffic Controller's entry point <u>wakeup</u>                                                         | 7-55        |
| 7-10          | Message format augmented with information about the sender as placed in the ITT                                            | 7-56        |
| 7-11          | Postal system analogy to Multics interprocess message transmission                                                         | 7-57        |
| 7-12          | The chain of calls: $\rightarrow \text{ipc}\$block \rightarrow \text{hcs}\$block \rightarrow \text{TC block entry}$        | 7-60        |
| 7-13          | Wait lists: General and Specific                                                                                           | 7-63        |
| 7-14          | A possible structure for a multi-purpose process                                                                           | 7-66        |
| 7-15          | How the ipc\$block functions as a Wait Coordinator                                                                         | 7-70        |
| 7-16          | Timing diagram showing execution of tasks triggered by event calls only                                                    | 7-73        |
| 7-17          | Timing diagram showing execution of tasks triggered by event calls but which are held up at wait points                    | 7-74        |
| 7-18          | Timing diagram (same as Figure 7-17) with masking and unmasking of event call channels                                     | 7-77        |

## LIST OF TABLES

| <u>Table</u> |                                                                                   | <u>Page</u> |
|--------------|-----------------------------------------------------------------------------------|-------------|
| 7-1          | The Static Set (minimum core requirements while an Active Process is not running) | 7-30        |
| 7-2          | The Dynamic Set (minimum core requirements while an Active Process is running)    | 7-31        |





## CHAPTER VII

### RESOURCE SHARING AND INTERCOMMUNICATION AMONG COEXISTING PROCESSES

#### 7.1 INTRODUCTION

The earlier chapters gave us an increasingly larger view of an executing process. We began in Chapter 1 with the microscopic view that focused on the minute, but nontrivial details in the fetch and execute of individual GE 645 instructions in an executing process. Upon completing Chapter 6, we have managed to enlarge our view of a process in execution about as far as possible from the "in-vacuo" point of view taken thus far. That is, for the most part, we have been considering the process in isolation, as if it were the only one employing the computer system's resources. We know that each executing process coexists in some sense with other processes, some may be executing simultaneously on other processors (if there be more than one in service), some are waiting a "turn" to execute on a processor, and still others may be waiting for some event whose occurrence will enable the process to proceed with execution. The collection of these coexisting processes clearly implies (a competition for and) a sharing of hardware resources, a sharing of system software and data bases and control over this sharing. Most of the control functions described previously were of a per-process nature and the data bases considered were of a one-per-process type, e. g. , stacks, descriptor segments, KST's, etc. In this chapter we will be examining controls of a per-system nature. Of course, the data bases that associate with these functions are central to the operation of the entire system. Hopefully, when a subsystem designer understands how a process functions (cooperates and/or competes) in a milieu of other processes, he can better anticipate the performance of the processes in which his subsystem(s) resides.

#### Types of Coexisting Processes

In this overview section we shall anticipate what follows by summarizing the types of processes that coexist in Multics and provide a rough indication as to the nature of their interaction. We identify three kinds of coexistence.

1. Sets of seemingly unrelated user processes. (Multi-programming.)

Experience with earlier operating systems including CTSS has shown that it is a rare console user whose process can fully utilize a fast processor. Characteristically, a user process makes frequent requests for relatively slow-to-commence block

transfers of information from drum, disk or other I/O devices. Even with devices that have high transfer rates, there is, to begin with, an associated latency of several milliseconds or more, i. e., a delay before the to- or from-core transfer may begin. During the delay and subsequent transfer time, it may not be possible for the requesting process to do any useful work. (This is certainly the case in Multics when a segment or page fault has led to the initiating of the request for a drum or disk transfer to core of the desired segment or page thereof.) In principle, either the CPU must remain effectively idle while the process waits for completion of the block transfer or the about-to-be idled process must somehow relinquish the processor to another process which is in a position to execute on a processor at this time. Systems for "passing the processor around" among several processes, so as to prevent the idling of a CPU during I/O waits or other delays, are known as multi-programming systems. Multics is, among other things, a multi-programming system.

The set of processes that share a processor in the fashion we just crudely described need not be related to one another in any explicit way. They may, for instance, be a set of arbitrary user processes. Nevertheless, interaction among these processes is clearly implied. First, they share the same supervisory procedures and certain system data bases (tables), as segments in their respective address spaces. Second, each process is compelled, while executing, to occasionally assist an idled process that is waiting for a particular event to "arrive." Although in each case the executing process must cooperate, the user should be, and is, completely oblivious to the fact that his process is performing as a Good Samaritan. This is because all such activities will occur while his process is trapped in certain ring-0 supervisory routines. Since these routines are common to all user processes, all processes are guaranteed to be Good Samaritans.

There are several ways an executing process may know about the arrival of an event which is of interest to another (non-executing) process. However this knowledge is acquired, the waiting process is alerted so that it may again compete for time on the processor (or on a processor, if there is more than one).

An executing process may, for example, be interrupted (by an identifiable signal from another active unit) and in this way "told" about an event of interest to another process. Should this occur, the interrupted process is forced to (in some sense) wake up the process for whom the interrupt signal is of primary

interest. Alternatively, the executing process may of its own discover the apparent arrival of such events. This type of discovery happens with great frequency in Multics. When a process incurs a page fault and must wait for the arrival of a requested page from the drum, it must put itself into a wait state. Just before doing so, it always checks a certain system-wide I/O request list in which it can discover which, if any, paging requests (of other processes) have been completed. The executing process then "notifies" the appropriate waiting processes before making itself idle.

## 2. User process and a system process

Multics provides a set of "ever-present" system processes that offer specialized services to user processes. Among these, for instance, is the output driver, a system process that drives line printers and other output devices to produce copies of user-designated segments. A user will usually communicate implicitly with such a process by executing a system library subroutine call. This routine in turn executes the explicit steps needed to communicate an unambiguous work request to the system process so the user is, in fact, insulated from the details of interprocess communication.

Because the system process is a separate and fully independent process, its functions may be achieved as a parallel operation. The system is free to fulfill the requests it receives at its convenience. With the current implementation, the user process proceeds to other chores without waiting for an acknowledgment from the output driver that the requested output task is done. For illustrative purposes, however, we can also imagine that when requested such a system process could send a meaningful completion signal to any user process. The latter might either wait for and be awakened by the completion signal or periodically inspect a special "mailbox" for the presence of a message sent by the former to indicate completion of the task.

## 3. Sets of deliberately cooperating processes.

The computation structures of many algorithms exhibit parallelism that can never be taken advantage of when using only one processor. There is an increasing interest in the computer community in providing the operating system machinery which would permit parallel computation. The Multics design provides a simple capability of this type.

The ordinary programmer notices parallelism at various levels, from the PL/I statement level on up to the subroutine. Thus, in the righthand side of the statement

$$Y = (A*B + C)/(D*F + E);$$

computation of the numerator can, in principle, be carried out in parallel with that of the denominator. Likewise, but at a more macroscopic structural level, it is conceivable that in the statement

$$T = \det(A) * \cos(x);$$

the function for computing the determinant of the matrix A could be executed in parallel with the computation for the cosine of x, if each subroutine could be invoked to execute in parallel on separate processors.

If one could invoke separate and parallel computations, a mechanism for synchronizing the two parallel functions would also be needed. Thus, the multiplication of  $\det(A)$  and  $\cos(x)$  clearly must be delayed until it is known that both subroutines have returned values. Suppose, for instance, we consider flow structure as depicted in Figure 7-1.

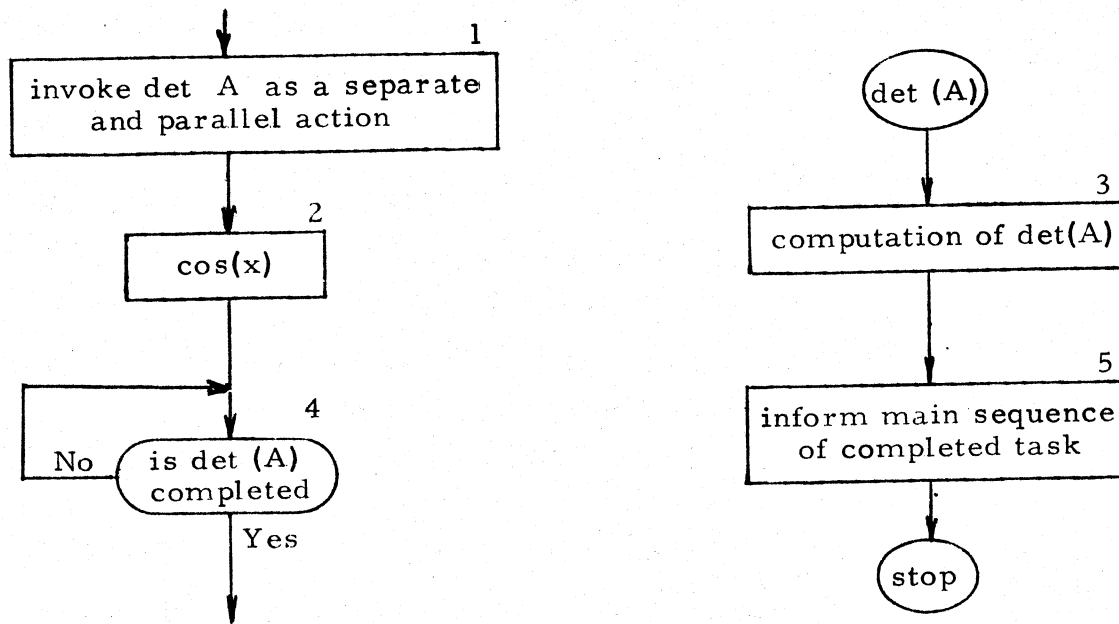


Figure 7-1 Invoking and synchronizing a parallel action

Boxes 1, 4 and 5 are representative of the logic required to invoke and to synchronize the parallel action. Boxes 1, 4 and 5 can be carried out only if some common data cells are shared between the two computations for use, so that one computation can communicate with the other.

Clearly, there is some trade-off between the savings in time that can be achieved in the parallel computation and the extra costs associated with use of the machinery for achieving these gains. In Multics, opportunity for such computation is provided. Parallel computation can be achieved by executing two or more processes concurrently. However, in a Multics system with several CPU's, the user is never given an opportunity to force their simultaneous allocation to his separate processes. Hence, in Multics, "parallel computation" is just a possibility, never something that can be guaranteed. Although it is more realistic to regard the execution of such separate computations as asynchronous rather than parallel, we shall generally use the latter term and understand it in its properly qualified sense.

Machinery for interprocess communication is provided in Multics by which to create and/or invoke other processes and also to synchronize with other processes using shared data bases known as "event channels." With this machinery, meaningful cooperation (e.g., parallel computation) can be conducted. (The scope of the parallel tasks may be large or small, as the user or users desire.) Explicit user programming, of a type to be described in this chapter, is required to achieve this objective. Moreover, the programming for each such "subsystem," for we can indeed regard such planned cooperation as a subsystem design, is specific to the objective at hand.

In review of the above three types of coexisting processes, we see that in all cases:

1. "Cooperation," whether voluntary or involuntary, preplanned by the system or explicitly planned by the programmer, implies communication between processes through the use of shared data bases.
2. Of necessity, all coexisting processes, whether they cooperate or not, also compete for processor time and core space.\*
3. By design, all processes share common supervisor modules and certain system tables.

For gaining additional perspective, it is well to consider how Multics differs in its design approach for achieving and controlling parallelism from the approach

---

\* We make the implicit assumption that nearly always there are more processes able to execute than there are available processors. A similar assumption is made with respect to core space, i. e., there is not enough to "go around."

taken in more traditional operating systems.

In earlier operating systems, parallel operations were limited to I/O activities, hence the mechanisms for controlling parallelism (interrupt handling and dispatching) became embedded in supervisory packages called I/O Control Systems. Of course, in the context of the more "modern" multi-access, multiprocessor systems the same or equivalent mechanisms together with some new ones are also inherent in (a) achieving the multiprogramming of unrelated coexisting processes (type 1), and (b) in the invoking and synchronizing of parallel computations in coexisting processes (type 3). The additional mechanisms include appropriate locking controls on shared data bases and the means of communicating (e.g., sending signals) between the independently operating hardware processors. For this reason, Multics has split out the traditional aspects of I/O Control Systems having to do with parallelism (interrupt handling and dispatching) and has combined these with the other aspects of parallel processing.

The combination resulting from this unified viewpoint has led to the development of a single, general purpose supervisor subsystem for control of all parallel operations. This subsystem is known as the Traffic Controller.\*

## 7.2 MULTIPLEXING PROCESSORS

Our immediate goal is to see how Multics achieves the orderly and effective multiplexing of its processor(s) among the coexisting processes. A set of modules referred to as the Traffic Controller is responsible for this activity.

We shall learn about the functions of the Traffic Controller (or TC) in an incremental fashion. First, we view those functions needed to support multiprogramming. These are the mechanisms to give away and get back a processor when predictable time delays, such as those due to paging, are forced on a process. Next, we shall consider the TC viewed as a general mechanism for time sharing, for interprocess communication, and for achieving still other control functions.

---

\* The Traffic Controller was first formulated by J. H. Saltzer in a lucid Ph. D. dissertation (MAC TR-30, July 1966, "Traffic Control in a Multiplexed Computer System"). A modification of Prof. Saltzer's original design, developed in the M. S. thesis by Robert Rappaport, has been incorporated in Multics. The principal MSPM references are the BJ sections.

### 7.2.1 A Simple Mechanism for Multiprogramming

Consider a set of  $n$  seemingly unrelated user processes in a system of  $k$  processors ( $k < n$ ). Each process coexists in one of three "execution states":

running, ready, or waiting.

A running process is one that is currently executing on a processor. (At most,  $k$  of the processes are in the running state.)

A ready process is one that would be running if a processor were available for it to run on. (There are at most  $n-k$  ready processes.)

Since we picture that processes will frequently incur page faults, some of the  $n-k$  non-executing processes will be waiting for the completion of previously invoked paging requests (normally from the drum). So, we define a waiting process as one that cannot make immediate use of a processor (even if one were available), because it is waiting for a so-called system event to happen. Arrival of a page in core is an example of a system event, which can be defined as an event which

- (a) is of interest to at least one coexisting process\* and
- (b) the waiting time for whose occurrence has a predictable upper bound.

Waiting processes compete for processors in the sense that once the waited-for event has occurred, the processes should then be regarded as being ready.

The Traffic Controller's tasks for multiplexing processors among this class of processes are conceptually simple. Its activities center around the maintenance of a list of the coexisting processes. (This list is called the Active Process Table, APT). For each process on the list, the Traffic Controller (TC), associates the current execution state and other vital data. Thus, for a process that is marked as running there is also recorded a code that identifies the particular processor on which the listed process is now executing. Entries for processes that are marked ready can be pictured as belonging to a so-called "ready list." Although we shall examine this list in greater detail later, for the moment it is

---

\* Two processes could conceivably take page faults for the identical page of the same shared segment. The page fault in the second process could occur after the page fault taken by the first process, but before the page request initiated by the first process has been completed. The net result is that when the system event (completing the page request) finally occurs, it will be of interest to two waiting processes.

best to view the ready list as a FIFO-managed list. Finally, for each listed process that is marked as waiting there is recorded an identifier for the event being waited for.

When a Good Samaritan process (executing in ring 0) notes that a waited-for system event has arrived, it calls the Traffic Controller to "notify" the appropriate process(es) that has (have) been waiting. The Good Samaritan is, in nearly all instances, any process that may have just taken a page fault. While trapped in the ring-0 supervisor for the purpose of initiating a page request, this process automatically scans a system-maintained list of (drum/disk) I/O requests in search of those that are marked as satisfied (i. e. , done). If any are found, the TC is called, giving it an identifier for the event that has occurred. The TC then uses this identifier to notify the appropriate processes in the following manner. For each given event identifier the TC locates on its list of processes those that are waiting for the identical event. For each such process, the TC then alters the code for its execution state from waiting to ready, and makes this process a part of the ready list. Eventually, this entry becomes topmost on the ready list. (Remember, we are thinking of it as a FIFO list). The TC will select the associated process for execution, and when this happens, the APT entry is recoded as running, thereby removed from the ready list. The code for the processor given to this process is also recorded in this APT entry.

In summary, we see that notifying a process of a system event does not immediately place it in the execution state. A process must first pass through the ready state enroute from waiting to running.

It may have occurred to you to ask the following question: Once put into the running state, is there any iron-clad guarantee that the page for which a process had been waiting will, in fact, be there? There is some possibility that in the interim, between the time the process was first "notified" that its requested page was in core and the time it finally reacquired a processor to reference that page, the page has again been removed from core. This situation could arise in the following way. Let "x" be the page in question. Then, during the said interim, which could be a long one, other running processes may have page demands that are satisfied by "pushing out" page x. If in fact this situation were actually to occur, the victimized process, when it regained the running state, would re-execute the original faulting instruction and again cause a page fault. The running



process would reinitiate the same page request, play Good Samaritan to call the TC to notify others of completed page requests, and again call the TC to "put itself" into the wait state and give up the processor to an eligible (ready) process.

The hypothetical situation just described pictures a process cycling through the running-wait-ready states without ever accomplishing anything but page faults. This situation is one of several types of "system thrashing" which the system designer is always bent on preventing. Thrashing is prevented in Multics in the following way. The number of processes eligible for CPU attention is kept below a limit which, if exceeded, would cause thrashing. This limit would be respected even if it means allowing a CPU to become idle occasionally. Moreover, a process that is forced into the wait state never loses its priority relative to the other eligible processes. So, when the process has been notified that it may resume execution, there can be at most a limited number of processes queued ahead of it. The possibility that the desired page would be pushed out before it can be used by each process that has faulted on it is thus made negligible. We defer until Section 7.4 a full elaboration on how this fine control is achieved.

#### 7.2.2 The TC Used for Time Sharing

If a running process is executing in a computation loop (deliberate or accidental) such that it takes no page faults over an extended period, what prevents this process from "monopolizing" the processor? The scheme for multiprogramming that was described in the preceding subsection does not indicate any way that other processes (ready or waiting) can get their "turn." Clearly, an additional mechanism is needed to force a sharing of the processor on the basis of elapsed time in execution. This mechanism is provided in the TC by assigning to each coexisting process an appropriate execution time allotment,  $q$ , such that when a process has executed for a total of  $q$  time units, it is forced to give up the processor. The time allotment is a value that is under the control of the system administrator. The control is achieved with the help of hardware as elaborated slightly in the following discussion.

First, it should be noted that the time allotment,  $q$ , for each process is assigned by a module of the TC called the "scheduler." The value for  $q$  is stored in the Active Process Table entry for the process. Also kept in the same entry is the amount of time,  $r$ , which has already been used in execution against the allotment,  $q$ . When a process enters the running state, one can picture that

the TC sets a timer register with the value  $q-r$ . The timer register then counts down to zero. When it reaches zero, a combination of hardware and software causes the generation of a process interrupt. The interrupted process then calls an appropriate entry point in the Traffic Controller which will

1. "reschedule" the now running process for execution at a later time, and
2. give away the processor to the "next" ready process.

The term "rescheduling" refers to the tasks of

- (a) giving the interrupted process a new time allotment  $q'$  (not necessarily equal to its last value of  $q$ ) for use the next time it is allowed to enter the running state;
- (b) putting the process into the ready state by marking the execution state as ready, resetting the value of the time used ( $r$ ) to zero, and making other updates to its APT entry. The business of deciding where on the ready list to place a process is discussed in Section 7.4.

When a running process moves to the wait state because it has incurred a page fault (or is forced to wait for some other system event, e. g., the unlocking of a system table), the value of  $r$  is kept in its APT entry and is incremented by an amount inferred from the reading of the timer register. Typically, the process' current time allotment is, in fact, used up over a sequence of short executions, each punctuated by a page fault or other system delay that causes the process to pass through the wait and ready states. Eventually, the time allotment is used up, at which time the process must be "rescheduled." Other things being equal, if a process must be rescheduled, it is given a larger time allotment, but it is also given a lower priority, which in effect means that its "insertion point" in the ready list is made correspondingly less favorable.

In addition to giving a more detailed look at rescheduling, Section 7.4 also describes an eligibility restriction and a set of pre-emption mechanisms. Eligibility refers to the depth or "degree" of multiprogramming. It is the number of processes that are permitted to compete for a processor at any one time. The number of eligible processes is necessarily restricted in order to prevent thrashing, i. e., destructive competition for the limited core resources. In simple terms, eligibility can be viewed as a conserved resource of the system that is passed about among the processes—like the fixed and limited number of lunch

trays that is circulated among the much larger number of daily customers that pass through a cafeteria. Eligibility is first conferred on a process when that process reaches a certain preferred point on the ready list. Eligibility is later withdrawn from a process when it is rescheduled (moved) to a less favorable position on the ready list or when it leaves the ready list for a long time.

Pre-emption refers either to the capture of a processor (CPU pre-emption) or to the capture of eligibility (eligibility pre-emption). The former allows an eligible process that is being readied, if it is "important" enough, to cause the capture of a processor. Capture by a high priority eligible process is either immediate, in case it is being notified that a page read has been completed (or some other system event has occurred) or at the end of the next time unit (currently one second), in all other cases.

CPU pre-emption in the current implementation of Multics occurs in a completely automatic way. For example, whenever an ineligible process that is being rescheduled has a high enough relative priority, that process becomes a candidate to pre-empt a processor. The process will, in fact, pre-empt a processor if a search of the APT entries for those processes now running (including that of the process doing the searching) reveals one (process) that is "less important." In a multi-processor configuration, if there is a choice, the running process that is "pre-empted" will be the one of lowest "importance." Any process that is CPU-pre-empted is rescheduled (e. g., put back on to an appropriate point in the ready list).

Eligibility pre-emption is also automatic, permitting a high-priority but ineligible process to capture the eligibility of a lower priority running process.

### 7.2.3 Block, Wakeup Functions for Use in I/O Control and in General Interprocess Communication

Two more mechanisms are provided in the TC that are designed mainly to facilitate the synchronizing of deliberately cooperating processes. These are called the block and wakeup functions. They are functionally different from the previously described wait and notify functions. The wait, notify mechanisms, which can be called only from ring 0, allow a process to wait on (and later be notified of) system events; block, wakeup mechanisms allow a process to wait on (and later be notified of) so-called process events.

By a process event we mean an occurrence that can be of interest to only a specific process (or set of specific processes). The waiting period for such an occurrence will not, however, be either predictable by the supervisor or bounded. To retain the distinction between the two types of waiting, we say that a process enters the blocked state when a process begins waiting for a process event. We will say that the process receives a wakeup when it is notified of the occurrence of that event.

Before going further into detail of these TC mechanisms, it will help to consider an illustrative example (somewhat contrived) of a user process synchronizing its activities with a hypothetical system process that manages teletyped I/O (tty manager)\*. We will picture an I/O operation involving the typing out of a string of several thousand characters.

Suppose the tty manager shares a segment with the user process that can be regarded as an output buffer. For simplicity, let it be 270 characters in length. We picture that the tty manager copies characters out of the buffer in amounts that range up to 270 characters at a time (enough to type up to three full lines on a certain brand of teletype). The user process attempts to move up to 270 characters of the long output string into the smaller buffer area on each transit through its write loop. With the aid of pointers into the buffer, each process is able to interpret the information in the buffer in an appropriate way. Thus, the pointers identify and delimit the next group (up to 270) characters in the buffer which may be moved out (in groups of up to 90 characters) by the tty manager. The same or other pointers tell the user process which set of spaces within the buffer are "open," i. e. , may be filled with the next group of characters from the output string. Two situations are apt to arise.

- a. The user process may find at this instant that there is not enough room in the buffer for the next group of  $i \leq 270$  characters to be copied into it. We will presume that under these circumstances, the user process would want to place itself into the blocked state

---

\* Readers should realize that in the present implementation of I/O Control in Multics, I/O supervisory procedures that control teletypes are part of each user process. No manager process is needed as a "middle man" or broker to execute these I/O functions. The hypothetical example of the manager process, once thought to be useful for system-wide service, is instructive and may prove applicable in the design of special subsystems.

until the tty manager has had an opportunity to "empty out" enough of the buffer to provide the room needed by the user process.

- b. The tty manager may find it has emptied out the buffer, i. e., there is no information in the buffer to be moved out. In the event there is nothing else that the tty manager can do, it would then have to wait for the arrival of more data into the buffer. The tty manager would then want to put itself into the blocked state to await the desired event.

Note (according to our earlier definition) that the events for which both the user and manager process might wait are process events. Thus, no other process but the manager will be interested in being notified that there are now characters in the buffer which are to be copied out onto the teletype. Moreover, there is no general way to predict how long the manager might have to wait for this notification, because the user process may incur various delays (including delays due to paging, due to computations of arbitrary length, or due to entering the blocked state) in the course of cycling through its write loop. Thus, the user process might be preempted or timed out during any one transit of its write loop.

There is, in fact, a symmetry here in the synchronizing of these two processes that can easily be seen. If one process, say the user, blocks itself, the other process (in this case the manager process) wakes up the first process and vice versa. Also, note the important implication that when each process blocks itself it is counting on the other process to "wake it up." It is usually unimportant for the blocked process to know when it will be awakened, but it is always crucial for that process to know it will be awakened.

We are now ready to see how the mechanisms block and wakeup that are provided in the TC would be applied. Our initial view will of necessity be greatly simplified. A more complete description is given in Section 7.5. Figure 7-2 will be helpful for our present purpose. It sketches some of the details in the write loop of the user process and in the synchronized read loop of the tty manager, which, taken together, characterize the buffered write operation. The synchronizing steps (loops) being described here are generated as a result of using ordinary source language I/O calls. In this chapter we shall not consider how the I/O control system converts user-written calls such as:

```
call write_out(string);
```

into the steps being described in Figure 7-2.

