6-1-70

Identification

The traffic Controller Functions
S. H. Webber

Introduction

The traffic controller consists of procedures to recognize and act upon changes in the multi-programming environment. Some of these changes invoke certain functions of the traffic controller which are invisible to the user. Other changes are brought about by explicit calls to the traffic controller by the user. This section describes in detail the workings of the important functions performed by the traffic controller.

The Schedule/Reschedule Function

A major role of the traffic controller is to determine which process to run and how long to run it. It is a basic goal of the current scheduling design that highly interactive users be given better access to the hardware available than other users. To this end the traffic controller keeps track of a per-process switch, the interaction indicator. This indicator is turned ON when a process goes blocked yet deserves high priority (e.g., typewriter I/O) and is turned OFF the next time that process is scheduled.

The scheduling algorithm makes use of the eligible and priority queues described in BJ.1. These two queues can be thought of as one large queue the head of which is the eligible queue. When the traffic controller looks for a process to run it first searches the eligible queue and then searches the priority queue. If no process if found in the eligible queue and the maximum allowed number of processes are already eligible then an idle process is run; otherwise the first process in the priority queue is awarded

eligibility.

A process remains in the eligiblity queue until it either goes blocked or its allotted eligible time (temax) is exceeded. The method of determining the eligible time for a process is described below. When a process's eligible time does expire, the process loses eligibility and may be rescheduled (by being re-sorted into the priority queue at the appropriate place as described below). The highest priority process in the priority queue (if any) is then given eligibility.

The following is a brief summary of the scheduling parameters:

Per-Process Variables

1) te — is the amount of cpu time the process has run since eligibility was last awarded.

2) temax — defines the maximum value of te (i.e., if te ≥temax, the process may lose its eligibility)

3) ts — is the amount of cpu time the process has run since it was last scheduled.

4) tsmax — defines the maximum value of ts (i.e., if ts ≥ tsmax, the process is rescheduled).

5) ti — is the amount of cpu time the process has run since its last interaction.

6) timax — defines the maximum value of ti and serves to define the last scheduling queue which the process may be scheduled. This

variable is usually set only once from the system wide default value of timax. However, in special cases, the per-process timax may be set lower than the system value to give a process higher priority.

Per System Variables

1) te_first specifies the minimum value of the per-process valiable temax.

2) te_last specifies the maximum value of the per-process variable temax.

3) timax specifies the system default value for the per-process variable of the same name.

When a process is first scheduled after going blocked with the interaction indicator ON the following initial assignments are made to its scheduling parameters:

te, ts, ti = 0

tsmax, temax = tefirst

Once a process begins executing, these variables are changed as pictured in the flow chart below (Figure 1). These updates are made whenever a process gives away the processor (i.e., page faults, blocks, timer runout, pre-empts, etc.). Whenever a process has run for more than tsmax seconds it is rescheduled. This rescheduling consists in updating ti and resorting the process in the priority queue.

A process is sorted in before all processes with a larger ti and after all processes with an equal ti. A process, however, is never sorted into