Write Loop of the User Process (A)

Read Loop of tty Manager Process (B)

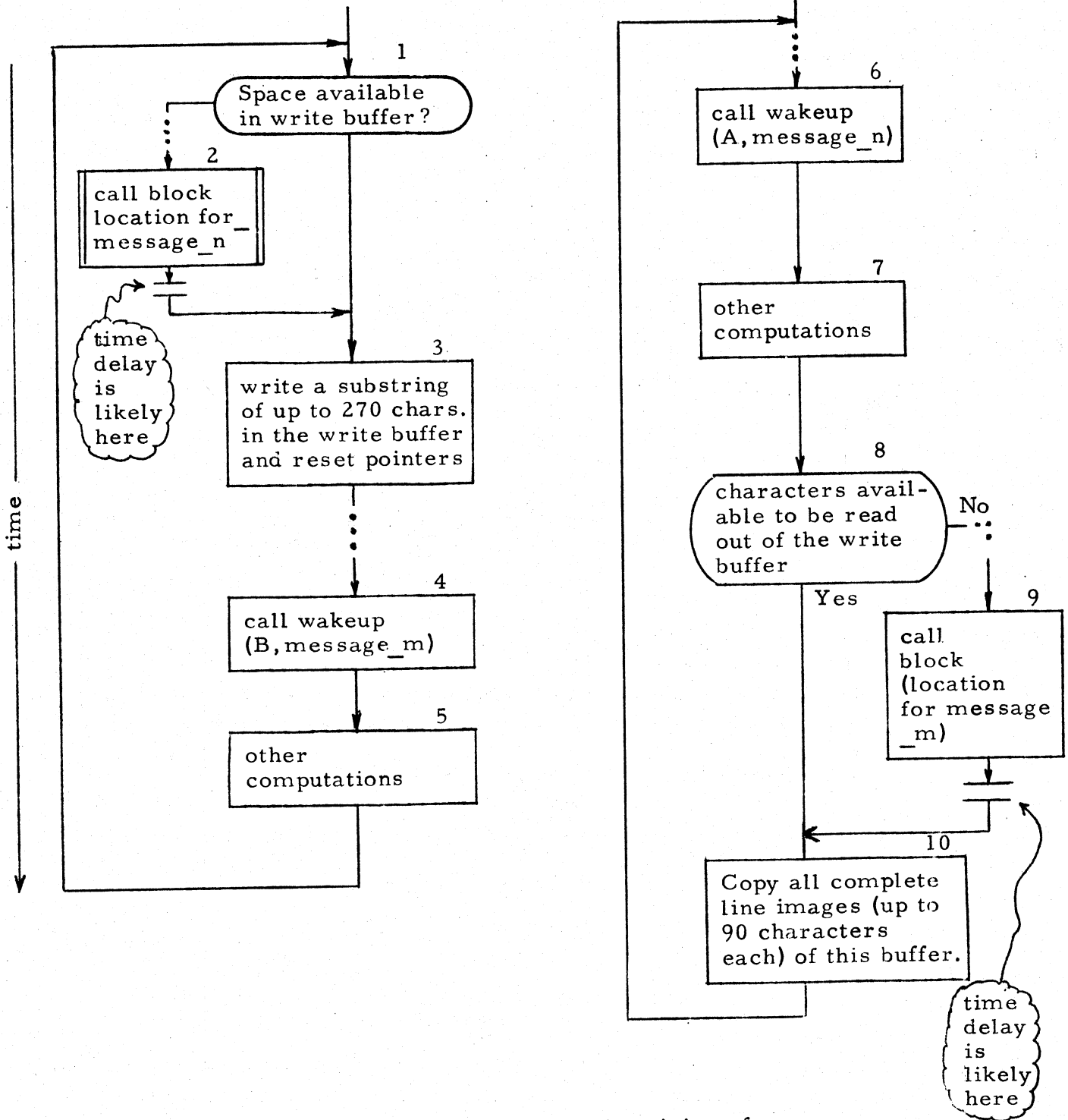



Figure 7-2 Schematic showing the synchronizing of a user process with the tty manager process for a write operation (simplified view).

It is advisable to begin the discussion of Figure 7-2 at box 1. If the user process fails the test in that box (i. e. , no place open in the buffer), it calls the "block" entry of the TC. The three dots on the line between boxes 1 and 2 and between boxes 2 and 3 are provided to suggest that the call to block and the return from it are really handled indirectly, i. e. , through a chain of intermediate (supervisory) routines to be described in Section 7.5. In actual fact, a user will not be aware that his process is calling block. The argument in the call is a returned pointer to a location where A can expect to find a "message" signifying the event(s) being waited for has (or have) arrived. Basically, what the TC does when called at box 2 is the following:

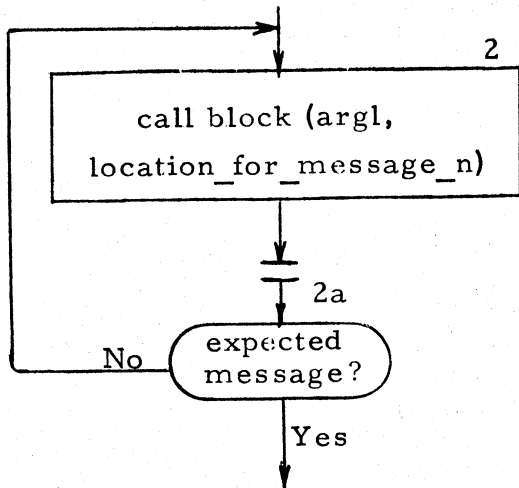
The APT entry for process A will be marked blocked and the processor will be switched to the process which is currently at the top of the ready list.

Process A has now been taken out of the running state and hence cannot return from its call to block (box 2) until after the event it waits for has arrived. This is why the line from box 2 to box 3 is shown with a break (). A can return to the running state and thereby "jump the line break" in the flow diagram (so to speak) only after process B has executed a call to the wakeup entry of the TC (in box 6). In this call, process B names as arguments A's unique process identifier and a message, message\_n, that represents the event A is expecting. The message must later be recognized by process A before A can re-enter the running state.

The TC upon being called at box 6 will place the given message\_n in a system-wide shared data base that is, of course, accessible to process A. The TC also places a pointer to the message in the APT entry for process A. Here, we bear in mind that only the TC is allowed access to the APT (which is also a system-wide shared data base). Next, the TC places process A in the ready state, making A's entry part of the ready list. Wakeup now returns to its caller and execution in process B proceeds through box 7.

Now that A is in the ready state, it can compete again for a processor. When A subsequently gets a processor, it will resume execution within the TC module in which it (A) was last executing and then return from block at the exit of box 2. Block returns the pointer to the message sent by B as a return argument (i. e. , the second argument in the call).

intermediate step between box 2 and box 3 which was omitted to keep its structure as simple as possible during a first view. To go from box 2 to box 3, an intermediate system routine makes a check to be sure that the message received is an expected event message and not a spurious or irrelevant one. \* If spurious, then box 2 is repeated. The schematic shows the flow between box 2 and the delay that follows.



More detail for box 2

structure is appropriate to replace box 9.

### System Interrupts into Wakeups

We will expand the detail of box 10 in the model given in Figure 7-2. After having initiated teletype output, the manager process can do other things while this relatively slow output operation is completed. We suppose that in certain circumstances it is appropriate for the tty manager to give up the calling block. In this case it would be more realistic to consider a structure in Figure 7-4 to replace box 10.

process P may be in communication with more than one other process and C. At any one point, however, process P may enter the state of waiting for a wakeup signal from B. Suppose that shortly thereafter a wakeup signal, for whatever reason, is received. Chaos would then result if P were permitted to proceed on the assumption that its expected signal had been received.



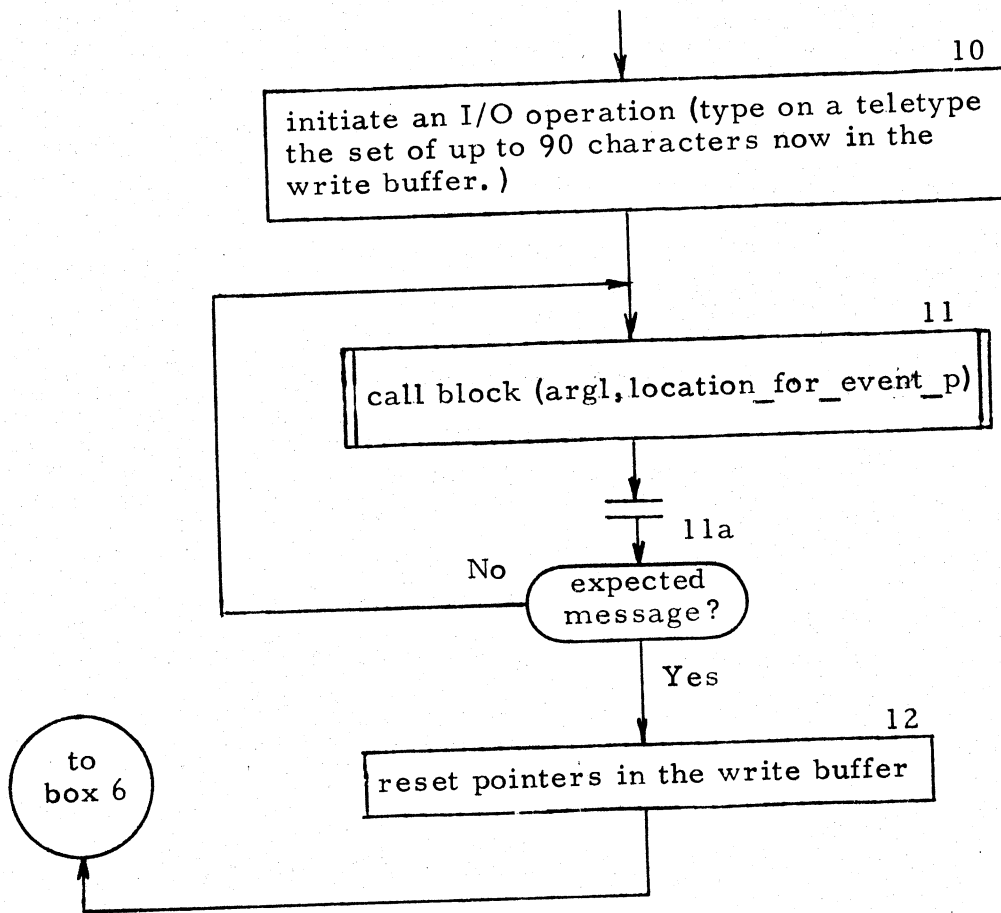


Figure 7-4 More detail of the initiation of an I/O operation

How, then, will the tty manager receive word of the completion of the I/O operation? That is, who (what process) wakes up the manager so that execution may proceed to box 12? The user process, A, may itself be in one of the nonrunning (blocked, waiting or ready) states while the tty manager is blocked. Thus, process A cannot be counted on for any help. Clearly, some "third" process must be involved. In the Multics I/O system design, the third process is any process that happens to be executing on the processor when it receives a hardware interrupt signal that is intended to indicate completion of the invoked I/O operation. The executing process is forced to play a Good Samaritan role because all system interrupt signals (I/O completion signals are examples of such interrupts) are handled by the Traffic Controller. By design, an I/O completion signal comes into the GE 645 memory and triggers the interruption (trapping) of whatever process happens to be executing on the affected processor. An invoked interrupt interceptor module then converts this signal into a wakeup call to the TC, identifying

the process that should be waked up and providing it a message that signifies the device on which the I/O task has been completed. Just as soon as the call to wake-up is completed, control returns from wakeup, and routine execution of the interrupted Good Samaritan continues. The foregoing concepts are suggested in Figure 7-5.

### General Interprocess Communication

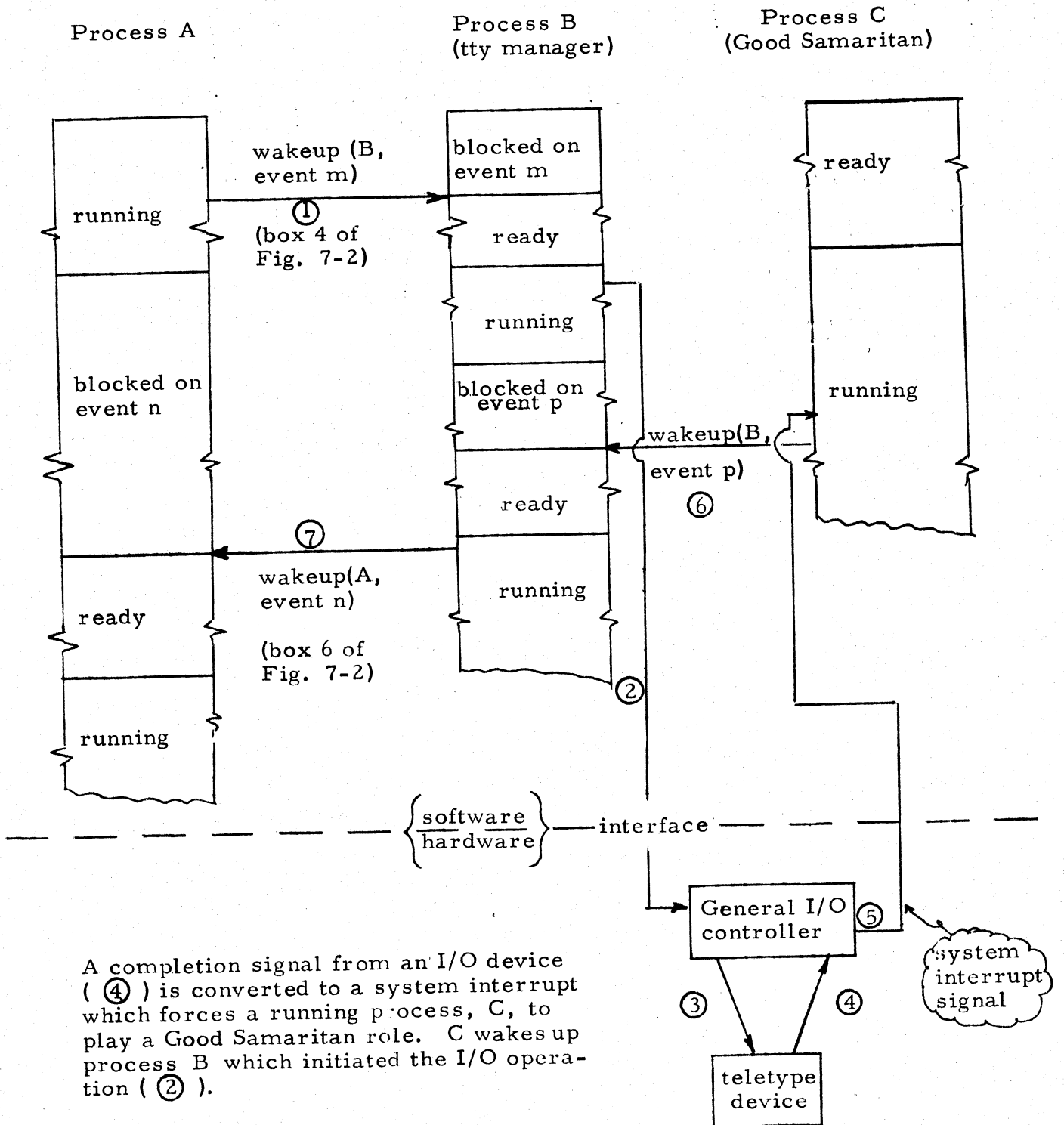
The test in box 2a of Figure 7-3 (and in box 9a, if it were drawn in a similar fashion) suggests an essential characteristic of meaningful communication among coexisting (and cooperating processes. A process may receive more than one message or signal (from one or more processes). Each legitimate signal could have different significance to a receiving process. In most instances it is essential that the reawakened process be able to identify the sender and the nature of the message, if proper interpretation of the "reawakening" is to be made.

For example, consider our hypothetical tty manager as the receiver of messages. Such a process could serve not just one user, but all users who are using teletype consoles for output or input. In that event, the loop (boxes 6 through 10) of Figure 7-2 would clearly be an oversimplification. When awakened the tty manager must identify which user process is sending a message and moreover which type of message it has received, so that it can act accordingly, i. e., so it can resume a read loop to initiate more output on the teletype or so it can resume a write loop to initiate more input to read a buffer from the teletype—for some process. While the manager is not running, it must somehow be in a position to receive such messages in an orderly way, so that when again in the running state the message(s) received in the interim can be properly interpreted.

Multics provides a general mechanism known as the IPC (interprocess communication facility) to achieve the transfer of messages (signals) between processes. "Receipt" of messages can occur while the process is in any execution state, because the sender, using the IPC facility, can place a message in a shared data base which the receiver will examine and interpret at a later time, also with the aid of the same IPC facility.\* Subsystem designers will have little interest in the details for transmitting messages between user processes and

---

\* The shared data base and its manipulating procedures are necessarily in ring-0 for protection reasons.



A completion signal from an I/O device ( ④ ) is converted to a system interrupt which forces a running process, C, to play a Good Samaritan role. C wakes up process B which initiated the I/O operation ( ② ).

Figure 7-5 Illustrating the conversion of system interrupts to process wakeups

system processes like the I/O driver, since these are entirely controlled by built-in functions of the I/O control supervisory procedures. On the other hand, the same techniques for interprocess communication also apply to subsystems in which two or more user processes must communicate with one another for effective operation. Here, the designer must provide the explicit calls on the IPC facility. For such subsystems, the designer must become more fully acquainted with the IPC. Section 7.5 provides the basics.

A final observation is in order in this introduction concerning interprocess communication. This has to do with the distinction between data communication and control communication. In the example of Figure 7-2, the data passing into and out of the write buffer may be regarded as data communicated between the two processes. The messages transmitted by the makeup function and examined by the block function, though also data in one sense, nevertheless serve as control communications in that their net effect, like stop and go signals, permit the starting-up of a blocked process. There is an analogy between these two types of communication and two types of computer instructions. Control communication corresponds to a " $\left. \begin{array}{l} \text{read} \\ \text{write} \end{array} \right\}$  memory" type of instruction.

#### 7.2.4 Other Control Functions of the Traffic Controller

The Traffic Controller contains modules needed for the purpose of creating processes, for destroying them, and for halting processes in anticipation of destroying them. Additionally, the TC is able to cause the loading of a process. Loading a process amounts to placing in memory a limited number of selected segments, page tables, and other information whose guaranteed presence in memory is essential if the process is put into the running state. We shall refer to this set of process information as the minimum core image, (MCI). Among the components of the MCI are the APT entry for the process, a ring-0 descriptor segment, and a special ring 0 process state segment named PDS (Process Data Segment).

Generally speaking, the subsystem designer need pay little attention to these essential supervisory functions, since they must be carried out as a matter of course in normal operations. Thus, during log-in a so-called "working process" is automatically created for the user and during logout that process is destroyed. Moreover, it is also the responsibility of the Traffic Controller to see to it that a

process reaching the top of the ready list has a minimum core image.\* In other words, loading of the MCI is supervised on behalf of the working process whenever necessary.

With all this machinery for "managing processes" already necessary (and available) as supervisory functions, it is not surprising that the Multics design is aimed at giving a sophisticated user the opportunity to exploit some of these functions for his own purposes.

Two types of user applications are envisioned. The first is almost fundamental because of its relationship to console debugging. The second relates to the user's management of a subsystem in which one process spawns others.

1. Stopping a process so as to debug it

During a console session the user will often find cause to stop a process now in execution (running, ready or blocked). He may notice, for instance, that his working process is in an endless (or undesirably long) output loop, suspect an endless computation loop is in progress, or for other reasons wish to halt the process and take stock of the situation, i. e., enter into certain on-line debugging activities. The Multics design makes it feasible to carry out such console debugging by providing the user a simple-to-use facility to accomplish the following:

- a. Cause his current working process to be "stopped".
- b. Cause a new working process to be created and activated on the user's behalf which will now respond to his console commands. (No new login is necessary, mind you.) The new process can now be used to "inspect" segments such as the stacks of the stopped process, using debugging procedures that execute in the newly created process.
- c. When Multics is fully implemented, a user will be able to achieve steps a and b simply by pressing the quit button and then issuing a "save" command on his console. The effect will be to signal an always-coexisting process, called the answering service, and asking it to do these chores.

---

\* The loading task is carried out, in fact, by waking up a system loading process to do the job. This special process is of the highest priority and is itself always loaded. Consequently, the Traffic Controller's request for the loading of a process will receive relatively quick response. We will add more detail to this frame of reference in Section 7.4.

- c. If, after inspecting the stopped process, the user deems it "resumable", possibly after "doctoring" one or more of its segments in some fashion, then he may destroy the new or current working process and resume (put back into the ready state) the old working process. This step, of course, implies that the console will be reattached to the resumed process.

Step c would be accomplished by typing a simple command.

- d. Alternate decisions might be either to save the old process for future resumption or save certain of the temporary files of this process. Saving a previous working process or any of its temporary parts is simple enough and is achieved by typing a simple command. Providing the system support for the practice of resuming a saved process at a much later time may await further system research and development. This is because such practice implies that the "state" of the Multics supervisor, and the Multics library will, at the time an old process is resumed, be sufficiently like its original state to make resumption of the process meaningful. But how can we be sure that the option to resume the process at some later time can be successfully exercised? This problem is intrinsic to all information utilities whose supervisory code and system libraries evolve at some finite rate. Presumably a process will be stopped and saved in a state where its execution point lies outside of any supervisory procedure or library code. If supervisory or system library code has been altered in the interim, resumption of the process at a later time can be "effective" if and only if re-execution of said altered system codes is (Case a) not required, is (Case b) functionally similar, i. e., the altered procedures retain their original interface (e. g., same argument list, external references, etc.), or (Case c) old copies of the altered code can be retained for use at process resumption time.

Clearly, Case a is purely a lucky circumstance. As a general solution, Case c implies serious and perhaps insurmountable problems in system design and system management. This approach requires that in the limit it would be necessary to furnish a user with a complete copy of an older Multics system to resume a saved process.

Only Case b implies serious promise for a general solution. When systems like Multics or their successors reach a sufficiently stable state of development, it is not inconceivable that a contract between system administrator and a subscriber will imply a commitment to provide (over a certain time span) a stable functional interface to the supervisor and other system-supplied code, i. e., sufficiently stable to provide effective resumption of long-saved processes.

(Needless to say, we are "not there yet", either in Multics or in any other system of like objectives.)

One remark is appropriate after considering this example. The TC is not only designed to assist in the stopping of a process, but is also able thereafter to recognize such a process, by marking the execution state (in its APT entry) as stopped. We see then that there are in actuality a total of five execution states that are recognized:

running, ready, waiting, blocked, and stopped.

A process is marked "stopped" as explained below, when it has no use for a processor and has no expectation of needing one, i. e., is not expecting a wake-up. Putting a process in the stopped state prevents later wakeups received from cooperating processes from accidentally restarting a quit process.

## 2. Stopping a process in the general subsystem case

A special entry is provided in the TC which can be used by one process (A) to stop another process (B). The form of the call is:

```
call stop (id_of_process_B);
```

Of course, the call must be and is quite privileged. No user can be permitted to use it in an indiscriminate fashion or the entire system would quickly collapse. On the other hand, with proper safeguards, it would be very useful to grant such permission for user process A to stop (and possibly even then destroy) user process B, provided, however, A and B were related to one another in a meaningful way. For example, Multics provides a subsystem with the capability for one process to spawn one or more other processes (much as certain system processes must be capable of doing). Each such spawned user process would then belong to the same "process tree" (as those that have a common ancestor user process). In a fully implemented version of Multics a supervisory module, such as the TC, would conceivably be able to recognize members of the same process tree\* so as to screen requests of the form:

```
call stop (B);
```

---

\* The coding scheme that will permit this recognition is not yet finalized as of this writing.

A subsystem designer will, therefore, be able to write code which makes an initial working process A spawn processes B, C, . . . , etc. Any of these may be coded to spawn others. All belong to the same process tree. Any one process might reach a decision to stop another in the same tree on the basis of "cross talk", i. e. , interprocess communication between or among two or more of those coexisting within the group. We leave to the imagination of the reader the possibilities for subsystem design that are implied by virtue of these capabilities. Further consideration of this topic here would be premature prior to an examination of the Multics interprocess communication facility (IPC) itself, which is introduced in Section 7.5.

### 7.3 CORE RESOURCES EMPLOYED AND MANAGED BY AN ACTIVE PROCESS

In this section we shall examine core requirements of an active Multics process during various phases of its existence (from the time the process is created until it is destroyed). We will also consider the system implications of these core requirements in the multi-process environment in which all coexisting processes compete or tend to compete for core. (When a user logs in, a process is created on his behalf by a pre-existing system process which responds to the login command. The newly created process is registered in the Active Process Table (APT) and then given active status by loading into core the page tables for a small group of key segments.\* The process normally remains active until the user logs out, at which time the process is destroyed.†)

The execution state of a process not only characterizes the process as a competitor for a processor, but it also suggests implicitly how a process functions as a competitor for core.

A running process will attempt to and in fact may capture as much core as it needs. It will be restricted from doing so while it is executing only by virtue

---

\* The initial Multics implementation rigidly couples activation and deactivation of a process with its creation and destruction. A more flexible connection, e. g. , dynamic activation, is also possible, and in principle, could be added to the System at some later time. Dynamic activation would make it convenient for Multics to support subsystems which exhibit a large number of only occasionally-used processes.

† A user's process can spawn other processes, which become "active" in the same way.



of competing demands of processes which are simultaneously executing on other processors. In a single-processor environment, the longer a process is allowed to execute without interruption, the larger can its "core holdings" become.

A ready process competes for a processor mainly with other ready processes. At any given time, a ready process will be queued in some fashion dependent, among other things, on the respective priority levels of the set of ready processes. The specific queuing discipline is discussed in Section 7.4. As a competitor for core, a ready process is a "loser". Because it is not executing, a ready process is unable to initiate the acquisition of core. Attrition can occur in its core holding due to the demands made by the executing process(es). A new page is brought into core for an executing process at the "expense" of some other page. The Multics algorithm for selecting pages to be "thrown out" in favor of new ones, is such that least recently referenced pages tend to be preferentially selected for removal. Thus, in principle, the longer a process remains in the ready state (hence, not making references to its pages, to the core-resident portion of its address space) the more likely it will suffer a loss of pages. In actual practice, a small group of processes will at all times belong to the class of so-called "eligible processes". At any given time a small number,  $n$ , of active processes are allowed by the Traffic Controller to compete for a processor. These are the eligible processes. The number  $n$  is recomputed periodically by the Traffic Controller.  $n$  is a measure of the capacity of the system to effectively serve its clients short of an overload. An eligible process will normally gain sufficiently frequent use of a processor, such that its most recently referenced pages will not be purged during any period of its residence in the ready state.

A waiting process is one that cannot make immediate use of a processor because it is waiting for a so-called system event to happen, for example, the arrival of a page into core, the request for which was initiated while this process was last executing. As a competitor for core, a waiting process is a loser in roughly the same sense as a ready process. However, because a process that goes to the wait state for a system event is expected to remain there for only a brief and predictable period of time, it is allowed to retain its eligibility during this period. It also retains its favorable position in the queue. If sufficiently favorable, the readied process may in fact pre-empt the processor. In general, residence in the ready state will tend to be for short periods, hence short periods

between execution states and hence minimal attrition of its core holdings.

A blocked process is waiting for a (process) event whose time of "arrival" is not in general predictable. As a competitor for core, all we can say is: the longer the process remains blocked, the more of its (non-shared) core-resident pages will be removed. If a process is blocked long enough, all its unshared pages, including its descriptor segments, will be paged out. Page tables for most of these segments will also be deleted. Care is taken, however, in the system design to retain the page tables for several critically important segments (such as for the process state segment (PDS) and descriptor segments)\*. By retaining page tables for these important segments, the process does not have to thrash about, taking an undue number of segment and page faults when it next re-enters the running state. Page tables and pointers to these segments and to all other segments which remain active are retained in a special wired-down table area known as the System Segment Table (SST)†. Of course, there will always also remain a "core residue" consisting of pages and page tables of the shared supervisory segments and perhaps also some shared library segments (both wired-down and otherwise).

A stopped process is the same type of core competitor as is a blocked process. The only difference is that a stopped process may be destroyed at the request of another process. †† In the course of being destroyed, segments that are categorized as temporary (and filed in its "process directory"), are deleted along with their file branches. These include the KST, the various individual and combined linkage segments of the process, etc. The space occupied by these files, on whatever device they happen to occupy, is returned to the free storage pool of that storage device by the storage allocating routines of the system. Thus, core space occupied by pages (and page tables) for temporary segments is immediately reclassified as free. (Free space is dispensed first in satisfying page requests of executing processes. Only after this pool is exhausted

---

\* Strictly speaking, even these page tables would be removed if the process were deactivated. The decision to deactivate (i. e., unload) a process would be made by a system control process on behalf of a process that has been blocked for a lengthy period. A deactivated process is still "remembered" in a process table of the system so that it can be reactivated when necessary.

† The detailed architecture of the SST is given in BG. 2.

†† Moreover, a stopped process may be deactivated.

will other pages be removed.) Blocks occupied by the remaining pages and page tables of the destroyed process will be reused as needed by the system's paging algorithm.\*

From the above discussions we see that, as a process cycles through its execution states, there also is a marked tendency for its core holdings to ebb and flow cyclically. It is to the user's advantage that in the typical cycle the pages lost in going from the "crest" reached during its current execution state to the "trough" reached during its next wait or block state, should not include those which will be needed during some reasonable interval after the process next re-enters the running state. There are two obvious reasons:

1. Increasing the number of page faults taken by a process will perforce increase the total elapsed time from the beginning to the end of a process, since the time spent in the wait state increases.
2. The processing of each page fault adds several milliseconds to execution time and this time is (and probably must be) charged to the faulting process.

The system's scheduling and page removal algorithms are designed to help keep this potential loss of efficiency from becoming severe. As previously mentioned, the Traffic Controller limits the number of users that may compete for a CPU at any one time. The eligibility restriction has the effect of keeping the paging activity in the system at a tolerably low level (as a percent of CPU usage). The by-product effect of this restriction is that the average number of pages allotted to each competing process can be kept above some desirable minimum value. If the number of eligible processes is made too low, however, eligible processes may compute efficiently (i. e., for very long periods between page faults), but there will be too much idle time when the page faults do occur, while at the same time the system's response to the non-eligibles may become unacceptably poor. As experience in the use of Multics grows, the sophistication of the various controls employed by the Traffic Controller can be expected to increase in the resultant direction of an optimal balance among these conflicting needs.

---

\* An excellent view of this algorithm can be found in the paper, "A Paging Experiment in the Multics System," by F. J. Corbató, Multics repository document M0104.

### 7.3.1 Minimum Core Requirements of an Active Process

We have already alluded to the idea that there exists in Multics a minimum core commitment for each active process. In this section we will begin to gain insight into the core management aspects of Multics that relate to these memory-resource requirements of an active process.

From an overall view, core can be viewed as composed of three parts:

- (a) wired-down supervisor code (very roughly 25K words as of the summer of 1969),
- (b) system-wide tables and I/O buffers (perhaps 50K words\*), and
- (c) the remainder, consisting of core blocks of a pool that is managed by a wired-down core allocator. These blocks are for the pages of non-wired procedure and data segments (approximately 300K words maximum, in the present configuration at Project MAC).

Of the system-wide tables mentioned in (b), the SST (System Segment Table) is of chief importance in this discussion. This is a table that includes an entry for every active segment in the system and cross-references each of these segments with the processes which presently share them. For each active segment there is also included in the SST its corresponding page table. Saying that a segment is active, therefore, mainly reflects the fact that its page table is currently in core. Segments are limited to 64 pages (1024 words each),† so their page tables are stored as 64-word blocks.††

---

\* Space allotted in these tables is a function of the total core space available and of the anticipated number of active processes permitted in the system at any one time.

† The SST provides page table space for enough page tables (npt) to ensure that there is, in fact, always an excess of page tables over the number of segments (nseg) having a page or more in core. The excess,  $npt - nseg$ , is used to retain page tables for vital segments of all active processes (including those not eligible), and also to retain page tables for segments that are the ancestor directories of all active segments. Experience shows that keeping page tables for these segments in core drastically depresses the number of segment faults (and by-product page faults) that are normally incurred as a result of process switching and as a result of file system operations. This amounts to a form of "preventive maintenance" for system efficiency, since each segment fault currently costs around 16 ms. and each page fault around 6 ms. Clearly, reducing the number of these faults also means reduced delay or latency in the execution of any one task.

††In Chapter 1 of this Guide, we mentioned that the GE 645 hardware was capable of supporting segments of up to  $2^{18}$  or 256K words. However, software considerations have recently resulted in a decision to limit segments to 64K.

In terms of the overall view of core just presented, we now discuss two types of minimum core requirements for an active process.

1. A static minimum, which is the core committed to the active process while it is not running. This is the core needed for an effective transition into the running state. As long as the process remains active (i. e., has an entry in the APT), the static minimum is retained. In the present implementations (Summer, 1969) it is a core space of approximately 230 words and, apart from the 16-word entry in the APT, is drawn entirely from the SST.
2. A dynamic minimum, which is the core needed when the process is running. The additional core space implied in this minimum, while partly drawn from the SST, is mainly drawn from the general pool of 1024-word core blocks. Approximately six blocks are currently needed for pages of several "vital" system-provided segments.

It will be useful to enumerate components that make up each of these minimum "sets" even though continuing implementation improvements may make these details rapidly obsolete. Table 7-1 lists the static set and Table 7-2 lists the dynamic set. The following discussion provides some functional explanation of these two sets. It is not intended as a complete discussion, only as the beginning of a plausible explanation for the curious.

First we shall suggest the reason for maintaining, in active status, the three listed segments of Table 7-1. The system is designed so that resumption of the running state will result in references to each of these segments, and hence segment faults to them either should or must be avoided. At least two of the listed segments will almost immediately be referenced when the process resumes the running state. These are: the process state (data) segment, <pds>, and the ring-0 descriptor segment. (In the current implementation, the process state segment includes data areas previously allotted to various ring-0 segments, e. g., <pdf>, <rtn\_stk>, <stack\_00> and <kst>. In making references to the first two of these, neither segment faults nor page faults to certain pages therein\*

---

\* The process executed last while in the Traffic Controller (ring 0). It was using page 1 of the ring-0 descriptor segment and a special stack (called the "process concealed stack") that is kept in the process state segment. This segment also contains vital process state information such as the current ring number, which must be accessible to the system while it is processing page faults.

Upon returning from the block (or wait) entry of the Traffic Controller, the process will still be in ring 0, so the ring-0 stack, also embedded in the process state segment, will also be needed.

TABLE 7-1

The Static Set  
(Minimum core requirement while an Active Process is not running)

| <u>Item</u>                                                                                                                                     | <u>Number of Words</u> |
|-------------------------------------------------------------------------------------------------------------------------------------------------|------------------------|
| 1. APT entry                                                                                                                                    | 16                     |
| 2. SST entries for segments:<br>(72 words/entry, including<br>a 64-word page table)                                                             |                        |
| a. ring-0 descriptor segment <desc_0 >                                                                                                          | 72                     |
| b. "process state segment"<br>(combines in one segment<br>the functions of<br><br><pds><br><pdf ><br><stack_00 ><br><rtn_stk >, *and<br><kst >) | 72                     |
| c. ring-i descriptor segment <desc_i >                                                                                                          | 72                     |
| Total                                                                                                                                           | 232 words              |

---

\* Discussion in earlier chapters of this Guide mentioned these components as separate segments. Condensation into one segment resulted from implementation improvement in the Summer of 1969.

TABLE 7-2

The Dynamic Set  
(Minimum core requirement while an Active Process is Running)

| <u>Item</u>                                                                 | <u>Number of Words</u> |
|-----------------------------------------------------------------------------|------------------------|
| 1. Items listed in Table 7-1<br>(3 SST entries and 1 APT entry)             | 232                    |
| 2. SST entries for segments:<br>(72 words/entry, including<br>a page table) |                        |
| process directory (ring 0)                                                  | 72                     |
| ring-i stack                                                                | 72                     |
| ring-i combined linkage segment                                             | 72                     |
| Subtotal                                                                    | 448                    |
| 3. Pages for Segments                                                       |                        |
| (a) ring-0 descriptor segment (1)*                                          | 1024                   |
| (b) process state segment<br>(see Table 7-1) (2)*                           | 2048                   |
| (c) ring-i descriptor segment (1)*                                          | 1024                   |
| ring-i combined linkage segment (1)                                         | 1024                   |
| ring-i stack (1)                                                            | 1024                   |
| Total                                                                       | 6600                   |
| (in round numbers)                                                          |                        |

\* These pages are preloaded before the process begins to run. All other pages listed here are brought in as a result of page faults. Pages of the process state segment and the first page of the ring-0 descriptor segment, once loaded, are treated as wired-down so long as the process is in the running state.

can be tolerated, so actual pages for these segments will be read (back) into memory, as necessary, before the process is switched to the running state.

Typically, the Traffic Controller was entered either as a result of a system interrupt, while the process was executing in some ring other than 0, or indirectly, as a result of a call from another ring *i*. The Gatekeeper must be able to effect a return to ring *i*, implying need for the presence of the page table to (and a page from) ring *i*'s descriptor segment.

Once a process enters the running state, the pages of its core holdings (beyond those (four) that are preloaded for it) will rapidly expand in number. Segment faults will be incurred in referencing other segments such as those listed in Table 7-2, item 2.

A segment fault results in the creation of an SST entry (72 words) and a page request for the referenced page. In a "busy" system new SST entries can be created only at the expense of old ones which are in some sense candidates for removal. Just as new pages replace ("old") pages that have not recently been referenced, new SST entries replace those for segments which have no pages remaining in core. (Of course, certain types of SST entries, such as the per-process segments listed in Table 7-1, are not candidates for removal while the process is active.)

As shown in Table 7-2, a process will tie down a minimum of six SST entries and a minimum of six pages while it performs even the simplest of tasks. Additional core space is needed for its non-supervisory procedures and data segments. Normally, the pages and SST entries in Table 7-2 will be referenced so frequently that the system removal algorithms will never select them as candidates for removal while the process is eligible. Presumably, the same will be true for frequently-referred-to pages of the user-provided segments of a process.

### The Working Set

Following Denning\* we shall refer to the Table 7-2 list, plus the other pages and page tables of the user segments that are being referred to very

---

\* Peter J. Denning, "The Working Set Model for Program Behavior," Communications of the ACM, May, 1968, Vol. 11, No. 5, pp. 323-333. Also, "Resource Allocation in Multiprocess Computer System," May, 1968, Project MAC Technical Report 50 (Ph.D. Thesis of P. J. Denning).



frequently, as the working set of a process. The page removal and SST-entry removal algorithms of Multics are expected to "honor" the working set in the sense that its components tend to remain in core over the period of time the process is executing. During this time, demands by the same or coexisting processes for a large number of less-frequently-needed pages and their SST entries can also be satisfied without seriously affecting the working set (or working sets of the eligible processes).

### 7.3.2 The System Segment Table and Shared Segments

This subsection and section 7.3.3 describe some of the inner workings of the file system's key modules and data bases used in creating and managing the Multics virtual memory. The material is provided mainly for the sake of completeness.\* It is certainly not essential in the flow of ideas for this chapter. These subsections do, however, help one to appreciate some of the challenges that have faced the Multics System designers and show why the success of some subsystems may well depend on how successfully its designer has minimized the load (segment and page faults) that the subsystem has placed on the file system.

Since space for entries in the SST is limited, it will be the usual case that some segment must be deactivated so that another may be activated. Deactivating a segment, therefore, means relinquishing SST table space for its SST entry. Each such entry consists of an 8-word "Active Segment Table" (AST) entry and its associated 64-word page table. An active segment becomes a candidate for deactivation if it satisfies these conditions:

1. The "wired-down" switch in its AST entry must be OFF.
2. If this is an AST entry for a directory, its inferior count, i. e., the number of its immediate descendents which are active, must be zero.

Among all those entries that satisfy the above conditions, select the segment whose page table shows the fewest pages now in core.

---

\* A more complete discussion (very readable and interesting) of this topic may be found in the paper, "The Multics Virtual Memory" by Bensoussan, A., Clingen, C. J., and Daley, R. C., Second ACM Symposium on Operating System Principles, Princeton University, Princeton, New Jersey, October 21, 1969.

When the entry for such a segment is selected for deactivation, the corresponding SDW for the process which lists this segment in its address space must be located and marked appropriately with segment-missing bits. (This process is termed in some of the Multics literature disconnecting a segment.) In this way one is assured that a subsequent reference to this segment will incur a segment fault and thereby invoke mechanisms to recreate an AST entry and page table. Having put a "stop" in the appropriate SDW, Segment Control is then free to proceed with the construction of the new AST entry and page table, and (call Page Control to) page-in the referenced page. The faulting SDW word is then altered appropriately and made to point to the newly constructed page table. (This process is termed connecting a segment.)

If we consider that the old segment may have been shared, then removal of its AST entry and page table implies the alteration of an SDW in the descriptor segment of each process that shares it.

Recall that SST cross-references every active segment with the processes that share it. Segment Control, by proper use of the SST, is therefore able and is sufficiently privileged to identify each process that currently shares any given segment and additionally determine its segment number in each of these processes (i. e., determine the segment pointer in each sharing process). To be more precise, the cross-referencing design of the SST permits Segment Control to get at and alter SDW's in each descriptor segment of every process that shares the segment that is being deactivated.

Suppose we picture that process A is deactivating a segment <s> that is shared by processes B, C, D, etc. Clearly, the job of marking with segment faults a sizeable number of SDW's (i. e., disconnecting a sizeable number of segments) cannot be done instantaneously. This means that some care must be (and is) taken to prevent page-faulting references to <s> from being serviced by processes B, or C, etc., while A, still executing in Segment Control, is attempting to deactivate <s>. To permit the freedom for B or C, etc., to request a new page in <s> at this time is to invite chaos.\* Process A prevents

---

\* This is because such references, if permitted, would imply that Page Control would be working at cross purposes for different processes. For one process it would be attempting to add pages for <s> (and remove page faults in the page table words of <s>'s page table), while for another process it would be attempting to reset the page table so it can be used for another segment.

this confusion from happening by setting certain flags and locks at key places in the SST at the start of its deactivation task. The result is that no other process sharing <s> will then be permitted to do more than idle in a loop, should it fault to a page of <s> while <s> is being deactivated. When fully completed, and all SDW's have been set with segment faults, the flags and locks are reset, permitting the sharing processes once again to reference <s>. The first such reference will incur a segment fault that will then result in a new activation of <s>. More details for those interested are provided in the accompanying footnote.\*

### 7.3.3 Handling of Segment and Page Faults

Segment faults and page faults are relatively costly in execution time and wherever possible the subsystem designer should be on the lookout for ways to avoid triggering these faults unnecessarily. Before examining ways to avoid these faults (see Section 7.3.4) it is a good idea first to get a feeling for how these faults are handled.

7.3.3.1 Segment faults currently require on the order of 15 to 20 ms of cpu processing time. In addition, segment faults will usually trigger page faults and possibly other segment faults as a by-product since both the segment fault handling procedure and the data bases it looks at are not wired down. The steps taken are roughly as given below. For purposes of illustration we shall picture that the fault is taken for a segment <t> in ring i. Here we imagine the branch for <t> is found in a user directory whose path name is >w\_dir\_dir>John.

1. Consult the KST entry whose index is the same as the segment number of <t>. (t# is determined from the saved machine conditions.) Since the KST is itself always active, there will be no segment fault incurred in referencing it, but since the KST is a paged segment, a page fault may be induced before a reference to the desired KST entry is eventually achieved. Segment Control will obtain from the KST entry the segment number and offset of the directory branch for <t> in the segment <John>.

---

\* If another process takes a page fault in the segment being deactivated, Page Control will notice the set flags and will properly interpret what is happening. It will then "tinker" with the process so that it will believe it has taken a segment fault instead of a page fault. This is accomplished by altering the SDW of the segment that has incurred the page fault so it will cause a segment fault when next referenced. Following this adjustment Page Control simply returns, whereupon a repeated execution of the faulting instruction will cause a segment fault. When the process next attempts to solve its segment fault problem, it will be forced to wait on a lock (of the parent directory) which has been set and will remain set until the process doing the deactivating of the segment has finished its job. Whereupon, the flags and locks are reset, permitting other processes to again activate the deactivated segment if necessary.

2. Appropriate information needed for activating  $\langle t \rangle$  is then copied from its branch if  $\langle t \rangle$  is not already active.\* In referencing the branch, another segment fault may be induced if  $\langle \text{John} \rangle$ , although guaranteed to be active (by the ancestor-is-always-active rule), is not connected to its page table, i. e., has fault bits set in its SDW.† However, the particular page wanted from  $\langle \text{John} \rangle$  may be missing, thus inducing a page fault. The data copied from the branch is used to create a new AST entry and page table. Page table words (PTW's) are set with missing page faults and file map data. That is, the address field of the PTW is either set with pointer information for the page's address in auxiliary storage or is set to null (for page numbers that are either beyond the current length of the segment or that correspond to pages yet to be created).
3. After the page table is constructed and the new AST entry is cross-referenced to the process requesting it, the SDW words are appropriately set in the ring 0 and ring i descriptor segments, pointing to the new page table.

7.3.3.2 Page faults currently require on the order of 3 to 7 ms of cpu processing time. We have already suggested the type of tasks that are involved in handling page faults in earlier discussions, so we will not go into much more detail on the matter here. Briefly, when Page Control is called to get a page, it is handed, via the faulting machine conditions, a pointer to the faulting page table word (PTW). Moreover, since the page table address in which the PTW is found has the same index as the corresponding AST entry, the latter's address in the SST can also be determined. The AST entry would then be consulted to ascertain if it is o.k. to proceed with the fetching (or creation) of the page. (You may recall, this entry may have been flagged to indicate the segment is in the process of being deactivated.) If the PTW address field is non-null, it contains device address information for the wanted page, but if null, it indicates that a page of zero-valued words is wanted.†† Page Control calls on a core-allocating

---

\* Strictly speaking, all references to branches are made by Directory Control on behalf of Segment Control.

† In fact, a recursive sequence of such segment faults can occur in the unlikely event that all parents (except the root) have fault-inducing SDW's (i. e., are disconnected). The recursion ends at the root node because this item is by design always immediately accessible.

†† Space for such "empty" pages is created only when first reference is made to them. It is never created and stored ahead of time. Hence, no page needs to be "transferred" from secondary storage.

routine\* to obtain the address of a free core block. (Such a request can easily trigger a page-removal request if no free core blocks are immediately available.) † Page Control zeroes out this acquired block, resets the faulting PTW to point to the new block, adjusts the AST entry to reflect the new condition of the page table, and returns control to the faulting procedure.

If the pointer in the PTW is not null, Page Control again asks for a core block address, initiates the drum or disc I/O request as appropriate to get the wanted page from secondary storage, and performs its Good Samaritan chores (notifies) as described in Section 7.1, and calls wait in the Traffic Controller. You can see from the foregoing discussion that processing time for a page fault will vary according to several factors. Certainly, the time to create a new page of zeros rather than to bring one in will be short, i. e., of the order of one millisecond (which is about the time required to write a thousand words of zeros). Processing of several milliseconds will be needed in the more complicated cases, where the page removal algorithm must be invoked and an I/O request initiated. Of course, this does not count the actual time spent in the page wait and in the (possibly) subsequent ready states.

#### 7.3.4 Ways to Reduce Segment Faults

By now the subsystem designer reading this chapter should be more than mildly receptive to suggestions for reducing the incidence of segment faults (that are under his control to reduce). Two relatively obvious principles serve as a guide.

1. Because other eligible processes compete for page table space in the SST, a process having a large number of segments will tend to suffer a larger number of segment faults than a process with fewer segments. Hence, a conscious effort to keep the number of segments to a minimum will tend to reduce segment faults.

---

\* Additional details on Core Control and its interaction with page control may be found in BG. 5 and BG. 6. Page Control itself is described in BG. 4.

To remove a page involves invoking the page-removal algorithm about which we spoke earlier, which "fingers" pages that are candidates for removal. Also invoked would be the appropriate machinery to copy out the contents of said removal candidates on to auxiliary storage. Copying of a page is performed only if the respective PTW indicates, via its page-has-been-written bit, that the page has been altered while in core.

2. For a given number of segments in a process it should be possible, by conscious programming effort, to organize a "computation" for a minimum of segment faults. Intuitively, this could be achieved if it were possible to sustain a high enough frequency of reference to the segments that were most recently referenced. In other words, if it is possible to design the process so that it maintains a high degree of locality\* with respect to its segment references, the process will incur fewer segment faults.

Here are several approaches the subsystem designer can take to reduce or limit the number of segments of a process:

- (a) Avoid specifying multiple rings unless necessary because each ring in which the subsystem executes automatically adds a number of segments to the process, e.g., a descriptor segment, a stack segment, a combined linkage segment, and, when used, a signals segment and its special linkage segment.
- (b) Bind† procedure segments that belong to the same ring and bind where feasible, data segments of the same ring that are to have the same access controls.
- (c) Proliferation of procedure segments can be limited by a conscious effort to define internal functions and procedures, i. e., those that are defined within the body of external functions. Algol, PL/I and MAD provide good facilities for defining internal functions. FORTRAN does not.
- (d) Use internal static and automatic variables (i. e., that will be placed in the ring-i stack or (combined) linkage segment) whenever possible, in deference to creating separate (external) segments for variables.

Here are several approaches the subsystem designer can take to increase the degree of locality of his process:

1. If the subsystem is multi-ringed, try to confine the computation within one user ring (or within as few rings as possible) for as long as possible.
2. Avoid frequent use of loops in which there are explicit calls to the ring-0 supervisor.

---

\* A thorough discussion of the locality concept is given by P. J. Denning in a paper entitled "Thrashing, Its Causes and Prevention", Proceedings of the 1968 Fall Joint Computer Conference, Vol. 33, Part 1, pp. 915-922.

† Refer to BX.14 for details of binding.

3. Avoid signaling across rings entirely, or reduce the frequency of such signaling, i. e., avoid executing calls to < signal > that incur searches across ring boundaries for the active handler. (This is a reference to discussions in Chapter 5. If you have not read those details, pay no attention to this remark.)

#### 7.3.5 Ways to Reduce Page Faults

Alas, there are no revelational remarks that can be made on this subject! Following the logic in preceding discussion on segments, it is clear enough that a process with fewer pages will, in the long run, generate fewer page faults. What is really wanted is the recipe for minimizing page faults when the number of pages in a process is already at its minimum. A high degree of locality (within the pages of) each segment is wanted and this is a property which only the individual programmer can, with his conscious effort, attempt to achieve. In some cases this will be easy and in other cases very difficult.

Source code for a procedure can be examined (or re-examined) in search of ways to regroup sections of code, especially loops, so that execution is "resident" within the fewest pages for the longest period. To do this it may require that the programmer identify points in the source code that correspond to "page breaks" in the target code. If there were a high enough payoff for this type of activity, compilers might be coded to optionally print page-break markers right on the program listings. In this way, it would be possible to avoid inspection of the target code in most cases. (I know of no compilers now operating in Multics that provide this service.) Unfortunately, any gains from such attention to page-breaks could easily be lost if the segment is bound with others in an effort to reduce segment faults.

#### 7.4 ASSIGNMENT OF PROCESSOR RESOURCES

The assignment of processors to processes that need them is a central management problem in any information utility. In Multics, two management functions are, in fact, to be simultaneously fulfilled in the second-by-second, minute-by-minute solution to this assignment problem. These are time sharing decision-making and multi-programming decision-making. Both functions quite naturally influence the kind of system response that can be expected in the execution of subsystems developed by users. Introductory details, hopefully adequate for the needs of the subsystem designer, are provided in this section.\*

---

\* The principal MSPM references are Sections BJ.5, BJ.6 and BJ.7.

The time-sharing function guarantees each user a chance to gain an equitable share of processor time. The term "equitable" is explained in the next subsection. Roughly speaking, sharing is always among those users that have the same priority. For reasons we shall see shortly, a user's priority needs may well vary with time. An ideal time-sharing system should therefore anticipate (or correctly guess) each user's current need for processor time and accord him a (higher or lower) level of priority that is consistent with this need.

Principally because of core memory limitations, however, a processor cannot be effectively time-shared with an unlimited number of equal-priority processes. The multiprogramming function is therefore restricted so that the processor is assigned in fact to a sufficiently small subset of the "most deserving". The subset, called the eligibles, is chosen small enough so that the work that is done for each member is effective work, and is not degraded, for instance, by thrashing. An implication of the "subset-of-eligibles" idea is that the processor may occasionally be forced to idle for very brief periods when and if all eligible members happen into page-wait or other system wait status at one time. (Occasional idleness is preferable to the alternative of adding another process to the list being multiprogrammed. The latter approach would greatly increase the risk of thrashing. If this occurred, the cost of recovery would prove greater than the small amount of deliberate idle time.)

#### 7.4.1 Time-Sharing Philosophy

To understand the basis for the scheduling algorithm used in Multics, one begins with a crude model in which every user is an "interactive user"; his process consists of executing a series of normally short spurts of computation, e.g., commands. If the execution time for each command were short, invariant, and known in advance, then a sensible scheduler might allot each user the time  $q_k$  that is needed to execute command  $k$  to completion. Users would be queued on a single list and permitted to execute in FIFO fashion. No timer run-out interrupts would be required.

A few commands may be fixed in duration, but, for most, their duration are functions of their arguments, such as the FORTRAN command whose argument is the program being compiled. Nevertheless, it is useful to pursue this line of reasoning because it provides us with a useful conceptual basis from which to understand the more realistic scheduler used by Multics.



## A Conceptual Model

Let  $q$  be some (albeit fuzzily defined) average duration for commands in the system. Suppose current experience indicates that  $q_0$  time units is an appropriate approximation to  $q$ . If the scheduler were initially to allot each user an amount of time  $q_0$ , then a sizeable fraction of users would complete their tasks before their time "ran out". We accept as a premise that the system should, in the default case, be designed to favor users who execute short commands by according such users a relatively high priority. Let each command be classified into one of  $n$  categories (numbered  $1, 2, \dots, n$ ) according to its duration. Commands in category  $1$  would respond to short durations with priority level  $1$  (highest priority), while lengthiest commands in category  $n$  would be in priority level  $n$  (lowest priority).

Our next step is to associate with each priority level a separate queue. If a user wishes to execute a command that falls into category  $j$ , where  $1 \leq j \leq n$ , his process would then be added to the queue numbered  $j$  to wait his turn. Following the management principle that higher-priority jobs should be executed before lower priority jobs, we promulgate the following default rule for the conceptual model scheduler to follow.

No entry on queue numbered  $j$  shall be considered until all higher priority queues are empty. This rule discourages users from executing long-duration commands while the load is high. There is also presumed to be a mechanism for overriding this rule, so that under certain circumstances a user can request an increase in priority level for his command. (He might, for instance, be requested to attract the attention of the supervisor by pressing a special button on his console. This completes our first model.

## The real model (Multics)

Here we shall realistically presume that a command's duration is in general not known in advance. However, we shall take the attitude that the unknown duration must be determined if meaningful scheduling is to be achieved. Some type of adaptive technique suggests itself. In the Multics scheduler, it is assumed that every process arriving on the ready list for the first time deserves a position on a high priority queue on grounds that the command to be executed will be a short one. Associated with the queue is some fixed time allotment  $q_0$  (say one second). When a process on this queue is picked to compete (in the eligible-for-multiprogramming sense), the command may run to completion. If so, the

process will then call block before the allotment  $q_0$  is used up. On the other hand, if the allotted time is exceeded, execution will be halted by a timer run-out mechanism. The command is then assumed to belong to the next category. The containing process can no longer compete on the first queue, so the process is awarded an additional allotment of time, say  $2 \times q_0$ , and placed at the end of the next lower level priority queue.

Each time the current time allotment is exceeded, execution is stopped so that the command's category can be "reappraised". That is, the process is given an additional allotment, say twice the preceding allotment (e.g.,  $2 \times (2 \times q_0)$ ) and placed at the end of the next lower priority level queue. In this fashion we see how the system learns adaptively about the "true" duration category,  $\ell$ , of a command's activation. When command execution is completed at some priority level,  $\ell$  (or more strictly speaking, whenever an interaction is accomplished) the process is rescheduled, anticipating execution of another command. Specifically, the process is dissociated from queue and reassociated with the top priority level queue with allotment  $q_0$ .

A price has been paid to learn a command's "true" duration category when more than  $q_0$  time units are needed. This price amounts to premature processing of a variable portion of its total execution. During this time the process is allowed to compete briefly with more favored commands. Thus, if a command's execution is in the range  $q_0 \times 2^r \leq \text{duration} \leq q_0 \times 2^{r+1}$ , then at least half of its processing ( $q_0 \times 2^r$ ) will be completed at a perhaps undeservedly high priority.\*

#### 7.4.2 Multics Design Details †

The Multics solution to the processor assignment problem is made

\* If we assume there are cost vectors C and R such that  $c_i$  is the charge to execute for  $q_0$  seconds at priority level  $i$ , and  $r_i$  is the cost of rescheduling the process from priority level  $i$  to  $i + 1$ , then the excess cost, EC, to learn the true category,  $s$ , of a command's particular activation is at most

$$EC = \sum_{i=1}^{s-1} (c_i - c_s) \times 2^i + r_i$$

Of course, a user would not necessarily be charged in this manner. EC is a quantity that is mainly of conceptual value in understanding the nature of the Multics scheduler.

† Modular design of the central supervisor has made it feasible for the Multics system architects to try many variations of the basic traffic control and scheduling algorithm whose concepts have been sketched earlier. The details presented here are, to the best of the writer's knowledge, consistent with a version of the scheduler used in the Spring of 1969 and are useful for illustrative purposes. Details of the actual scheduler may differ from those described here, but in all likelihood differences from those described here should be of no significance to the subsystem writer. System designers wishing to learn about the current algorithm are expected to consult the appropriate BJ sections of the MSPM.

conceptually simple when it is discussed with the ready list as the focal point. For then, albeit at risk of some oversimplification, one sees that:

- (1) The decision process that determines where new entries will be inserted in the ready list and what time allotments to give them (scheduling) amounts to fulfilling the time-sharing decision function.
- (2) The decision process that determines which entries on the ready list to run amounts to fulfilling the multiprogramming decision function. (Removing an entry means awarding a processor to the process associated with that entry.)

#### 7.4.2.1 Insertions in the ready list (scheduling)

The ready list may be viewed as a set of  $n$  queues (each a ready list), one per priority level. At the time it is created, each user process is assigned a range of priority levels ( $\ell_1, \ell_2$ ) with initial execution started at level  $\ell_1$ . The range ( $\ell_1, \ell_2$ ) offers some clue as to the type of time-sharing service the process will be given. At any given time a ready process has a current priority level,  $\ell$ , such that  $1 < \ell_1 \leq \ell \leq \ell_2 < n$ . For interactive and absentee user processes, the ranges ( $\ell_1, \ell_2$ ) fall on the scale 1 to  $n$  as follows:

Interactive processes would range from 3 to some value  $k_1$  while absentee processes would range from some value  $k_2$  to  $n-2$ , with  $k_2$  having a lower priority level than  $k_1$  as suggested by these straddling brackets:

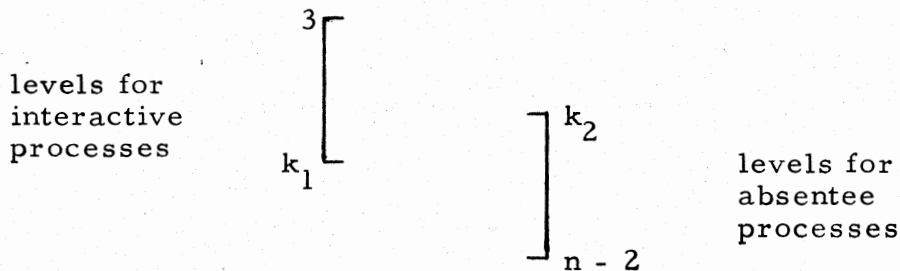


Figure 7-6 summarizes these ideas and reminds us that there exist (and indeed, that there must exist) priority levels both higher than the highest user priority level and lower than the lowest user priority level. There is room "at the top" for special system processes\* that can, if necessary, preferentially

---

\* One such process is the Loader Daemon System Process. This process is awakened by the Traffic Controller for the purpose of "loading" a process that has been identified as ready and eligible to run on a processor. Loading consists of placing in core and wiring down any or all of the starred items of Table 7-2 that are not now in core. To be of any value, the Daemon should run immediately and briefly upon being awakened. For this reason, it must have its own vital segments and pages in core at all times. As soon as the loading is completed, the Daemon calls block and waits for another such wakeup.

|             |                                                  | Remarks                                                                                                                                                                                                              | Eligibility     |
|-------------|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|
| Level 1     | Special system processes                         | This level is reserved for certain system processes like the Loader Daemon Process. They always run in short bursts and it is essential that they can almost immediately pre-empt a processor when waked up.         | Always eligible |
| Level 2     | Special system processes                         | Less critical system processes, such as the I/O driver, e.g., for the line printers, which should pre-empt overall user processes, and the System Control Process (commonly referred to as the "answering service"). |                 |
| Level 3     | Interactive users and system processes           | Top priority level for interactive user.                                                                                                                                                                             |                 |
| ...         | ...                                              |                                                                                                                                                                                                                      |                 |
| Level $k_1$ | Interactive users, system and absentee processes | Lowest priority level for interactive users.                                                                                                                                                                         |                 |
| ...         | ...                                              |                                                                                                                                                                                                                      |                 |
| Level $n-1$ | Absentee processes                               | Lowest priority level for absentee users.                                                                                                                                                                            |                 |
| Level $n$   | Idle processes                                   | One idle process is earmarked for (exclusive use of) each processor in the system.                                                                                                                                   | Always eligible |

Note: System processes in the range (3,  $k_1$ ) compete with user processes.

Figure 7-6 The  $n$  priority level queues of the ready list

capture a processor. Likewise, there is room "at the bottom" for an "idle" process that can capture the processor should all ready queues at higher priority levels ever become empty.

Priority level 3 is reserved for processes that have an urgent, but very brief, expected need for a processor. E.g., a process which has initiated a console read command, is willing to give up the processor (go blocked) while waiting for the console typist to type the next input line, but wants the opportunity to resume execution, i. e., to respond, just as soon as the input step has been completed. The console read subroutine is a privileged (ring-0) procedure which, after initiating I/O activity, calls the block entry of the Traffic Controller after turning on a so-called "interaction" switch in the process state segment. Any call to block with this switch ON results in rescheduling the process for the highest user priority queue (level 3).

Lower priority levels would be reserved for absentee processes. These are processes which run without console assistance, i. e., non-conversational with the external environment. Absentee processes are akin to foreground-initiated background jobs in CTSS\*. (The priority range for absentee processes is expected to be  $(k_2, n-1)$  where  $3 \leq k_2 \leq k_1$ . This would provide some straddling of the interactive user's priority range, which is  $(3, k_1)$ . In this way overnight service on relatively short absentee jobs would in effect be "guaranteed".)

#### 7.4.2.2 Eligibility Management - design details

Eligibility management superimposes a needed control on the number of processes, call it nep, that are being multiprogrammed. This control prevents thrashing, since the number nep is, in a sense, an approximation of the maximum number of processes whose complete working sets can fit in the available core memory. There should be no substantive alteration to the efficacy of the general multi-level scheduling algorithm as a consequence of superimposing eligibility control. This subsection and the next one on pre-emption outline the design details.

---

\* For more information on absentee processes, see BQ.2 and BQ.3. Actually, absentee processes are not yet provided for in the current Multics.

The number, nep, which will eventually be a value that can be varied by the system administrator, is a function of available core memory and the number of available processors. At the present time, nep is fixed at 2, but may well be increased when 384K of core is used by the system. (There are also certain special processes that are allowed to and, indeed, must be eligible at all times. As mentioned earlier, these include, for instance, the Loader Daemon Process, and the one or more idle processes.)

APT entries for eligible processes can be thought of as linked in a list such that their positions on this list determine their relative priority to capture a processor. When a running (and eligible) process enters a wait state for the occurrence of a system event, the CPU will be given to the ready process on the eligibles list that has the highest relative (or positional) priority. When the event waited for occurs, the now running process notifies the waiting process by marking its APT entry accordingly (changing its state bit to ready). If the process so notified has higher relative priority, the notifying process (executing in the supervisor, of course) immediately yields the CPU to the notified process. (We refer to this behavior as CPU pre-emption.) In this way the system favors processes that become eligible, one at a time, giving each eligible process when it reaches the lead position the greatest chance to run to completion of its time allotment.

The number, nep, is treated as a system resource that can be allocated among the active processes, somewhat as core is allocated. Processes gain and lose eligibility cyclically. The cycle can be traced as follows:

An ineligible process is made eligible when it appears on the top of the ready list at the time an eligibility vacancy occurs. A vacancy will occur when a process loses its eligibility for any one of a number of reasons:

- (a) An eligible process enters the blocked or stopped state,
- (b) A process incurs a timer run-out interrupt and is rescheduled,  
or
- (c) A process' eligibility is pre-empted. (We explain eligibility pre-emption as distinct from CPU pre-emption, in the next subsection.)

Since a process specifically retains its eligibility when it enters the wait state, it is entirely possible for all nep of the eligible processes to be in the

wait state simultaneously. In this event, there being no eligible processes on the first  $n-1$  ready queues, the processor is given to an idle process.

Each time a process gains or loses eligibility, its APT entry is marked accordingly. When the process loses its eligibility the Traffic Controller selects the next candidate to be marked eligible. If that candidate is not loaded, the TC sends a wakeup to the fast-responding Loader Daemon process to load\* that process, and finally re-enters the blocked state.

#### 7.4.2.3 Pre-emption of Eligibility

The required conditions (for an ineligible process B to pre-empt the eligibility of a running process C are:

- (1) B's priority level exceeds that of C (i. e. , B's queue number is lower than C's),
- (2) The eligibility of a running process C will not be pre-empted unless it has executed at least as much of its present allotment as the higher-priority process B "intends" to run when it captures the processor. Let  $r$  be B's time allotment and  $s$  the amount already used of C's time allotment. The condition to be satisfied is that  $r \leq s$ .

If condition (1) is met, but condition (2) is not, process B must reside on the ready list until (to the nearest time unit) condition (2) is met. Notice that eligibility pre-emption is a necessary (though not sufficient) prerequisite for CPU pre-emption.

For whatever reason a process is pre-empted (its CPU or its eligibility), that process must be immediately rescheduled. A pre-empted process that has a priority level  $k$ , is rescheduled by being placed on top of the queue at level  $k$  with a time allotment equal to whatever time is still unused from its last scheduling allotment.

The net effects are as one would like them to be, namely that a pre-empted process is favorably treated, relatively speaking. Thus, suppose B, at priority level 3, pre-empts C at level 4. C is rescheduled at the top of level 4. If, shortly afterward, B incurs a timer run-out, it will lose its eligibility and be rescheduled at the bottom of the queue at level 4. Each time a process loses

---

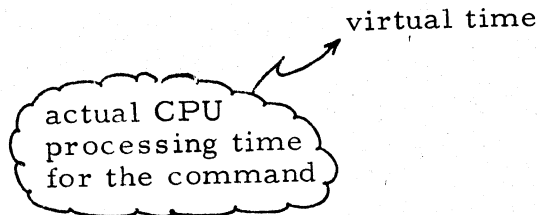
\* Loading, you recall, involves placing and wiring into core the pages listed in Table 7-2.

eligibility, the Traffic Controller immediately tries to fill the created vacancy before identifying the next eligible process to be selected from the ready list. In this instance, the eligibility vacancy will be given to C, and C will be picked to run next, if no other process has been added to the ready list ahead of C while B was running. This, we should note, is precisely the behavior pattern we want. Namely, other things remaining the same, a pre-empted process should be placed in a favored position to recapture the processor when the pre-empting process next loses its eligibility. Because its position is favorable, the pre-empted process is unlikely to lose a significant part of its working set. No core losses to speak of will be experienced if the user process is pre-empted by a high priority system process, since these processes do not "consume significant quantities of core."

#### 7.4.3 Expected System Response

With the benefit of the foregoing discussions on scheduling, eligibility control, and pre-emption, a user is now ready to anticipate the type of system that can be expected for his process. We summarize these ideas here.

During slack load periods, e.g., 2 a.m., Sunday morning, users will be satisfied with the system response to most commands. The ratio, R, is defined as

$$R = \frac{\text{elapsed time for completion of a command}}{\text{virtual time}}$$


will approach 1.

During peak load periods, however, users who execute commands of long duration are likely to observe that execution appears to proceed rapidly at first, then slower and slower, as suggested by the solid curve in Figure 7-7. Thus, commands that under light system loads might require three minutes of processor time may require hours to complete in extreme cases. This is because the user's process sinks to lower and lower priority levels as virtual execution proceeds. A point is likely to be reached where the process is rarely or never



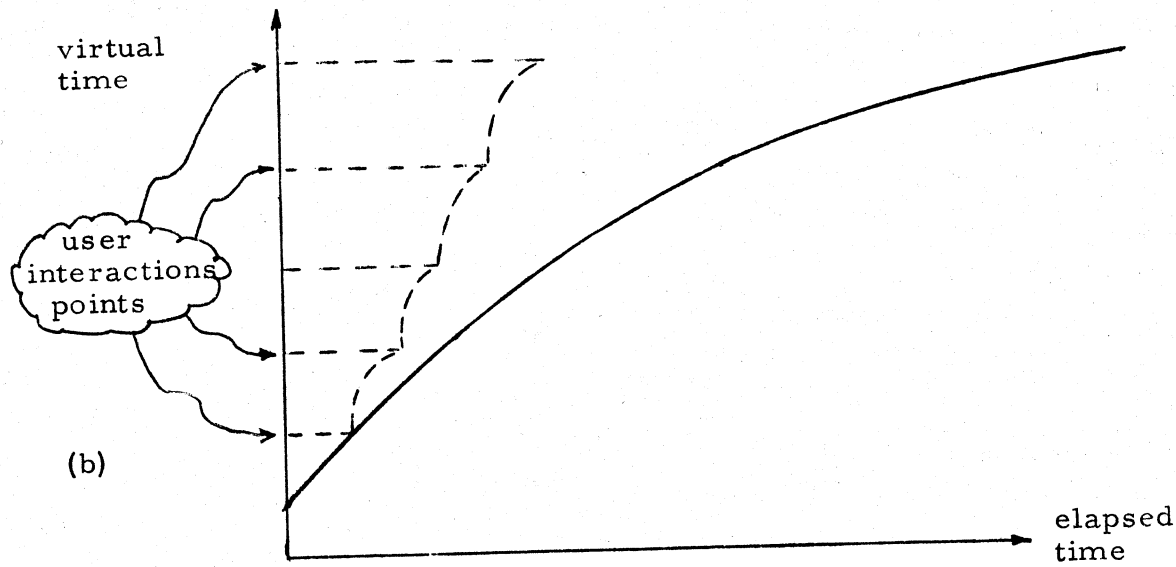
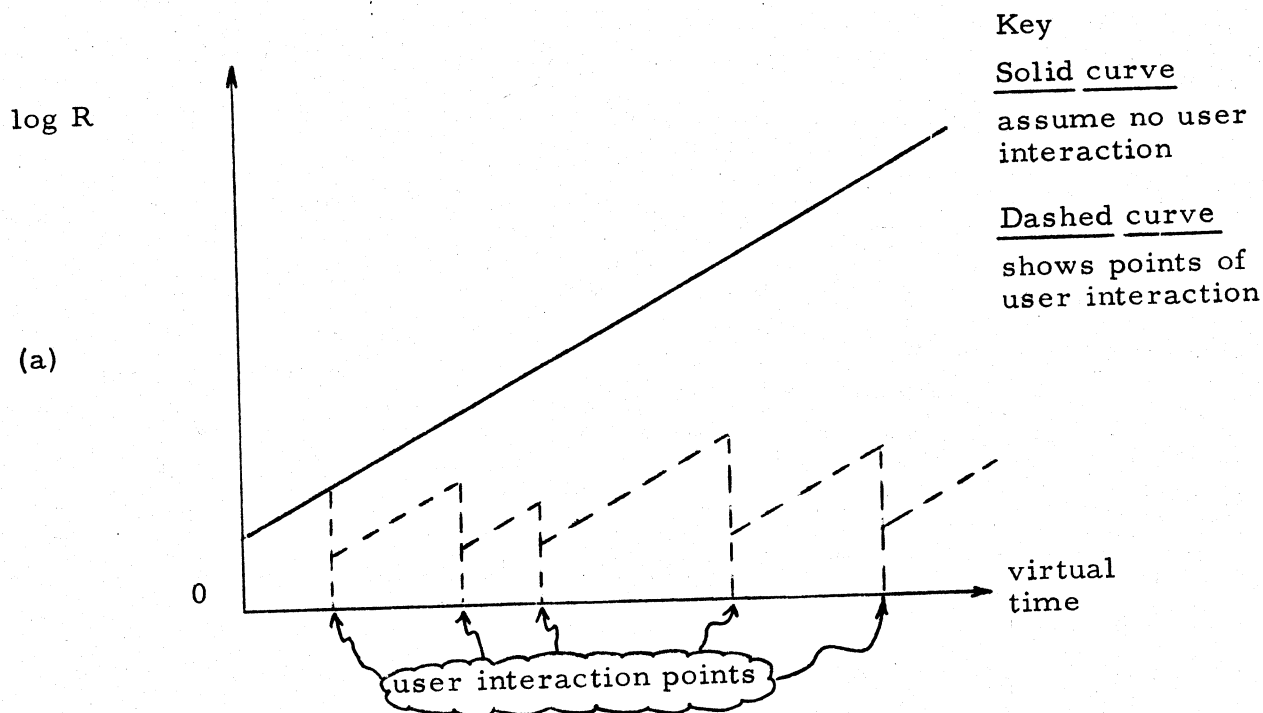


Figure 7-7 Variation of R (the ratio elapsed time to virtual time) as a function of virtual time.

picked to execute simply because there are one or more processes that are always queued at higher priority levels. Only after the peak load subsides (fewer users logged in) will the long-duration command again have a good chance to be picked for execution.

### User Recourse to slow response

A user that is impatient with this response is encouraged to restructure his process to run in absentee fashion. Alternatively, he can force one or more "interactions" each of which will have the effect of rescheduling his process (back up) to level 3. The simplest way to force an interaction is to press the "quit" button on the console and then type

start (carriage return)

after the console responds to the quit signal by typing

ready.

The effect of hitting the quit button is to cause the user's quit responder to call the Listener to accept the user's next input. Since a console read-in always sets the interaction switch in <pds> to ON, there is a consequent rescheduling to level 3. The effect of using the quit button is suggested by the dashed line in Figure 7-7. Users will learn that the use of the quit button for purposes of speeding up execution will prove to be an unpleasant way to use the system. (It turns out to be not much fun hitting quit and typing "start" repeatedly, especially if it must be done a large number of times.)

## 7.5 INTERPROCESS COMMUNICATION

### 7.5.1 The Nature of Processes and the Nature of Their Intercommunication

In the overview of this chapter (Section 7.1), we initiated a discussion of interprocess communication--although without taking a serious look into the nature of their intercommunication. Here we provide a more thorough discussion of this topic. We shall also provide for interested readers a description of the tools available (and how to use them), i. e., for the operation of subsystems that comprise two or more intercommunicating processes.

We already know from previous discussions that processes may properly function to achieve a common goal only if they communicate as senders and as

receivers of messages via shared data bases. Hence, an understanding of communication mechanisms, e.g., information content of messages, "mailboxes," message switching and routing techniques, validation and protection of messages, etc., is likely to be essential for detailed subsystem design.

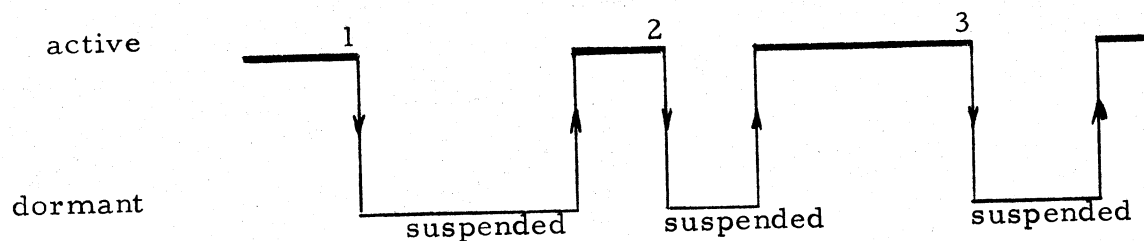


Figure 7-8 Time-line characteristics of sequential processes

A process A that wishes to alert a process B of an event of interest to the latter must send a wakeup and a message to B. The message must be formatted in a standard fashion, so that its source (Process A) and its target (Process B) and the specific receiving point within the target's address space (i. e., a "channel" or mailbox) may be recognized, validated, and accessed. The receiving process must in turn exercise known scanning habits which will find, properly interpret, and dispose of messages that have been received in its mailboxes. Details are discussed below.

### 7.5.2 Communication Mechanisms

A process may reach any number of suspension points, as suggested in Figure 7-8 where, for example, three such points are marked. In a very simple process, these suspension points, however many in number, may occur at the same program point, i. e., the process loops through this program point. For  $n$  transits of the loop, there will be  $n$  suspensions. In each case, then, the nature of the event waited for will be the same, and even the sender of the notifying message that permits resumption of the active state may be the same. In a more general case, however, we can expect suspension points at different program points, e.g., possibly occurring in different procedures, and even in

different rings. Suspensions at different program points will, in general, be for different reasons. (Such program points will hereafter be referred to as wait points.) This means that the nature of the events waited for at different wait points will in general be different. Moreover, the corresponding sending processes are not likely to be the same either. Basic to all of this discussion is the following: For each distinct wait point of a process it must be assured by prior arrangement that some process (at least one) will send a wakeup message when the looked-for event has occurred. Suppose then a process that has several distinct wait points reaches one of them. The process must now be prepared to wait, if necessary, for a particular message to arrive (possibly from a particular sender). Of course, there need not be any waiting at all if the wanted message has already arrived. However, whether or not waiting is required, either upon reaching the wait point or after being awakened following a suspension, the suspended process must be sure that the right message has been received before continuing with its active efforts.

What is involved in searching for the right message? If one pictures that the receiving process has a single mailbox for all messages, then determining if the right message has arrived is simply a matter of scanning the contents of one mailbox--either by an indexed or by an associative search, depending on the data structure of the mailbox. But, wait! Won't protection considerations dictate that there may be at least one mailbox per ring of the process? The answer is yes, but since the reasons are secondary to our main line of thought here, the explanation is left to our footnote\*.

Are there cases where one mailbox per ring would be insufficient? In principle perhaps the answer is no, provided each message fully identified the wait point, the sender, and the exact time that the message was sent. In practice, however, the Multics designers have chosen to implement the system in such a way that several different mailboxes per ring are available. For example, a process may contain a programmed wait point that asks to wait for receipt of

---

\* A one-mailbox approach would mean that a ring-32 procedure, for instance, could read mail intended for a ring-1 procedure! Clearly, this is unacceptable. So we must picture a process arranging for the receipt of mail in different, ring-related mailboxes. In this way, a ring 1 procedure can scan mail in a ring-1 mailbox (and even in a ring-32 mailbox if desired), but a ring-32 procedure would be able to scan mailboxes only in rings  $\geq 32$ .

a message in any of a given list of designated mailboxes. We shall discuss these ideas in more detail later. We mention them here only to motivate the notion that a process may have what amounts to sets of mailboxes (each mailbox possibly empty), one set per ring.\* Clearly, each mailbox must bear a unique designation within the receiving process so that a sender can transmit his messages to their proper destination.

#### 7.5:2.1 Messages, Mailboxes (event channels) and Transmission

The technical name used in Multics for a mailbox is event channel. An event channel is uniquely designated by a 72-bit identifier †. This name is generated by the system as a result of executing a user-written subroutine call for the creation of an event channel.††

##### Origin of the Message

By added convention, every message originates as a 72-bit item of arbitrary content (set by the sender). (However, in the course of transmitting the message, system routines expand it with self-identifying information.)

A message is sent in the form of a call to the hard-core system routine, hcs\_\$wakeup:

```
call hcs_$wakeup (receiving_process_id,  
                  channel_name,  
                  message,  
                  code);
```

---

\* There is one byproduct benefit that comes from the implementation decision to have multiple mailboxes per ring. Let the distinct wait points in some ring-r of a process A be designated as wp1, wp2, ..., etc. Suppose the wait at each of these points is for a message from a correspondingly different process, e.g., from processes p1, p2, ..., etc. Prior arrangements between the process pairs (A, p1), (A, p2), (A, p3), etc., for the sending of messages to A need not be fully coordinated in the sense A is not forced to give (or to divulge) to p1, p2, p3, etc., the very same mailbox name. One can regard this flexibility as an advantage in that there may be less risk of confusion if separate senders are asked to send messages to different mailboxes, with each mailbox having a different meaning.

† The substructure of the event channel name includes three items, a ring number, a key (52 bits), and an ECT address. The key is a unique name representing the wall clock time at which the event channel was created for this process. The ring number identifies the ring in which the receiver expects to examine messages placed in this channel. Received messages are saved (until inspected) in a one-per-ring segment called an ECT (Event Channel Table). The ECT address is simply the offset within this segment at which the possibly-queued messages for this channel may be found. A channel is in effect a FIFO list. Details of the ECT data structures should be of no interest to users. They may be found in BJ.10.02.

††Details on how to create event channels may be found in BJ.10.01.

Note that although the actual text of a message is small and fixed in size, it is large enough to be used as a pointer to messages of arbitrary size. We defer momentarily answering the obvious question, namely, how will the sender know both the process\_id of the receiver and its receiving point (the event channel name). This matter is taken up in the section entitled Setup for Interprocess Communication.

The hcs\_\$wakeup routine makes some simple (routine) checks on the first two arguments so that if they are obviously erroneous\*, due to programmer error, the caller can be alerted if he chooses to examine the returned error code. After this partial validation, the Traffic Controller's wakeup entry is called, at which point steps are taken to forward the message to the intended receiver, as outlined below.

1. If the message indeed has a receiver, then it must be possible to match the receiver process\_id with the id of one of the processes that now have entries in the Active Process Table. Failure to find such a match means that the message is meaningless. Such a case results in an appropriate error code being reflected to hcs\_\$wakeup's caller. (Note that a post-office analogy to this case is -- "addressee unknown at this address -- return to sender".)
- 2a. A message aimed at a bona fide target process will be copied into a ring zero system table where it is properly augmented with "truthful" information about the sender. The system table (central storage) is called the ITT (for Interprocess Transmission Table). The receiving process will later fetch the message out of this table.
- 2b. The last step is to call the Traffic Controller at its entry point wakeup to wake up the receiving process.

The above steps are summarized in the Figure 7-9 flow chart. Note that hcs\_\$wakeup serves as the user's only interface with the otherwise inaccessible wakeup entry in the Traffic Controller. Protecting this entry from direct user calls simplifies the logic of the Traffic Controller which, because it is locked to all other processes when entered, must be kept as simple (and fast-executing) as possible.

---

\* For example, if the process id or certain of the subfields of the 72-bit channel name are zero, this is clearly an error.

Receivers must be protected against receipt of false messages, whether accidental or intentionally sent. Certain information about the sender is therefore added to the copied message that is placed in the ITT. This information, which is of critical importance for the protection of the receiver, consists of the sender's process id and the validation level of the sender's call to hcs \$wakeup (i. e., the ring in which this call was made). The user cannot be trusted to transmit these items accurately. Figure 7-10 shows the message format as stored in the ITT. The Interprocess Transmission Table is a wired-down system table in ring-0 that is large enough to hold messages for all known processes. The table is organized as a set of message queues, one per process. The head of each queue is pointed to from a fixed position in the APT entry for the corresponding process, so that when any process re-enters the running state in the Traffic Controller, as a result of being awakened, it can quickly determine if there have been any messages deposited in the ITT on its behalf since the last time it ran.

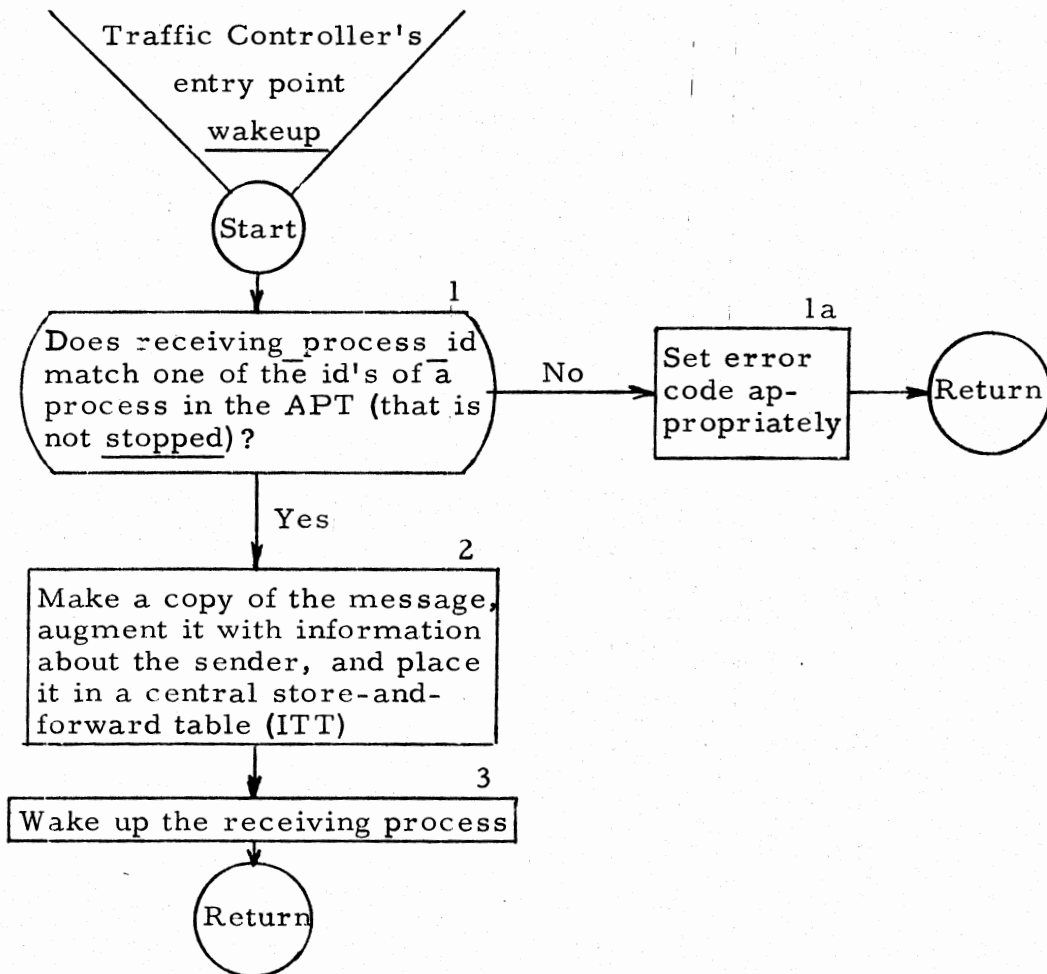


Figure 7-9 Some details of the Traffic Controller's entry point wakeup

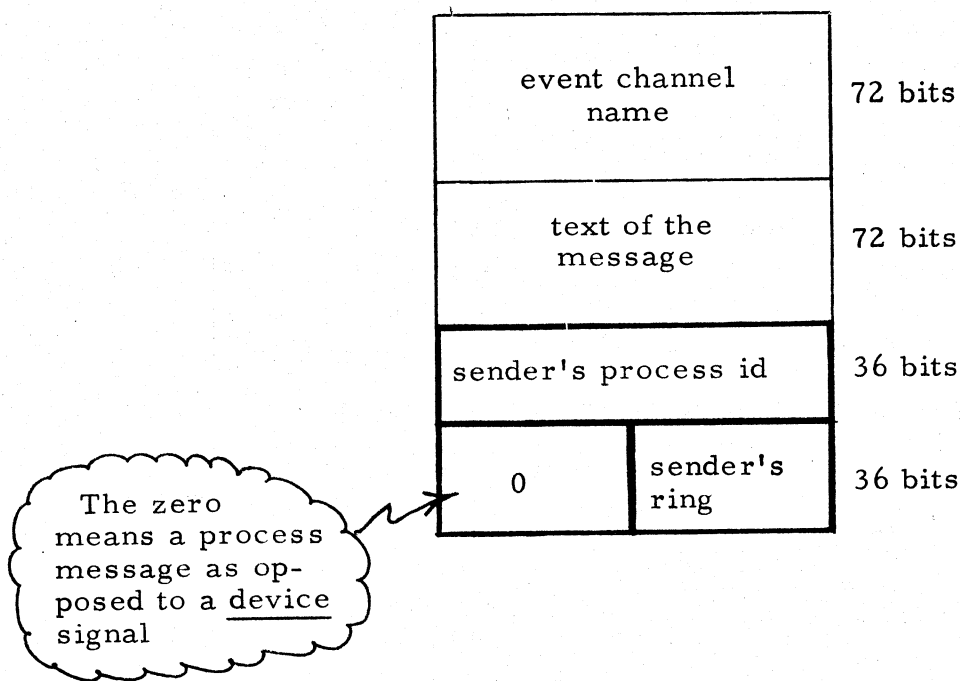


Figure 7-10 Message format augmented with information about the sender as placed in the ITT

Getting the message from the central store-and-forward point to the receiver

So far we have considered mainly the mechanics of sending a message as far as a central forwarding center. In a postal system analogy such as shown in Figure 7-11, this is the halfway point, e.g., a regional post office. No ordinary citizen is able to walk up to this center and ask for his mail. Nor, by analogy, can the Multics user expect to get his mail by attempting to read messages while they are still in the ITT. He needs help in moving the messages to data areas that are ring-accessible for his purpose. While the post office automatically pushes the mail through to its receiver from the central p.o., without any special coaxing, the Multics analogy is somewhat different. Here some initiative is always taken by the receiver to pull the message(s) out of central storage and to place them into the individual ring-accessible event channels of the process. Recall that a receiver's process will have a table of one or more event channels (an ECT) in every ring in which there occurs a distinct wait point in that process.



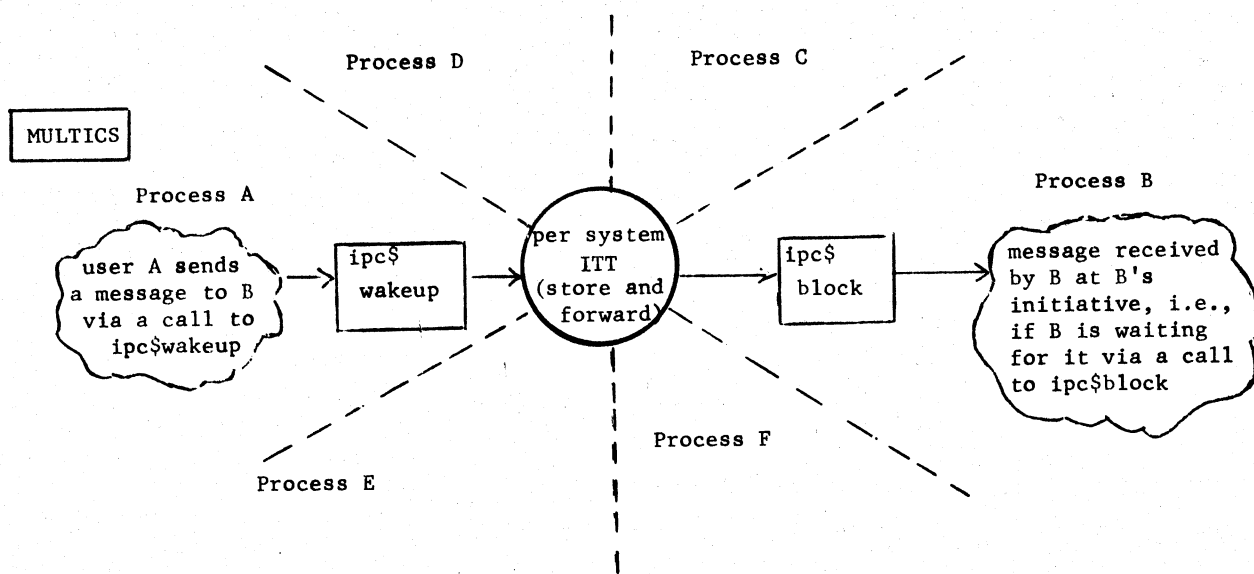
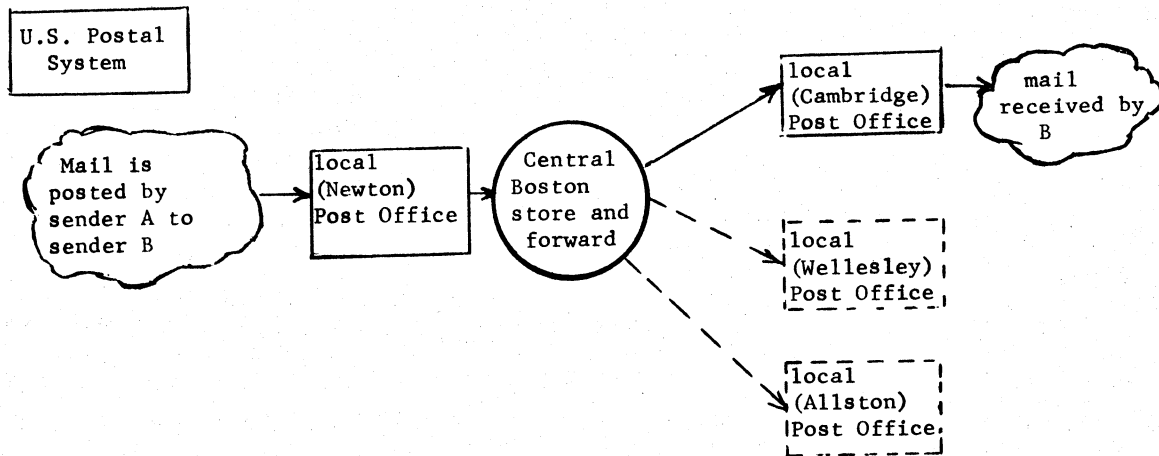




Figure 7-11 Postal system analogy to Multics interprocess message transmission

A wait point is always programmed as a call to `ipc$block` which is the entry point in the so-called wait coordinator, the heart of the interprocess communication facility. A user program should call this entry point whenever it must enter the blocked state while awaiting the receipt of a message.

The form of this call is

```
call ipc$block (wait_list_ptr, message_ptr, error_code);
```

 A list of one or more event-channel names

 A pointer supplied as an input argument that specifies the location where the caller expects to receive a message which he can examine

When called, `ipc$block` scans the event channels in the list pointed to by the first argument. Scanning of the channels is done in the listed order, and if lucky enough to find a message in one of these channels, `ipc$block` transfers the first such message found into the location given by the second argument, and returns to its caller. The message that is actually transferred consists of the six-word message whose format was shown in Figure 7-10, augmented by a seventh word consisting of the wait-list index. Thus, if a message is found in the third of eight channels on a wait list, the seventh word of the returned message will have the value 3.

Note that `ipc$block` has been executing in the ring of its caller. (`ipc$block` has ring brackets which are (1, 63, 63).) Consequently, this procedure does not have ring access for scanning central storage (i. e., the ITT) which may have received one (or more) of the desired messages. Therefore, `ipc$block` is forced to call a privileged routine in ring-0 (at an entry point `hcs_$block`). This routine in effect transfers all valid messages that have accumulated in the ITT event queue for this process. Each message is placed in the event channel that is designated in that message. Invalid messages, such as those whose channel names do not match existing channels in the receiving process' ECT's, are summarily discarded. If, in the course of making these message transfers, not a single message was transferred into a ring  $\geq$  the validation ring (i. e., that of `ipc$block`), then it is clear the wanted message cannot have yet been received. Hence `hcs_$block`, which is fully privileged to do so, calls the corresponding entry in the Traffic

Controller to give away the processor.\* If, on the other hand, at least one such message was moved to a ring that is accessible to ipc\_\$block, then hcs\_\$block will return so that the former can again scan its given list of channels in hopes of finding the wanted message. The chain of calls we have just discussed is summarized in Figure 7-12.

If we were to follow the return path from the TC backwards toward the point of call to ipc\$block, it becomes easy to see how a fresh message, received at ipc\$wakeup can be thought of as being forwarded from central storage to the appropriate event channel of the receiver.

It should be recalled that when an awakened process finally recaptures a processor, effective execution will resume as a return from the block entry in the TC to its caller, hcs\_\$block. The latter then "transfers" all newly arrived messages from its ITT event queue into the appropriate event channels. If no messages were transferred into rings  $\geq$  that of the caller, ipc\$block, then the process cannot have received the message it was waiting for. Hence, hcs\_\$block again calls the TC at entry point block to give away the processor. But if at least one potentially suitable message was transferred from the ITT, hcs\_\$block returns to its caller (ipc\$block). (This is how the pulling of messages is done--in the absence of an explicit effort, e.g., a call to ipc\$block, messages for this process can in principle pile up in central storage without ever being drawn out.) Note that the return to ipc\$block is no guarantee of a return to its caller. If ipc\$block finds no message in one of the listed event channels, it simply recalls hcs\$block.

#### 7.5.2.2 Setup for Interprocess Communication†

Here we shall discuss how a sender learns the identification of a receiver process and the identification of that receiver's event channel. For convenience let us adopt the following notation.

---

\* To be absolutely precise about things, there is still a possibility for a last minute "reprieve", if anytime up to the very last instant before giving the processor away a wakeup arrives, control will return to the TC's caller. For more details you could review J. H. Saltzer's Ph. D. thesis or BJ. 3.01 to see the function of the so-called "wakeup waiting" switch.

† For a more basic discussion of this topic, the reader may wish to examine the paper, "The Multics Interprocess Communication Facility", by M. J. Spier and E. I. Organick, submitted for presentation at ACM's Second Symposium on Operating Systems Principles, Princeton University, Princeton, New Jersey, October, 1969.

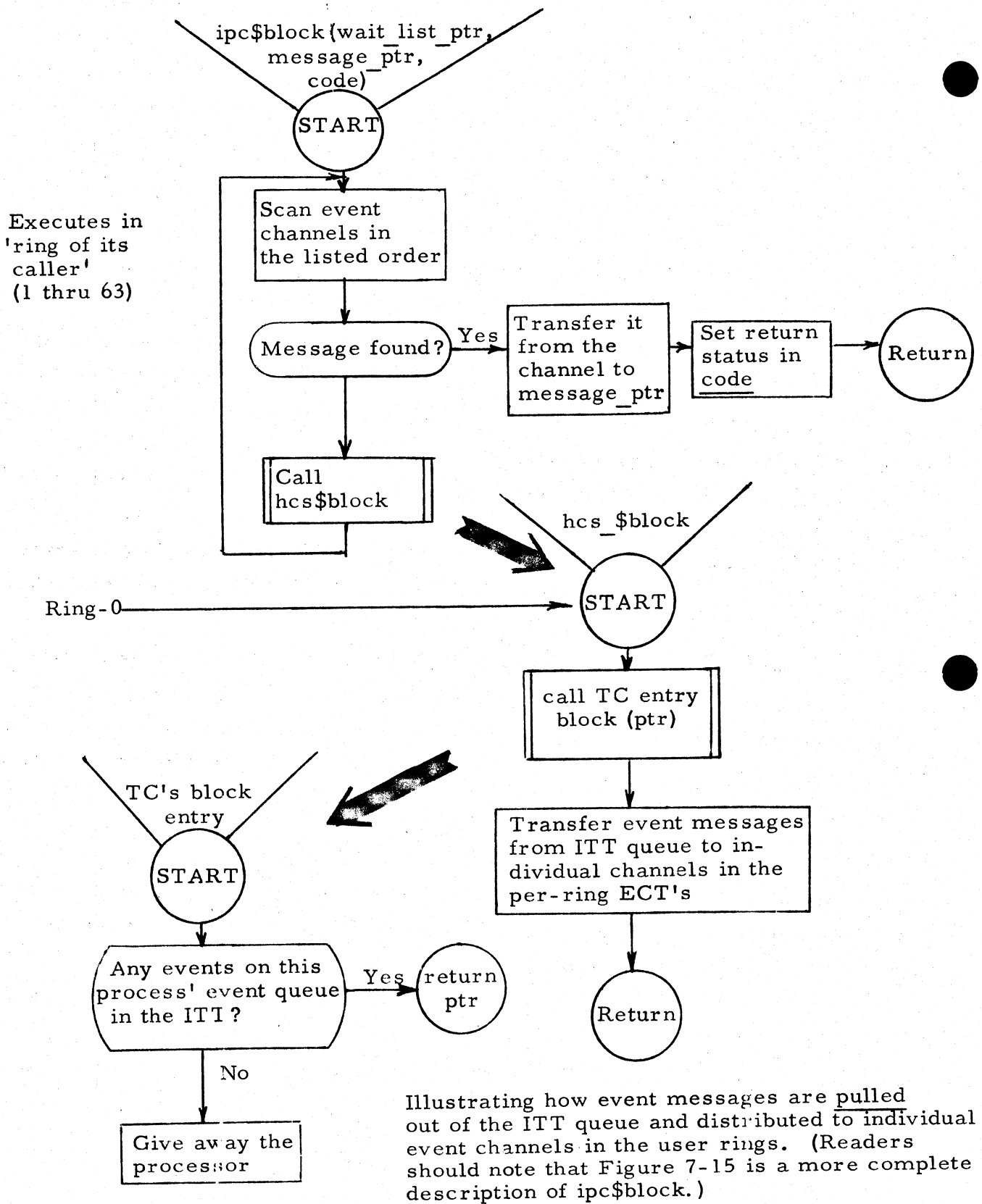


Figure 7-12 The chain of calls: → ipc\$block → hcs\$block → TC block entry.

Let B-to-A setup info be that basic information that is required by a sender process B so that it can send a message to a receiver process A. This information consists of A's 36-bit process id and A's 72-bit event channel name.

Let p(A) and p(B) refer to the people responsible for programming A and B, respectively. (To be sure, they may be the same individual, wearing "two hats".) Clearly, the system-provided message transmission facility (IPC) cannot be employed to transmit B-to-A setup info, else why would setup info be needed in the first place. Note also that p(A) cannot supply p(B) with the B-to-A setup info by telephone or by other direct personal communication until after A has been created. This is because a process id is a clock-dependent unique bit string that is generated by the system at process creation time. Furthermore, A's event channel name, which is also a clock-dependent unique bit string, will not be known until after A's declaration that creates the event channel has been executed. There appears to be only one sensible plan for passing setup info. The plan is as follows:

- (1) p(A) and p(B) agree in advance on the (unambiguous) name of a segment that is to be shared by A and B. Call this segment <shared>. Also agreed upon is an offset within <shared>; call it [setupBA] which is to be regarded as a 3-word mailbox initially set to zero.
- (2) After A and B have been created, and after A has declared (created) the appropriate event channel, A places in shared\$setupBA the desired setup info.
- (3) B fetches the three words at shared\$setupBA, and if non-zero, assumes by convention that the required A-to-B setup information has been obtained.

Note that if A is also to become a sender to B (not just a receiver), then A-to-B setup (as opposed to B-to-A) is also needed. This info can be sent by a similar prearrangement, although in fact a form of "boot strapping" can now be achieved if it is desired, to avoid further use of <shared>. That is, the first message B sends to A can, by further convention, contain the A-to-B setup information.

But how does A know its B-to-A setup info so that it can place it at shared\$setupAB?

#### How A obtains its own process id

When a process is created, one of the temporary segments that is created for it and placed in its process directory is called <process\_info>. This segment

contains special information about the process that can be read by all procedures. Among these is the process id which is stored in `process_info$processid`.<sup>\*</sup> Any user procedure can snap a link to and read this word.

### How A obtains an event channel name

A user creates an event channel simply by calling `ipc$create_ev_chn`. The first of two return arguments in the call contains (upon return) the 72-bit name that the IPC has established for this channel. Henceforth, it is the user's responsibility to keep track of this name.

### Summary

The steps that would be coded by p(A) and p(B) to establish interprocess communication with B as a sender and A a receiver can now be summarized.

1. p(A) codes the following steps in some procedure of A:
  - (a) call `ipc$create_ev_chn(channelBA, code)`; this call creates an event channel which can hereafter be referred to by the name, `channelBA`, because the value of the first return argument is a 72-bit unique id of `channelBA`.
  - (b) assign to the 3-word mailbox at `share$setupBA` values of `process_info$processid` and `channelBA`. Illustrative epl coding is provided in the accompanying footnote.<sup>†</sup>
2. p(B) can code B to pick up the required setup info at any time and use it to send a two-word message to A. Illustrative epl coding

---

<sup>\*</sup> Some consideration is currently being given to merging `<process_info>` with another segment in order to reduce the working set. It is for this reason that this argument was not listed in Table 7-2. If this change should be implemented, however, the name, `process_info`, will still be used in referencing the process id.

<sup>†</sup> In epl, this might be accomplished with coding that relies on a 3-word based structure for a mailbox:

```
dcl 1 mailbox3 based (p),
    2 pid bit (36) /* a process id*/
    2 cname fixed bin (71) /* a channel name*/;
```

Then, the executable code which would follow creation of the desired event channel might look like:

```
p = addr(shared$setupBA);
p -> mailbox3.pid = process_info$processid;
p -> mailbox3.cname = channelBA;
```

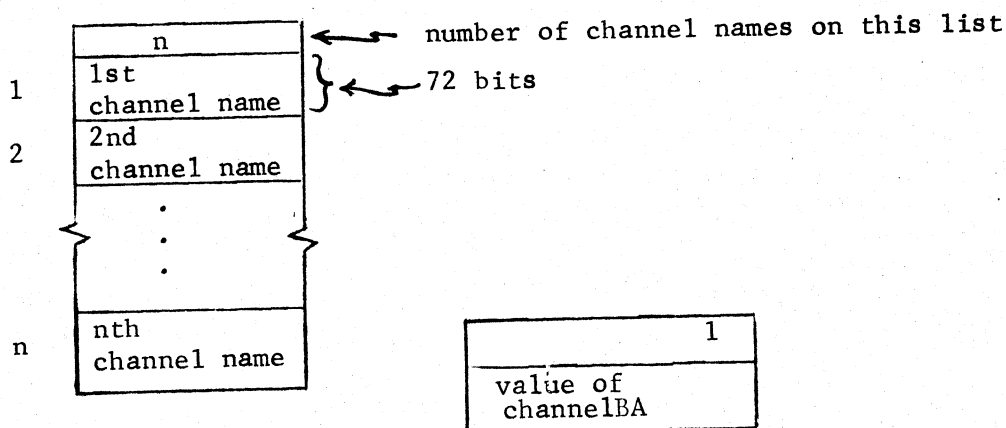
is provided in the accompanying footnote.\*

3. p(A) is now able to code appropriate calls to ipc\$block at various points in A, to wait for messages from B. A call of the form

```
call ipc$block (argptr, msgptr, code);
```

gets the job done on the assumption that the first two arguments are pointers to the base of structures, the first being to a wait list of channel names, and the second to an area of sufficient size for receipt of the message.

The general structure for the wait list is of the form shown in Figure 7-13.



(a) general form of a wait list

(b) appearance of wait list for example in the text.

Figure 7-13 Wait lists. General and Specific.

\* We shall assume that B also uses a declaration for a 3-word mailbox identical to the one in the preceding footnote. Then coding in B might appear as:

```
p = addr(shared$setupBA);
receiver_pid = p -> mailbox3.pid
channel_name = p -> mailbox3.chname
if ¬(receiver_pid = 0 and channel_name = 0)
    then call hcs$wakeup (receiver_pid, channel_name, message, code);
    else call print_error;
```

Here message is a 72-bit message and print\_error might be a routine to print an appropriate error message before proceeding with whatever steps are then deemed appropriate.

### 7.5.3 Programming of a Multi-purpose Process

A Multics process is basically sequential in nature by virtue of the fact that but a single execution point (or point of control) is free to traverse over its address space at any one time. For this reason, it is natural to think of such a process as having a single purpose.

If two or more independent computations are to be performed, albeit related to one another, it is entirely appropriate for the programmer to create a separate process, one per each defined purpose, and have these processes execute in any interrelated fashion that seems appropriate. In fact, this approach is recommended for most initial efforts of this kind.

Subject to processor availability, concurrent computation of the separate but related processes may occur in some fashion, but it is not predictable, of course, since the Traffic Controller and its functions are outside the control of the programmer. In any case, by proper use of IPC, the separate processes (purposes) may synchronize with one another.

It is worth noting, however, that the establishing and maintaining of separate address spaces, one per process, incurs an appreciable system overhead. Such costs are ultimately passed on to the user directly or indirectly. Hence it may well be worth considering under what circumstances it is feasible to coalesce (and condense) the address spaces of several processes into a single, now multi-purpose process having one address space (and one execution point).

Certainly, it is necessary that concurrent pursuit of the separate purposes, i. e., parallel executing of the separate tasks be no requirement. (But, then such a requirement, even without coalescing, cannot be guaranteed in Multics anyway.) Beyond this, the order in which these tasks may be initiated and executed should in some sense be of secondary importance and perhaps be independent of the tasks themselves. This requirement may be satisfied in the case where events external to the process drive the multi-purpose process. That is, IPC messages received by the process are the basis for deciding which task to execute next. Examples of multi-purpose processes are common among system software processes, e. g., answering services, I/O device managers, automatic recorders, automatic file dumpers, etc.

A process that must behave in multi-purpose manner, can in principle be coded using flow chart logic described in Figure 7-14. The basic idea suggested



in the figure is to create  $n$  event channels, one for each of the  $n$  distinct purposes of the process. After furnishing setup information for use of each of the  $n$  channels to the  $n$  senders (not necessarily  $n$  different processes) the process calls `ipc$block` to await one of the  $n$  types of events. Each time an event arrives, `ipc$block` returns with a message. The message is examined (in box 5 of the flow chart) to determine which channel (type of event) has occurred so as to invoke the associated task. When the task is completed, a call is again issued to `ipc$block`.

We have not yet defined what we mean by a task. The simplest idea is to suppose it corresponds to a call to a procedure that is associated with the corresponding event channel. We will be interested in understanding what restrictions are imposed, if any, as to what may go on inside the called procedure. For instance, are calls to `ipc$block` to be permitted from within the associated procedure and/or from any of its dynamic descendents? We shall consider this possibility in the next subsection. For the moment, however, we shall assume such repeated calls do not happen.

#### 7.5.3.1 Event call channels

Note that further logical simplification (from the point of view of the user) arises and a slight increase in efficiency can be gained if the control logic of boxes 3, 4, 5 and 6 are made part of `ipc$block`. At the top-most logic level the process would be characterized simply as the execution of boxes 1 and 2. That is, initializing of channels, transmitting of setup information, etc., followed by a single call to `ipc$block`. There would be no return and of course no repeated calls on `ipc$block`. Of course, it would be necessary to furnish IPC with more information so it can perform its more elaborate job. Basically, this amounts to telling IPC what are the procedures that should be called (invoked) upon receipt of respective messages.

The "simplification" we have been discussing is in fact provided for in Multics by allowing the user to designate event channels of his choice for special interpretation. Event channels marked in this fashion are referred to as event call channels, as opposed to the ordinary event wait channels. Messages found in event call channels are examined and interpreted while the process is executing inside the IPC. Interpretation amounts to execution of a call to the associated procedure.

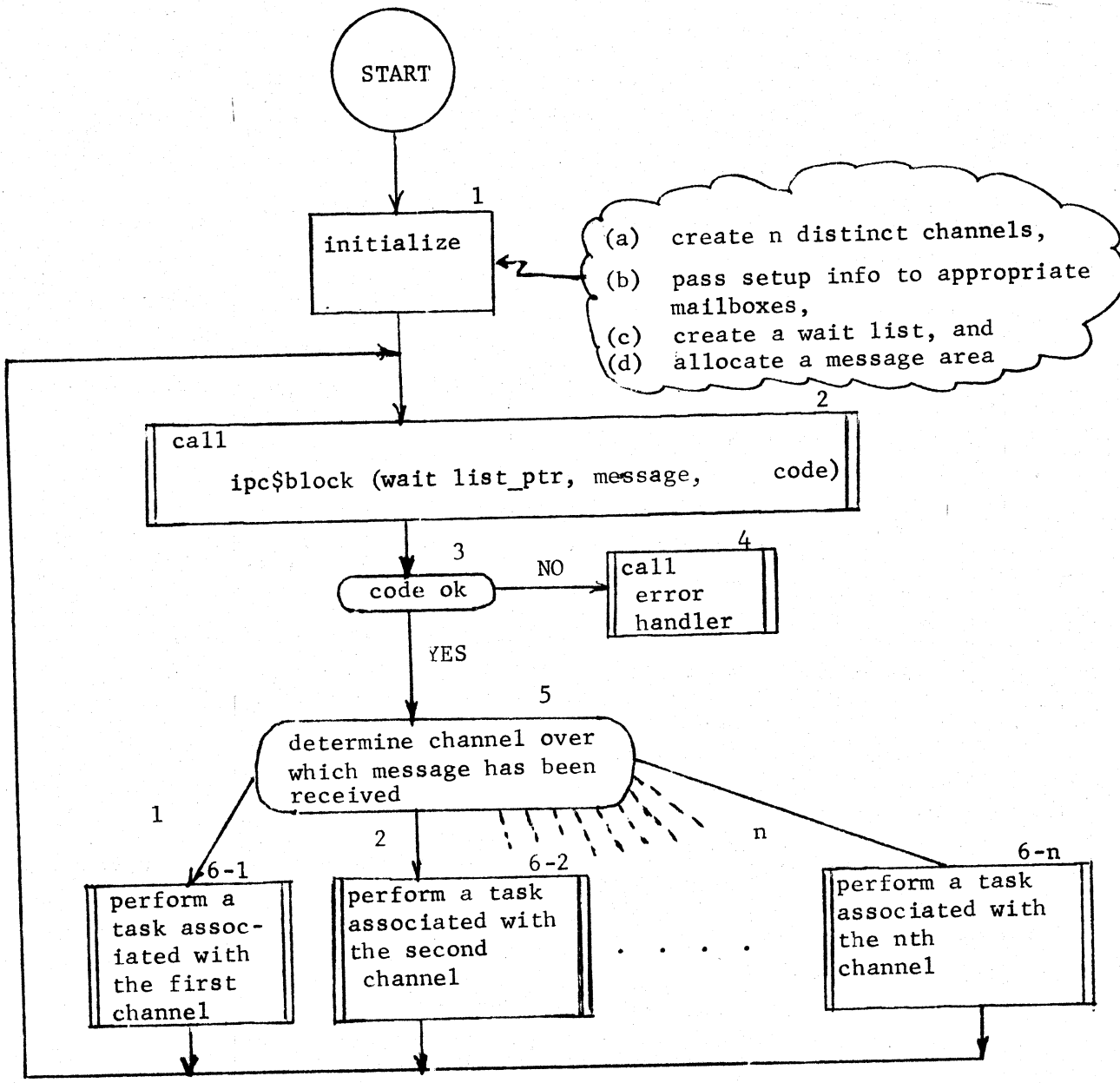


Figure 7-14 A possible structure for a multi-purpose process

### 7.5.3.2 Concept of the Wait Coordinator

As can now be seen, the code associated with entry point `ipc$block` is in fact more sophisticated than a simple scanner for messages received in event channels, since some action decisions (i. e., interpretation) are in fact delegated to this procedure. The code is referred to in MSPM documentation as the Wait Coordinator\*, and aptly so.

Once an event channel has been created, a programmer is free to declare, by a call to an appropriate IPC entry point, that said channel is thereafter to be regarded by the Wait Coordinator as event call type. Subsequently, when the process is executing inside the Wait Coordinator a scan of event channels that turns up a message in an event call channel will trigger a call to the associated procedure. It should be stressed that return from this call is to a return point within the Wait Coordinator. The net effect therefore, is that, in the case of event call channels, the action of boxes 5 and 6 of the Figure 7-14 flow chart is accomplished implicitly, i. e., on behalf of the procedure that calls the Wait Coordinator.

When a Multics user wishes to establish an event channel to be of the call type, he takes the following action:

- (1) Create the event channel by a call to `ipc$create_ev_chn`. (This step sets up the channel, but its default interpretation is of the event wait type, i. e., while given this interpretation it may only be used as pictured in Figure 7-14.)
- (2) Declare said channel to be of the event call type by a call to `ipc$decl_ev_call_chn`. The form of this call is

obtained from step (1)

call `ipc$decl_ev_call_chn` (`channel_name`, `associated_`  
`procedure_ptr`, `data_ptr`, `priority`, `code`).

The second and third arguments in this call are saved in the event channel table (ECT) for later use by the Wait Coordinator so it can construct the desired call to the associated procedure. Since a user is free to declare more than one event channel to be of the call type, it is necessary to provide the Wait Coordinator

---

\* The principal MSPM reference is BJ.10.03.

a scanning order for these channels. The user furnishes an integer argument, priority\*, to be used by the Wait Coordinator as a scanning index. Thus, if channel billy and channel tilly are both declared to be event call channels and priorities 2 and 1 are associated with them, respectively, then if messages have been received on both channels, the procedure associated with channel tilly will be called before the one associated with channel billy.

The next three subsections (7.5.3.3, 7.5.3.4, and 7.5.3.5) round out the design details of the Wait Coordinator that may be of interest to some readers. They can easily be skipped on a first reading.

### 7.5.3.3 Call-Wait Polling Order

Although we have just suggested the rule for scanning event call channels, we have yet to explain the dependency relationship that exists between rules for scanning event call channels and those for scanning event wait channels. Each call to the Wait Coordinator (in reality a call to `ipc$block`) is in fact a request to scan, not one, but two lists of channels, the wait list, and the call list. The wait list is the list of event wait channels that is pointed to (first argument) in the call to the Wait Coordinator. The call list is the list of event call channels that are currently kept in the ECT for the ring of the Wait Coordinator's caller.

We shall say that a W-C polling order is one in which the wait list is scanned first and then the call list, while a C-W polling order is the reverse (i. e., call list before wait list). The system's default polling order is W-C, but a user is given the opportunity, by calling a special entry point in the IPC, to reverse the current polling order.

It should be remembered that whenever a message is found in a channel of the wait list, channel scanning ends immediately. The discovered message (augmented by the wait list index) is copied into the caller's message area, and the Wait Coordinator returns to its caller. This means that when functioning in the default (W-C) polling order, the event call list is scanned only if no event wait message is found.

---

\* Strictly speaking, this argument is a priority level, the lower the integer (level), the higher the priority.

Whenever a call list is scanned, it is scanned in its entirety. Each event call channel is inspected for a message. If one is found, the associated procedure is called and, following a return from this call, if any, the next event call channel in the list is inspected, etc. The above scanning logic is summarized in the Figure 7-15 flow chart.

#### 7.5.3.4 Invoking an associated procedure and controlling its repeated use

A designer of a subsystem that uses an event call channel will, of course, be required to designate the name (or pointer) of the associated procedure at the time he declares the channel to be of event call type. The same designer may also be required to code this procedure. (We will call it AP, for convenience.) The Wait Coordinator, when it issues the call to AP, will always use a standard form for this call. AP's author must therefore code it so that it is compatible with this standard call. The rules are explained in the accompanying footnote.\*

Several messages can be queued up over the same event call channel. But the Wait Coordinator must see to it that it treats only one message at a time (the top most). It should not recognize the next message in the queue until processing of the top most is completed. This means that an associated procedure must return control before the Wait Coordinator can permit itself to again inspect the same event call channel. The following paragraphs show why the controls are needed and how they are achieved.

It is easy to see how the Wait Coordinator could get into this situation. Suppose AP1 is called for the first message of a call channel "1" and suppose that during its execution AP1 must call ipc\$block to await a message on some event wait channel. Further, suppose this message has not arrived at the time of this fresh call to ipc\$block. The Wait Coordinator might then find itself scanning channel 1 once again, and if it finds another message (assuming no controls were set to prevent it), would call AP1 once again. We would then have a

---

\* Conventions used in calling associated procedure (AP) are:

1. The AP has one argument, message\_ptr, which is a pointer to an eight-word based structure, the first six words of which are as given in Figure 7-10.
2. Before issuing the call, the Wait Coordinator appends to these six words a word pair whose value is that of data\_ptr. This item, you recall, is the third argument furnished in the declaration of the event call channel. The data\_ptr can, therefore, be thought of as an ordinary arglist ptr of a procedure, except it is available only indirectly in the message argument. In addition, of course, AP has access to the first six words of the message area, which is also useful information.

- b. the last thing it does, prior to returning, is to execute an ipc call that unmasks all event call channels, i. e.,

call ipc\$unmask\_ev\_calls (code);

This approach has the merit that any task that is invoked is treated as having absolute top priority. In a sense, it can be regarded as an extreme approach to solving the problem. Figure 7-18 illustrates what we mean. Here we show the effects of masking and unmasking event call channels using the same event timing sequence as in Figure 7-17. Note that task EC4, because of its low priority, is not even begun during the time span being considered.

- (3) This approach will no doubt prove to be the most attractive: Give up trying the multi-purpose approach in the first place! Go back to the principal-design approach of Multics and let each task that is now programmed as an event call task and in need of better response from its Wait Coordinator be made into an independent process to accomplish the same objective. Each such created process would have a single event call channel over which it can be signalled. Hence, competition for good response by its Wait Coordinator will now be eliminated.

Bear in mind, however, that after establishing separate processes for the several tasks, these can now, in principle anyway, be executed in parallel, whenever two or more processors can be awarded to these tasks during a single period of time. The mere fact that execution can proceed in parallel as a result of following this approach carries with it the need for greater care in the handling of shared data segments.

a message in any of a given list of designated mailboxes. We shall discuss these ideas in more detail later. We mention them here only to motivate the notion that a process may have what amounts to sets of mailboxes (each mailbox possibly empty), one set per ring.\* Clearly, each mailbox must bear a unique designation within the receiving process so that a sender can transmit his messages to their proper destination.

#### 7.5:2.1 Messages, Mailboxes (event channels) and Transmission

The technical name used in Multics for a mailbox is event channel. An event channel is uniquely designated by a 72-bit identifier †. This name is generated by the system as a result of executing a user-written subroutine call for the creation of an event channel.††

##### Origin of the Message

By added convention, every message originates as a 72-bit item of arbitrary content (set by the sender). (However, in the course of transmitting the message, system routines expand it with self-identifying information.)

A message is sent in the form of a call to the hard-core system routine, hcs\_\$wakeup:

```
call hcs_$wakeup (receiving_process_id,  
                  channel_name,  
                  message,  
                  code);
```

---

\* There is one byproduct benefit that comes from the implementation decision to have multiple mailboxes per ring. Let the distinct wait points in some ring-*r* of a process *A* be designated as *wp1*, *wp2*, ..., etc. Suppose the wait at each of these points is for a message from a correspondingly different process, e.g., from processes *p1*, *p2*, ..., etc. Prior arrangements between the process pairs (*A*, *p1*), (*A*, *p2*), (*A*, *p3*), etc., for the sending of messages to *A* need not be fully coordinated in the sense *A* is not forced to give (or to divulge) to *p1*, *p2*, *p3*, etc., the very same mailbox name. One can regard this flexibility as an advantage in that there may be less risk of confusion if separate senders are asked to send messages to different mailboxes, with each mailbox having a different meaning.

† The substructure of the event channel name includes three items, a ring number, a key (52 bits), and an ECT address. The key is a unique name representing the wall clock time at which the event channel was created for this process. The ring number identifies the ring in which the receiver expects to examine messages placed in this channel. Received messages are saved (until inspected) in a one-per-ring segment called an ECT (Event Channel Table). The ECT address is simply the offset within this segment at which the possibly-queued messages for this channel may be found. A channel is in effect a FIFO list. Details of the ECT data structures should be of no interest to users. They may be found in BJ.10.02.

††Details on how to create event channels may be found in BJ.10.01.

Note that although the actual text of a message is small and fixed in size, it is large enough to be used as a pointer to messages of arbitrary size. We defer momentarily answering the obvious question, namely, how will the sender know both the process\_id of the receiver and its receiving point (the event channel name). This matter is taken up in the section entitled Setup for Interprocess Communication.

The hcs\_\$wakeup routine makes some simple (routine) checks on the first two arguments so that if they are obviously erroneous\*, due to programmer error, the caller can be alerted if he chooses to examine the returned error code. After this partial validation, the Traffic Controller's wakeup entry is called, at which point steps are taken to forward the message to the intended receiver, as outlined below.

1. If the message indeed has a receiver, then it must be possible to match the receiver process\_id with the id of one of the processes that now have entries in the Active Process Table. Failure to find such a match means that the message is meaningless. Such a case results in an appropriate error code being reflected to hcs\_\$wakeup's caller. (Note that a post-office analogy to this case is -- "addressee unknown at this address -- return to sender".)
- 2a. A message aimed at a bona fide target process will be copied into a ring zero system table where it is properly augmented with "truthful" information about the sender. The system table (central storage) is called the ITT (for Interprocess Transmission Table). The receiving process will later fetch the message out of this table.
- 2b. The last step is to call the Traffic Controller at its entry point wakeup to wake up the receiving process.

The above steps are summarized in the Figure 7-9 flow chart. Note that hcs\_\$wakeup serves as the user's only interface with the otherwise inaccessible wakeup entry in the Traffic Controller. Protecting this entry from direct user calls simplifies the logic of the Traffic Controller which, because it is locked to all other processes when entered, must be kept as simple (and fast-executing) as possible.

---

\* For example, if the process id or certain of the subfields of the 72-bit channel name are zero, this is clearly an error.



Receivers must be protected against receipt of false messages, whether accidental or intentionally sent. Certain information about the sender is therefore added to the copied message that is placed in the ITT. This information, which is of critical importance for the protection of the receiver, consists of the sender's process id and the validation level of the sender's call to hcs \$wakeup (i. e., the ring in which this call was made). The user cannot be trusted to transmit these items accurately. Figure 7-10 shows the message format as stored in the ITT. The Interprocess Transmission Table is a wired-down system table in ring-0 that is large enough to hold messages for all known processes. The table is organized as a set of message queues, one per process. The head of each queue is pointed to from a fixed position in the APT entry for the corresponding process, so that when any process re-enters the running state in the Traffic Controller, as a result of being awakened, it can quickly determine if there have been any messages deposited in the ITT on its behalf since the last time it ran.

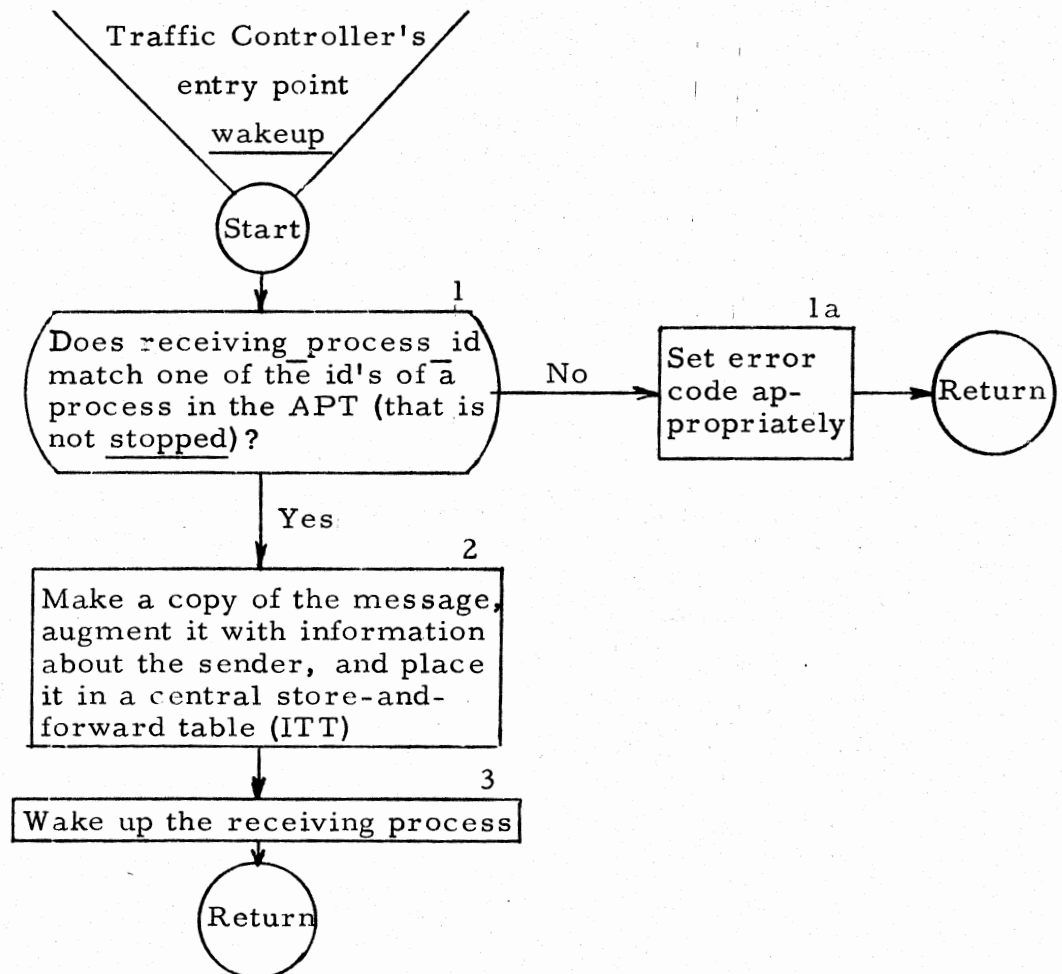


Figure 7-9 Some details of the Traffic Controller's entry point wakeup

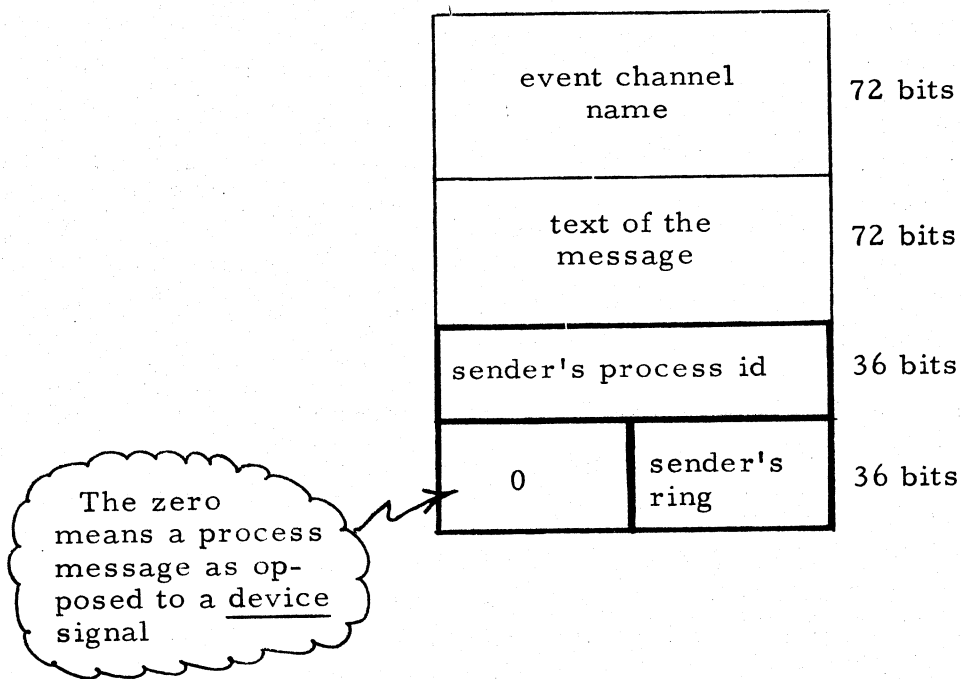


Figure 7-10 Message format augmented with information about the sender as placed in the ITT

Getting the message from the central store-and-forward point to the receiver

So far we have considered mainly the mechanics of sending a message as far as a central forwarding center. In a postal system analogy such as shown in Figure 7-11, this is the halfway point, e.g., a regional post office. No ordinary citizen is able to walk up to this center and ask for his mail. Nor, by analogy, can the Multics user expect to get his mail by attempting to read messages while they are still in the ITT. He needs help in moving the messages to data areas that are ring-accessible for his purpose. While the post office automatically pushes the mail through to its receiver from the central p.o., without any special coaxing, the Multics analogy is somewhat different. Here some initiative is always taken by the receiver to pull the message(s) out of central storage and to place them into the individual ring-accessible event channels of the process. Recall that a receiver's process will have a table of one or more event channels (an ECT) in every ring in which there occurs a distinct wait point in that process.

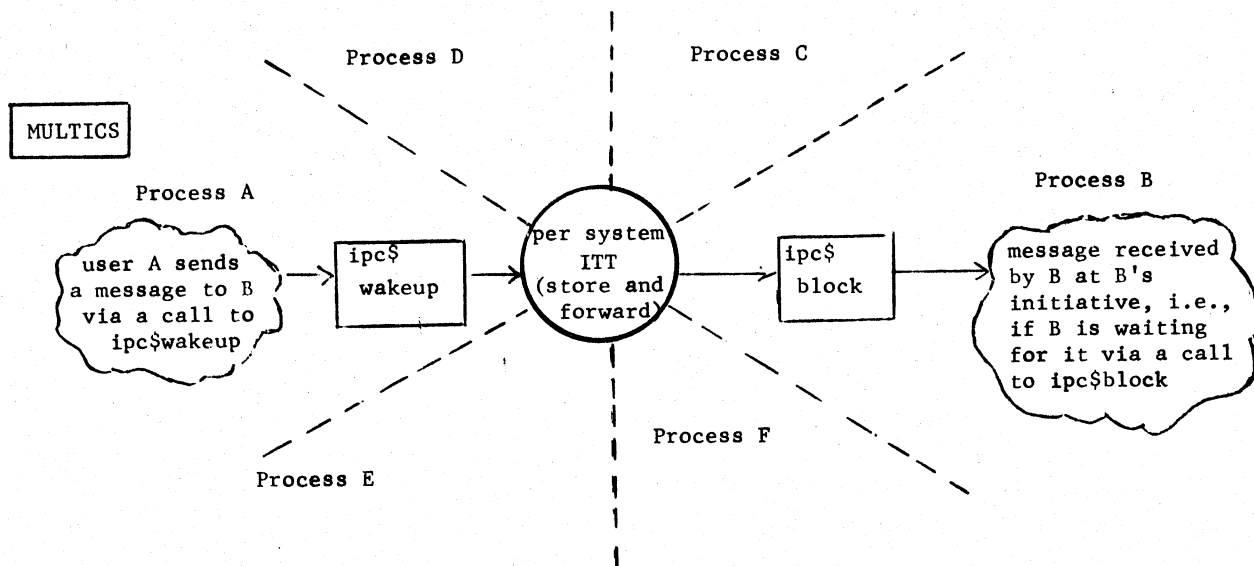
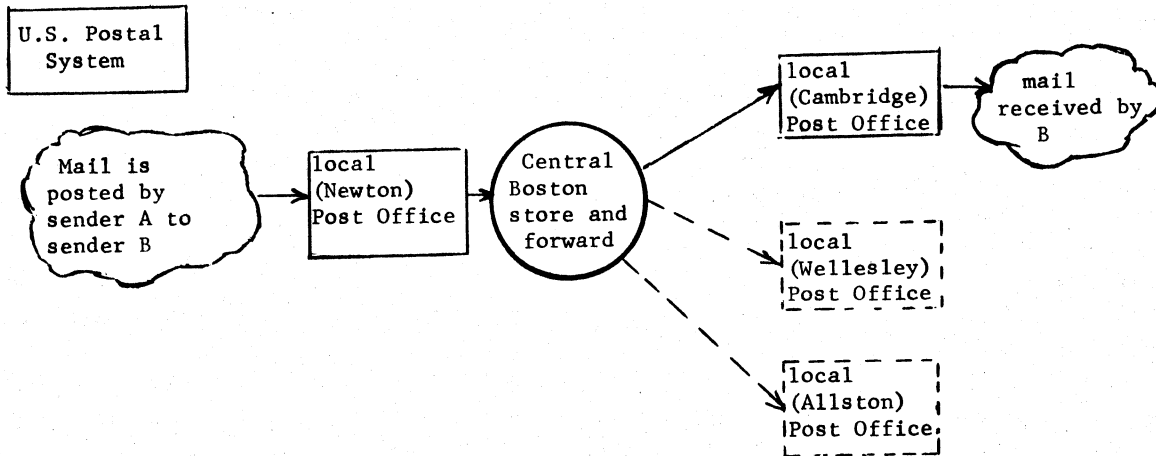




Figure 7-11 Postal system analogy to Multics interprocess message transmission

A wait point is always programmed as a call to `ipc$block` which is the entry point in the so-called wait coordinator, the heart of the interprocess communication facility. A user program should call this entry point whenever it must enter the blocked state while awaiting the receipt of a message.

The form of this call is

```
call ipc$block (wait_list_ptr, message_ptr, error_code);
```

 A list of one or more event-channel names

 A pointer supplied as an input argument that specifies the location where the caller expects to receive a message which he can examine

When called, `ipc$block` scans the event channels in the list pointed to by the first argument. Scanning of the channels is done in the listed order, and if lucky enough to find a message in one of these channels, `ipc$block` transfers the first such message found into the location given by the second argument, and returns to its caller. The message that is actually transferred consists of the six-word message whose format was shown in Figure 7-10, augmented by a seventh word consisting of the wait-list index. Thus, if a message is found in the third of eight channels on a wait list, the seventh word of the returned message will have the value 3.

Note that `ipc$block` has been executing in the ring of its caller. (`ipc$block` has ring brackets which are (1, 63, 63).) Consequently, this procedure does not have ring access for scanning central storage (i. e., the ITT) which may have received one (or more) of the desired messages. Therefore, `ipc$block` is forced to call a privileged routine in ring-0 (at an entry point `hcs_$block`). This routine in effect transfers all valid messages that have accumulated in the ITT event queue for this process. Each message is placed in the event channel that is designated in that message. Invalid messages, such as those whose channel names do not match existing channels in the receiving process' ECT's, are summarily discarded. If, in the course of making these message transfers, not a single message was transferred into a ring  $\geq$  the validation ring (i. e., that of `ipc$block`), then it is clear the wanted message cannot have yet been received. Hence `hcs_$block`, which is fully privileged to do so, calls the corresponding entry in the Traffic

Controller to give away the processor.\* If, on the other hand, at least one such message was moved to a ring that is accessible to ipc\_\$block, then hcs\_\$block will return so that the former can again scan its given list of channels in hopes of finding the wanted message. The chain of calls we have just discussed is summarized in Figure 7-12.

If we were to follow the return path from the TC backwards toward the point of call to ipc\$block, it becomes easy to see how a fresh message, received at ipc\$wakeup can be thought of as being forwarded from central storage to the appropriate event channel of the receiver.

It should be recalled that when an awakened process finally recaptures a processor, effective execution will resume as a return from the block entry in the TC to its caller, hcs\_\$block. The latter then "transfers" all newly arrived messages from its ITT event queue into the appropriate event channels. If no messages were transferred into rings  $\geq$  that of the caller, ipc\$block, then the process cannot have received the message it was waiting for. Hence, hcs\_\$block again calls the TC at entry point block to give away the processor. But if at least one potentially suitable message was transferred from the ITT, hcs\_\$block returns to its caller (ipc\$block). (This is how the pulling of messages is done--in the absence of an explicit effort, e.g., a call to ipc\$block, messages for this process can in principle pile up in central storage without ever being drawn out.) Note that the return to ipc\$block is no guarantee of a return to its caller. If ipc\$block finds no message in one of the listed event channels, it simply recalls hcs\$block.

#### 7.5.2.2 Setup for Interprocess Communication<sup>†</sup>

Here we shall discuss how a sender learns the identification of a receiver process and the identification of that receiver's event channel. For convenience let us adopt the following notation.

---

\* To be absolutely precise about things, there is still a possibility for a last minute "reprieve", if anytime up to the very last instant before giving the processor away a wakeup arrives, control will return to the TC's caller. For more details you could review J. H. Saltzer's Ph. D. thesis or BJ. 3.01 to see the function of the so-called "wakeup waiting" switch.

† For a more basic discussion of this topic, the reader may wish to examine the paper, "The Multics Interprocess Communication Facility", by M. J. Spier and E. I. Organick, submitted for presentation at ACM's Second Symposium on Operating Systems Principles, Princeton University, Princeton, New Jersey, October, 1969.

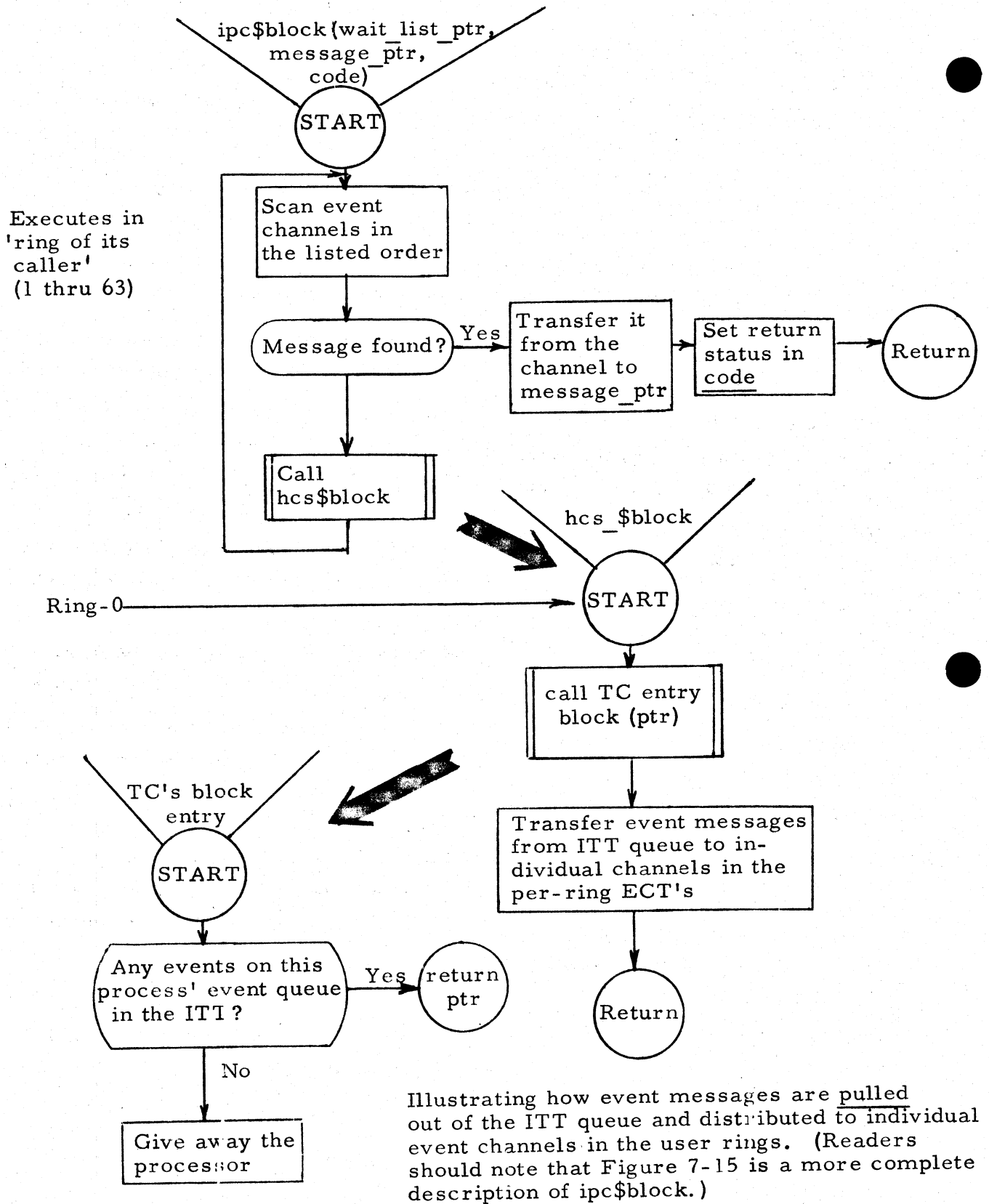


Figure 7-12 The chain of calls: → ipc\$block → hcs\$block → TC block entry.

Let B-to-A setup info be that basic information that is required by a sender process B so that it can send a message to a receiver process A. This information consists of A's 36-bit process id and A's 72-bit event channel name.

Let p(A) and p(B) refer to the people responsible for programming A and B, respectively. (To be sure, they may be the same individual, wearing "two hats".) Clearly, the system-provided message transmission facility (IPC) cannot be employed to transmit B-to-A setup info, else why would setup info be needed in the first place. Note also that p(A) cannot supply p(B) with the B-to-A setup info by telephone or by other direct personal communication until after A has been created. This is because a process id is a clock-dependent unique bit string that is generated by the system at process creation time. Furthermore, A's event channel name, which is also a clock-dependent unique bit string, will not be known until after A's declaration that creates the event channel has been executed. There appears to be only one sensible plan for passing setup info. The plan is as follows:

- (1) p(A) and p(B) agree in advance on the (unambiguous) name of a segment that is to be shared by A and B. Call this segment <shared>. Also agreed upon is an offset within <shared>; call it [setupBA] which is to be regarded as a 3-word mailbox initially set to zero.
- (2) After A and B have been created, and after A has declared (created) the appropriate event channel, A places in shared\$setupBA the desired setup info.
- (3) B fetches the three words at shared\$setupBA, and if non-zero, assumes by convention that the required A-to-B setup information has been obtained.

Note that if A is also to become a sender to B (not just a receiver), then A-to-B setup (as opposed to B-to-A) is also needed. This info can be sent by a similar prearrangement, although in fact a form of "boot strapping" can now be achieved if it is desired, to avoid further use of <shared>. That is, the first message B sends to A can, by further convention, contain the A-to-B setup information.

But how does A know its B-to-A setup info so that it can place it at shared\$setupAB?

#### How A obtains its own process id

When a process is created, one of the temporary segments that is created for it and placed in its process directory is called <process\_info>. This segment

contains special information about the process that can be read by all procedures. Among these is the process id which is stored in `process_info$processid`.<sup>\*</sup> Any user procedure can snap a link to and read this word.

### How A obtains an event channel name

A user creates an event channel simply by calling `ipc$create_ev_chn`. The first of two return arguments in the call contains (upon return) the 72-bit name that the IPC has established for this channel. Henceforth, it is the user's responsibility to keep track of this name.

### Summary

The steps that would be coded by p(A) and p(B) to establish interprocess communication with B as a sender and A a receiver can now be summarized.

1. p(A) codes the following steps in some procedure of A:
  - (a) call `ipc$create_ev_chn(channelBA, code)`; this call creates an event channel which can hereafter be referred to by the name, `channelBA`, because the value of the first return argument is a 72-bit unique id of `channelBA`.
  - (b) assign to the 3-word mailbox at `share$setupBA` values of `process_info$processid` and `channelBA`. Illustrative epl coding is provided in the accompanying footnote.<sup>†</sup>
2. p(B) can code B to pick up the required setup info at any time and use it to send a two-word message to A. Illustrative epl coding

---

<sup>\*</sup> Some consideration is currently being given to merging `<process_info>` with another segment in order to reduce the working set. It is for this reason that this argument was not listed in Table 7-2. If this change should be implemented, however, the name, `process_info`, will still be used in referencing the process id.

<sup>†</sup> In epl, this might be accomplished with coding that relies on a 3-word based structure for a mailbox:

```
dcl 1 mailbox3 based (p),
    2 pid bit (36) /* a process id*/
    2 cname fixed bin (71) /* a channel name*/;
```

Then, the executable code which would follow creation of the desired event channel might look like:

```
p = addr(shared$setupBA);
p -> mailbox3.pid = process_info$processid;
p -> mailbox3.cname = channelBA;
```



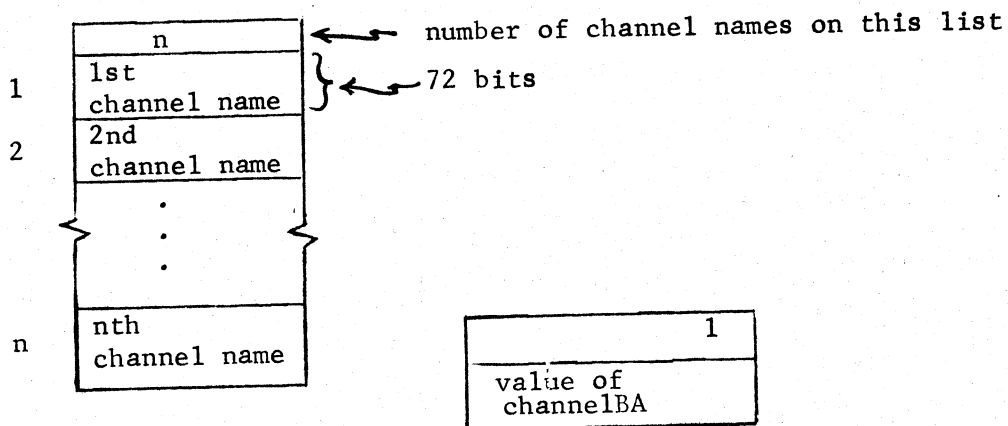
is provided in the accompanying footnote.\*

3. p(A) is now able to code appropriate calls to ipc\$block at various points in A, to wait for messages from B. A call of the form

```
call ipc$block (argptr, msgptr, code);
```

gets the job done on the assumption that the first two arguments are pointers to the base of structures, the first being to a wait list of channel names, and the second to an area of sufficient size for receipt of the message.

The general structure for the wait list is of the form shown in Figure 7-13.



(a) general form of a wait list

(b) appearance of wait list for example in the text.

Figure 7-13 Wait lists. General and Specific.

\* We shall assume that B also uses a declaration for a 3-word mailbox identical to the one in the preceding footnote. Then coding in B might appear as:

```
p = addr(shared$setupBA);
receiver_pid = p -> mailbox3.pid
channel_name = p -> mailbox3.chname
if ¬(receiver_pid = 0 and channel_name = 0)
  then call hcs$wakeup (receiver_pid, channel_name, message, code);
  else call print_error;
```

Here message is a 72-bit message and print\_error might be a routine to print an appropriate error message before proceeding with whatever steps are then deemed appropriate.

### 7.5.3 Programming of a Multi-purpose Process

A Multics process is basically sequential in nature by virtue of the fact that but a single execution point (or point of control) is free to traverse over its address space at any one time. For this reason, it is natural to think of such a process as having a single purpose.

If two or more independent computations are to be performed, albeit related to one another, it is entirely appropriate for the programmer to create a separate process, one per each defined purpose, and have these processes execute in any interrelated fashion that seems appropriate. In fact, this approach is recommended for most initial efforts of this kind.

Subject to processor availability, concurrent computation of the separate but related processes may occur in some fashion, but it is not predictable, of course, since the Traffic Controller and its functions are outside the control of the programmer. In any case, by proper use of IPC, the separate processes (purposes) may synchronize with one another.

It is worth noting, however, that the establishing and maintaining of separate address spaces, one per process, incurs an appreciable system overhead. Such costs are ultimately passed on to the user directly or indirectly. Hence it may well be worth considering under what circumstances it is feasible to coalesce (and condense) the address spaces of several processes into a single, now multi-purpose process having one address space (and one execution point).

Certainly, it is necessary that concurrent pursuit of the separate purposes, i. e., parallel executing of the separate tasks be no requirement. (But, then such a requirement, even without coalescing, cannot be guaranteed in Multics anyway.) Beyond this, the order in which these tasks may be initiated and executed should in some sense be of secondary importance and perhaps be independent of the tasks themselves. This requirement may be satisfied in the case where events external to the process drive the multi-purpose process. That is, IPC messages received by the process are the basis for deciding which task to execute next. Examples of multi-purpose processes are common among system software processes, e. g., answering services, I/O device managers, automatic recorders, automatic file dumpers, etc.

A process that must behave in multi-purpose manner, can in principle be coded using flow chart logic described in Figure 7-14. The basic idea suggested

in the figure is to create  $n$  event channels, one for each of the  $n$  distinct purposes of the process. After furnishing setup information for use of each of the  $n$  channels to the  $n$  senders (not necessarily  $n$  different processes) the process calls `ipc$block` to await one of the  $n$  types of events. Each time an event arrives, `ipc$block` returns with a message. The message is examined (in box 5 of the flow chart) to determine which channel (type of event) has occurred so as to invoke the associated task. When the task is completed, a call is again issued to `ipc$block`.

We have not yet defined what we mean by a task. The simplest idea is to suppose it corresponds to a call to a procedure that is associated with the corresponding event channel. We will be interested in understanding what restrictions are imposed, if any, as to what may go on inside the called procedure. For instance, are calls to `ipc$block` to be permitted from within the associated procedure and/or from any of its dynamic descendents? We shall consider this possibility in the next subsection. For the moment, however, we shall assume such repeated calls do not happen.

#### 7.5.3.1 Event call channels

Note that further logical simplification (from the point of view of the user) arises and a slight increase in efficiency can be gained if the control logic of boxes 3, 4, 5 and 6 are made part of `ipc$block`. At the top-most logic level the process would be characterized simply as the execution of boxes 1 and 2. That is, initializing of channels, transmitting of setup information, etc., followed by a single call to `ipc$block`. There would be no return and of course no repeated calls on `ipc$block`. Of course, it would be necessary to furnish IPC with more information so it can perform its more elaborate job. Basically, this amounts to telling IPC what are the procedures that should be called (invoked) upon receipt of respective messages.

The "simplification" we have been discussing is in fact provided for in Multics by allowing the user to designate event channels of his choice for special interpretation. Event channels marked in this fashion are referred to as event call channels, as opposed to the ordinary event wait channels. Messages found in event call channels are examined and interpreted while the process is executing inside the IPC. Interpretation amounts to execution of a call to the associated procedure.

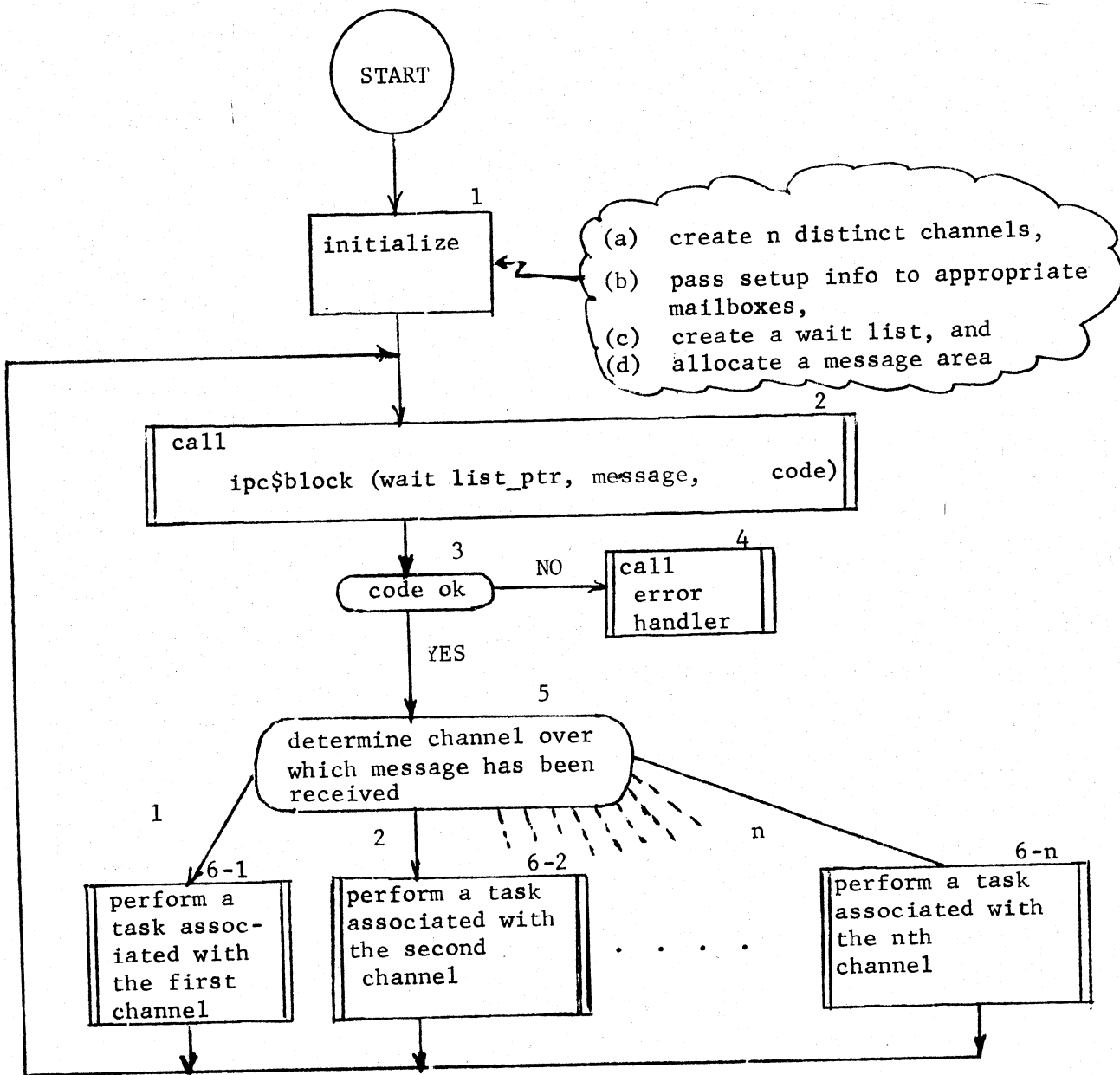


Figure 7-14 A possible structure for a multi-purpose process

### 7.5.3.2 Concept of the Wait Coordinator

As can now be seen, the code associated with entry point `ipc$block` is in fact more sophisticated than a simple scanner for messages received in event channels, since some action decisions (i. e., interpretation) are in fact delegated to this procedure. The code is referred to in MSPM documentation as the Wait Coordinator<sup>\*</sup>, and aptly so.

Once an event channel has been created, a programmer is free to declare, by a call to an appropriate IPC entry point, that said channel is thereafter to be regarded by the Wait Coordinator as event call type. Subsequently, when the process is executing inside the Wait Coordinator a scan of event channels that turns up a message in an event call channel will trigger a call to the associated procedure. It should be stressed that return from this call is to a return point within the Wait Coordinator. The net effect therefore, is that, in the case of event call channels, the action of boxes 5 and 6 of the Figure 7-14 flow chart is accomplished implicitly, i. e., on behalf of the procedure that calls the Wait Coordinator.

When a Multics user wishes to establish an event channel to be of the call type, he takes the following action:

- (1) Create the event channel by a call to `ipc$create_ev_chn`. (This step sets up the channel, but its default interpretation is of the event wait type, i. e., while given this interpretation it may only be used as pictured in Figure 7-14.)
- (2) Declare said channel to be of the event call type by a call to `ipc$decl_ev_call_chn`. The form of this call is

obtained from step (1)

call `ipc$decl_ev_call_chn` (`channel_name`, `associated_`  
`procedure_ptr`, `data_ptr`, `priority`, `code`).

The second and third arguments in this call are saved in the event channel table (ECT) for later use by the Wait Coordinator so it can construct the desired call to the associated procedure. Since a user is free to declare more than one event channel to be of the call type, it is necessary to provide the Wait Coordinator

---

\* The principal MSPM reference is BJ.10.03.

a scanning order for these channels. The user furnishes an integer argument, priority\*, to be used by the Wait Coordinator as a scanning index. Thus, if channel billy and channel tilly are both declared to be event call channels and priorities 2 and 1 are associated with them, respectively, then if messages have been received on both channels, the procedure associated with channel tilly will be called before the one associated with channel billy.

The next three subsections (7.5.3.3, 7.5.3.4, and 7.5.3.5) round out the design details of the Wait Coordinator that may be of interest to some readers. They can easily be skipped on a first reading.

### 7.5.3.3 Call-Wait Polling Order

Although we have just suggested the rule for scanning event call channels, we have yet to explain the dependency relationship that exists between rules for scanning event call channels and those for scanning event wait channels. Each call to the Wait Coordinator (in reality a call to `ipc$block`) is in fact a request to scan, not one, but two lists of channels, the wait list, and the call list. The wait list is the list of event wait channels that is pointed to (first argument) in the call to the Wait Coordinator. The call list is the list of event call channels that are currently kept in the ECT for the ring of the Wait Coordinator's caller.

We shall say that a W-C polling order is one in which the wait list is scanned first and then the call list, while a C-W polling order is the reverse (i. e., call list before wait list). The system's default polling order is W-C, but a user is given the opportunity, by calling a special entry point in the IPC, to reverse the current polling order.

It should be remembered that whenever a message is found in a channel of the wait list, channel scanning ends immediately. The discovered message (augmented by the wait list index) is copied into the caller's message area, and the Wait Coordinator returns to its caller. This means that when functioning in the default (W-C) polling order, the event call list is scanned only if no event wait message is found.

---

\* Strictly speaking, this argument is a priority level, the lower the integer (level), the higher the priority.

Whenever a call list is scanned, it is scanned in its entirety. Each event call channel is inspected for a message. If one is found, the associated procedure is called and, following a return from this call, if any, the next event call channel in the list is inspected, etc. The above scanning logic is summarized in the Figure 7-15 flow chart.

#### 7.5.3.4 Invoking an associated procedure and controlling its repeated use

A designer of a subsystem that uses an event call channel will, of course, be required to designate the name (or pointer) of the associated procedure at the time he declares the channel to be of event call type. The same designer may also be required to code this procedure. (We will call it AP, for convenience.) The Wait Coordinator, when it issues the call to AP, will always use a standard form for this call. AP's author must therefore code it so that it is compatible with this standard call. The rules are explained in the accompanying footnote.\*

Several messages can be queued up over the same event call channel. But the Wait Coordinator must see to it that it treats only one message at a time (the top most). It should not recognize the next message in the queue until processing of the top most is completed. This means that an associated procedure must return control before the Wait Coordinator can permit itself to again inspect the same event call channel. The following paragraphs show why the controls are needed and how they are achieved.

It is easy to see how the Wait Coordinator could get into this situation. Suppose AP1 is called for the first message of a call channel "1" and suppose that during its execution AP1 must call ipc\$block to await a message on some event wait channel. Further, suppose this message has not arrived at the time of this fresh call to ipc\$block. The Wait Coordinator might then find itself scanning channel 1 once again, and if it finds another message (assuming no controls were set to prevent it), would call AP1 once again. We would then have a

---

\* Conventions used in calling associated procedure (AP) are:

1. The AP has one argument, message\_ptr, which is a pointer to an eight-word based structure, the first six words of which are as given in Figure 7-10.
2. Before issuing the call, the Wait Coordinator appends to these six words a word pair whose value is that of data\_ptr. This item, you recall, is the third argument furnished in the declaration of the event call channel. The data\_ptr can, therefore, be thought of as an ordinary arglist ptr of a procedure, except it is available only indirectly in the message argument. In addition, of course, AP has access to the first six words of the message area, which is also useful information.

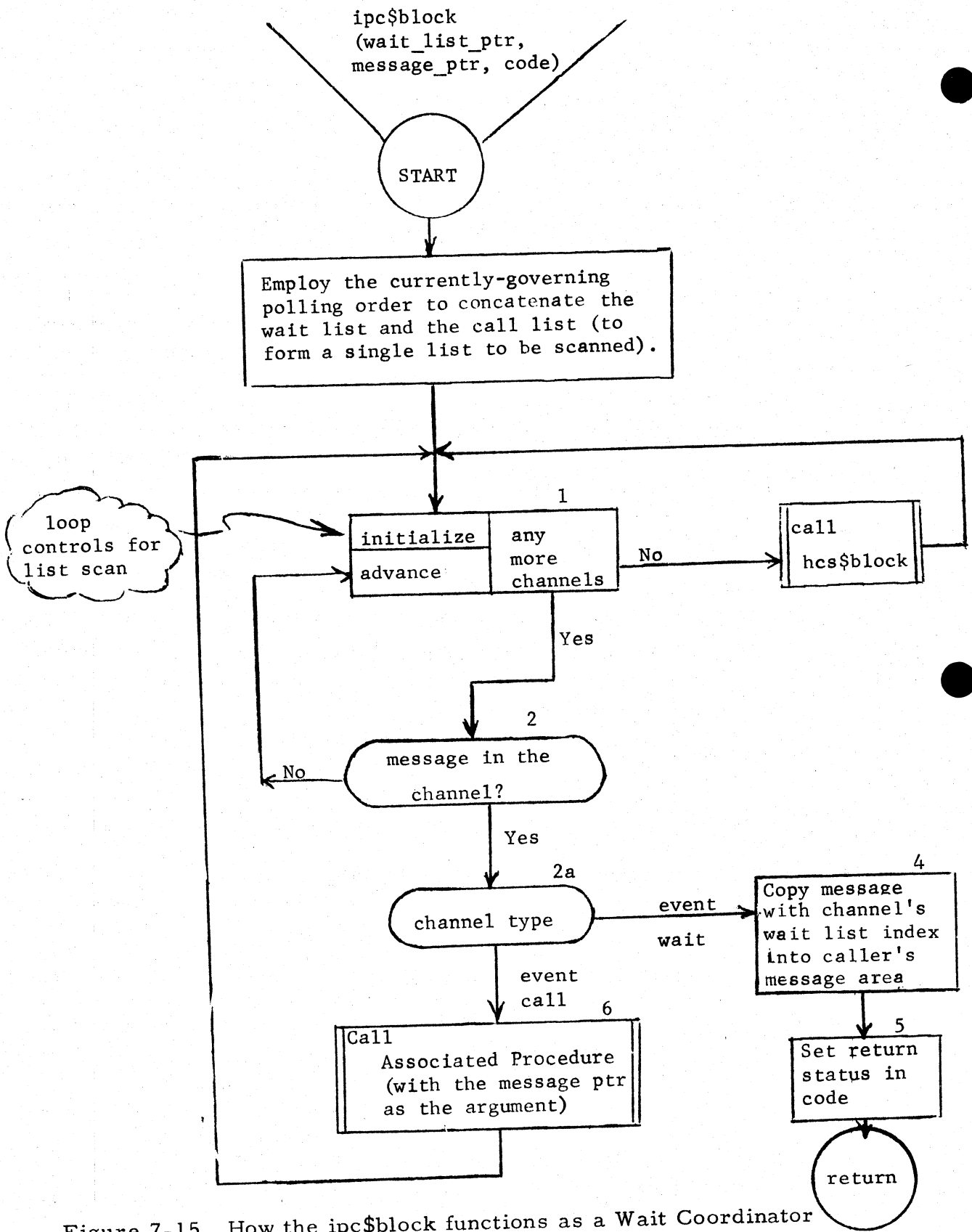
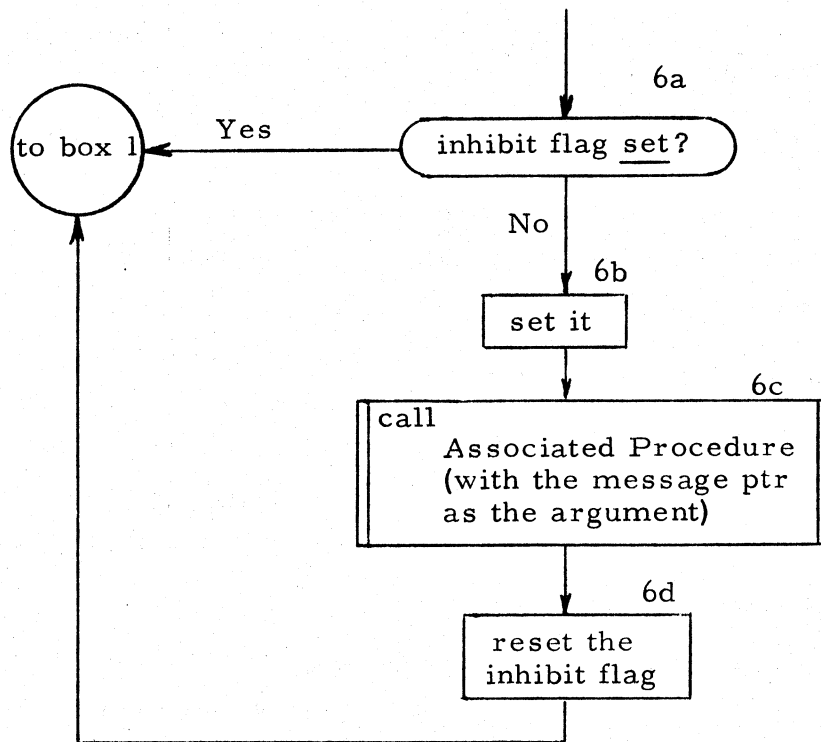


Figure 7-15 How the ipc\$block functions as a Wait Coordinator (Note that this is an expanded version of the flow chart given in Figure 7-12.)



situation that there are two invocations of API, each permitted to perform operations over the same set of external (and static internal) variables. Since the first activation could be suspended after making incomplete alterations to such variables, chaos could easily result.

To avoid this kind of confusion, the Wait Coordinator associates and maintains an inhibit flag for each event call channel. This flag is set immediately prior to the call to —and is reset immediately following the return from— the associated procedure. Moreover, event call channels that have inhibit flags set are ignored by the Wait Coordinator whenever the list of channels is scanned. This simple set of controls has been omitted from the picture given in Figure 7-15 to keep things simple, but could be added simply by replacing box 6 with the following amplification:



### 7.5.3.5 Other Channel Management Functions

The subsystem designer who has a further need to know about event channels and their management will be pleased to learn that the IPC offers a number of other services. Using these capabilities, a user may, for instance, control the polling order, delete as well as add new event channels, drain or flush out unwanted messages from existing channels, cause a given list of channels to be

masked during certain periods, i. e., skipped over during normal scanning by the Wait Coordinator, convert event call channels back to the event wait category, associate a given call channel with a new procedure and/or data pointer, and last, but of considerable importance, read the messages in a given channel (without waiting). Details for making the appropriate IPC calls can be found in BJ.10.01.

#### 7.5.4 Limitations of Multi-purpose Processes

The study of the Wait Coordinator has provided us with a new frame of reference for discussing multi-purpose processes. With additional study we can understand the potential as well as the limitations of such Multics Processes.

Because each task of a Multics process must share the same stack with its sister tasks, the order in which events arrive and their time spacing clearly determines the order in which tasks are started and completed. A feeling for this event dependency can be gained by studying timing diagrams for specific cases. We present in Figures 7-16 and 7-17 two cases, each for a multi-purpose process having four event call channels whose associated procedures are EC1, EC2, EC3, and EC4. The respective priorities for these channels are assumed to be 1, 2, 3, and 4.

##### 7.5.4.1 Two Case Studies Using Timing Diagrams\*

Figure 7-16 covers a period of time which commences while associated procedure EC1 is executing. Events arrive for channels in order 2, 3, 4, 2, 1, spaced as shown in the vertical time line on the left side of the figure. The vertical line segments in columns marked EC1, EC2, etc., correspond to execution times for the respective tasks, which are carried out in the order 2, 3, 4, 1, and 2. This is somewhat different from the order in which the event messages were received due to priority considerations.

Figure 7-17 is a similar case, but exhibits one important complication, namely: at some point during its execution each task must make a call to ipc\$block to await a specific message (on an event wait channel). Arrival times for event wait messages are labeled  $ew_1$ ,  $ew_2$ , etc.

We gain valuable insight by "walking" through this timing diagram to see why things happen the way they do.

---

\* This section may be skipped on a first reading without loss of continuity.

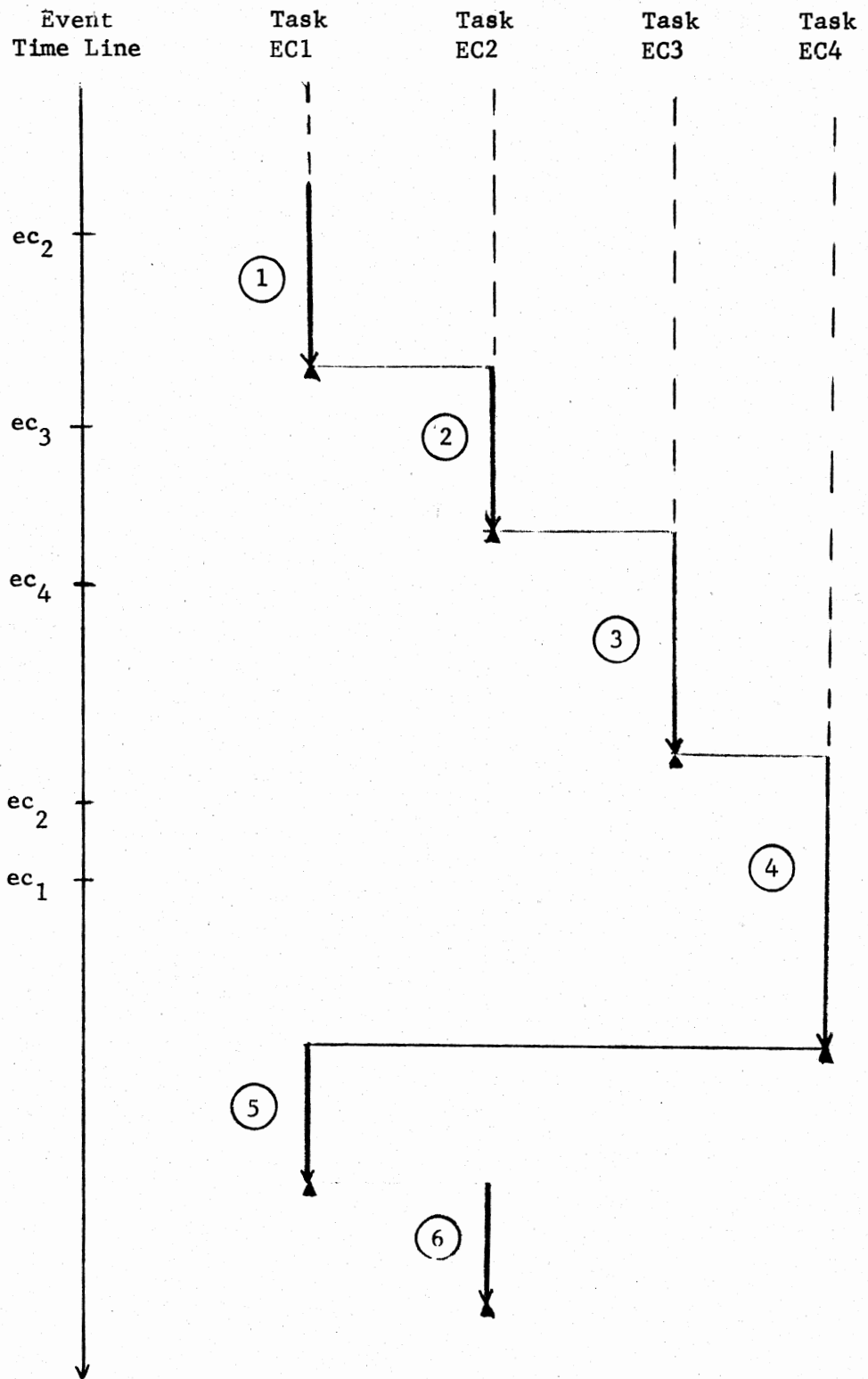


Figure 7-16 Timing diagram showing execution of tasks EC1, EC2, EC3, and EC4 when triggered by events that arrive at points in time labeled  $ec_1$ ,  $ec_2$ , etc., as indicated on the (vertical) event time line. Circled numbers show the sequence in which tasks are executed in virtual time.

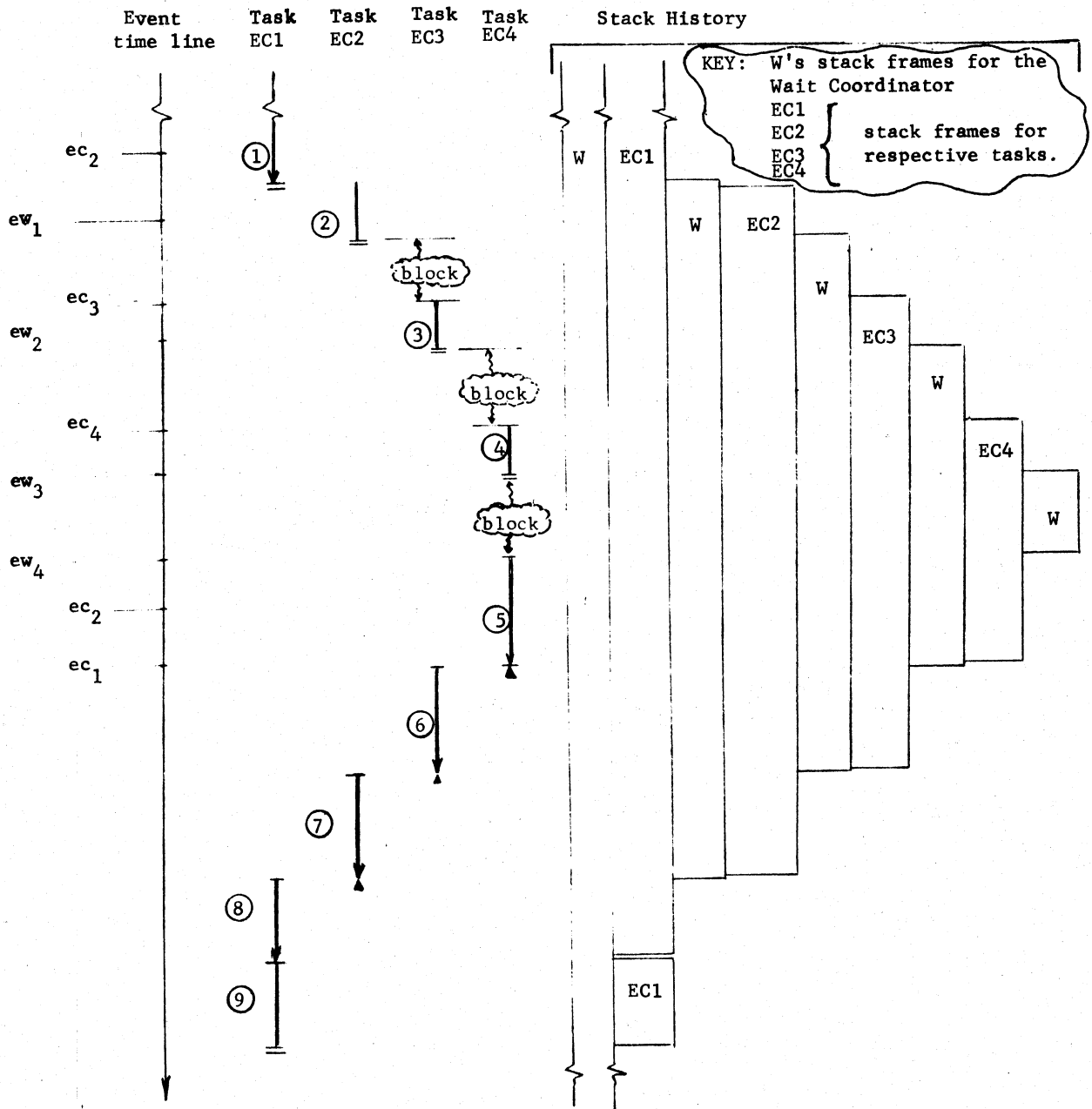


Figure 7-17 Timing Diagram showing execution of tasks EC1, EC2, EC3, and EC4. This example assumes each task executes one call to ipc\$block to await distinct events, labeled ew<sub>1</sub>, ew<sub>2</sub>, ew<sub>3</sub>, and ew<sub>4</sub>, respectively. Note that a W-C polling order is assumed.

- (1) When task EC1 has called the Wait Coordinator to await arrival of event  $ew_1$ , the Wait Coordinator discovers event  $ec_2$ , so it triggers task EC2 which proceeds until it reaches a wait point.
- (2) During execution of EC1, the event  $ew_1$  that task EC1 has been waiting for, has arrived, but cannot be recognized by the Wait Coordinator when it is called by EC2 to await event  $ew_2$ . Consequently, the process is forced to block itself until the next recognizable event arrives, which, in our particular example, is  $ec_3$ . Arrival of this event causes the Wait Coordinator to invoke task EC3.

The reason why the Wait Coordinator fails to recognize  $ew_1$  when it is called by task EC2 is simple: The event  $ew_1$  is not on the wait list of this call! The Wait Coordinator is in fact executing with a new stack frame. Hence, this activation of the Wait Coordinator will not be "looking for"  $ew_1$ . When will the Wait Coordinator again look for  $ew_1$ ? In a moment we will have the answer to this question, but first let us continue our walk through Figure 7-17.

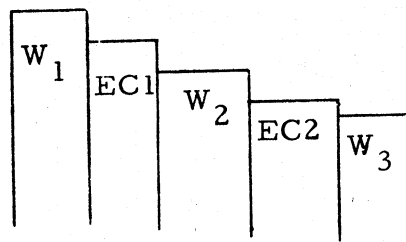
- (3) As a result of invoking EC3, this task executes until it reaches a wait point and calls the Wait Coordinator with the wait list,  $ew_3$ . Since this event has not yet arrived, and since no event call message is initially present, the process is again forced to block itself.
- (4) The process is revived following arrival of  $ec_4$ , at which time task EC4 is invoked.
- (5) After going blocked again for a short period,  $ew_4$  arrives. The Wait Coordinator recognizes  $ew_4$  because it is on its wait list, so task EC4 resumes, executes to completion and returns to the Wait Coordinator.
- (6) A return (as opposed to a call) to the Wait Coordinator implies reversion to a preceding stack frame of the Wait Coordinator (i. e., to a prior activation). Execution in the prior activation means that the Wait Coordinator can now recognize the arrival of  $ew_3$ . It may be noted that events  $ec_2$  and  $ec_1$  have also arrived. However, we are assuming a W-C polling order in the example. As a result,  $ew_3$  will be the first message discovered in the scan. As a consequence, task EC3 is resumed and completed.
- (7) & (8) The above reasoning may be repeated to see how tasks EC2 and EC1 may be completed in this order.
- (9) Upon completion of task EC1 and the return to the Wait Coordinator, events  $ec_2$  and  $ec_1$  are discoverable, since there are no event wait messages on hand. The message for  $ec_1$  is discovered first because it has higher priority causing EC1 to be invoked. (Thus endeth this 9-step walk.)

#### 7.5.4.2 Ways to prevent Sluggish Event Wait Response of Event Call Tasks

A serious shortcoming of the multi-purpose process should now be evident. A difficulty may arise any time an event call task is forced to call ipc\$block for an expected event. Even though that event may arrive with reasonable dispatch, there is no guarantee that the Wait Coordinator will "respond" in a reasonable length of time by giving this task an opportunity to resume. For instance, suppose

- (a) Task EC1 calls ipc\$block to event ew<sub>1</sub>.
- (b) The Wait Coordinator then invokes task EC2, and shortly thereafter ew<sub>1</sub> arrives. Then,
- (c) Task EC2 calls ipc\$block for an event ew<sub>2</sub>, which takes an unexpectedly long time to arrive.

Nothing can be done to give control back to EC1 even though its awaited message has long since arrived. (Changing the polling order does not help.) The difficulty stems from the fact that a stack history has been built up of the form:



It is impossible to resume EC1 without doing an abnormal return, i. e., from W<sub>3</sub> to EC1. But this action would have the effect of aborting task EC2, which could cause chaos.

Three approaches are open to the programmer to circumvent this problem. Two of these, (1) and (2), are less than fully satisfactory.

- (1) Program all associated procedures so that they and all their dynamic descendents (if any) execute no calls ipc\$block. (This will be difficult because if an associated procedure or any of its descendents calls a system library routine or one written by another individual, there is no easy way to be sure, without reading the code, if said targets do or do not call ipc\$block.)
- (2) Program each associated procedure so that
  - a. the first thing it does upon being called is to execute an ipc call that masks all event call channels, i. e.,

```
call ipc$mask_ev_calls (code);
```

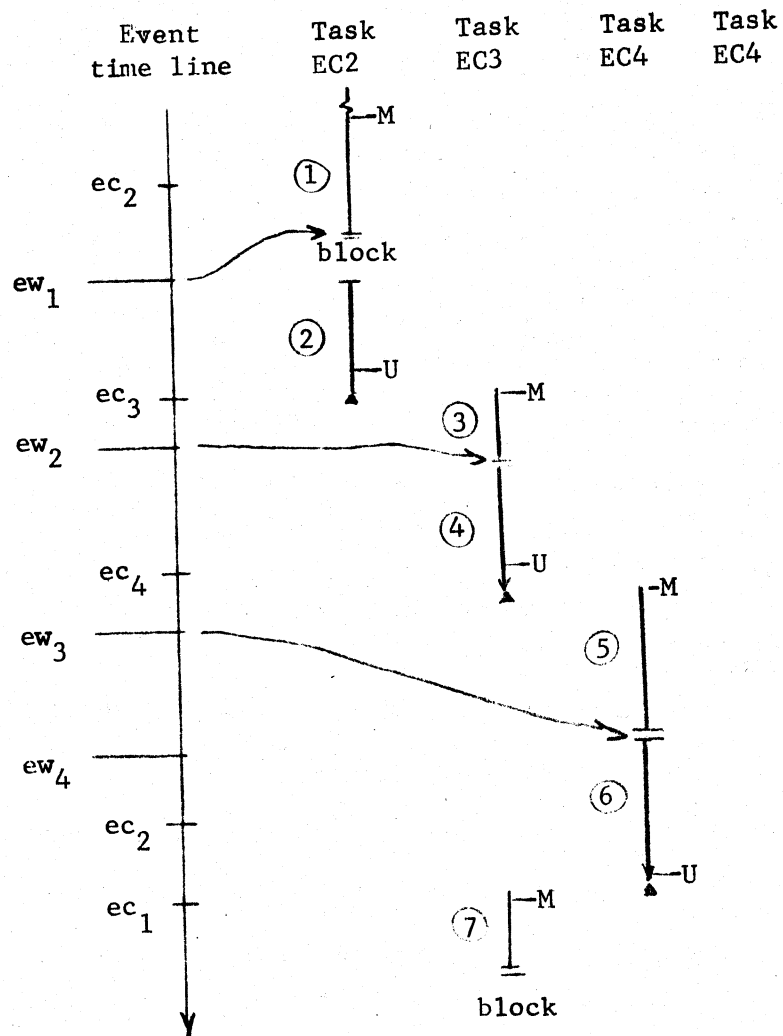


Figure 7-18 Same case as in Figure 7-17 except that calls to  $ipc\$mask\_ev$  calls and to  $ipc\$unmask\_ev$  calls are made at points marked M and U, respectively.

- b. the last thing it does, prior to returning, is to execute an ipc call that unmasks all event call channels, i. e.,

call ipc\$unmask\_ev\_calls (code);

This approach has the merit that any task that is invoked is treated as having absolute top priority. In a sense, it can be regarded as an extreme approach to solving the problem. Figure 7-18 illustrates what we mean. Here we show the effects of masking and unmasking event call channels using the same event timing sequence as in Figure 7-17. Note that task EC4, because of its low priority, is not even begun during the time span being considered.

- (3) This approach will no doubt prove to be the most attractive: Give up trying the multi-purpose approach in the first place! Go back to the principal-design approach of Multics and let each task that is now programmed as an event call task and in need of better response from its Wait Coordinator be made into an independent process to accomplish the same objective. Each such created process would have a single event call channel over which it can be signalled. Hence, competition for good response by its Wait Coordinator will now be eliminated.

Bear in mind, however, that after establishing separate processes for the several tasks, these can now, in principle anyway, be executed in parallel, whenever two or more processors can be awarded to these tasks during a single period of time. The mere fact that execution can proceed in parallel as a result of following this approach carries with it the need for greater care in the handling of shared data segments.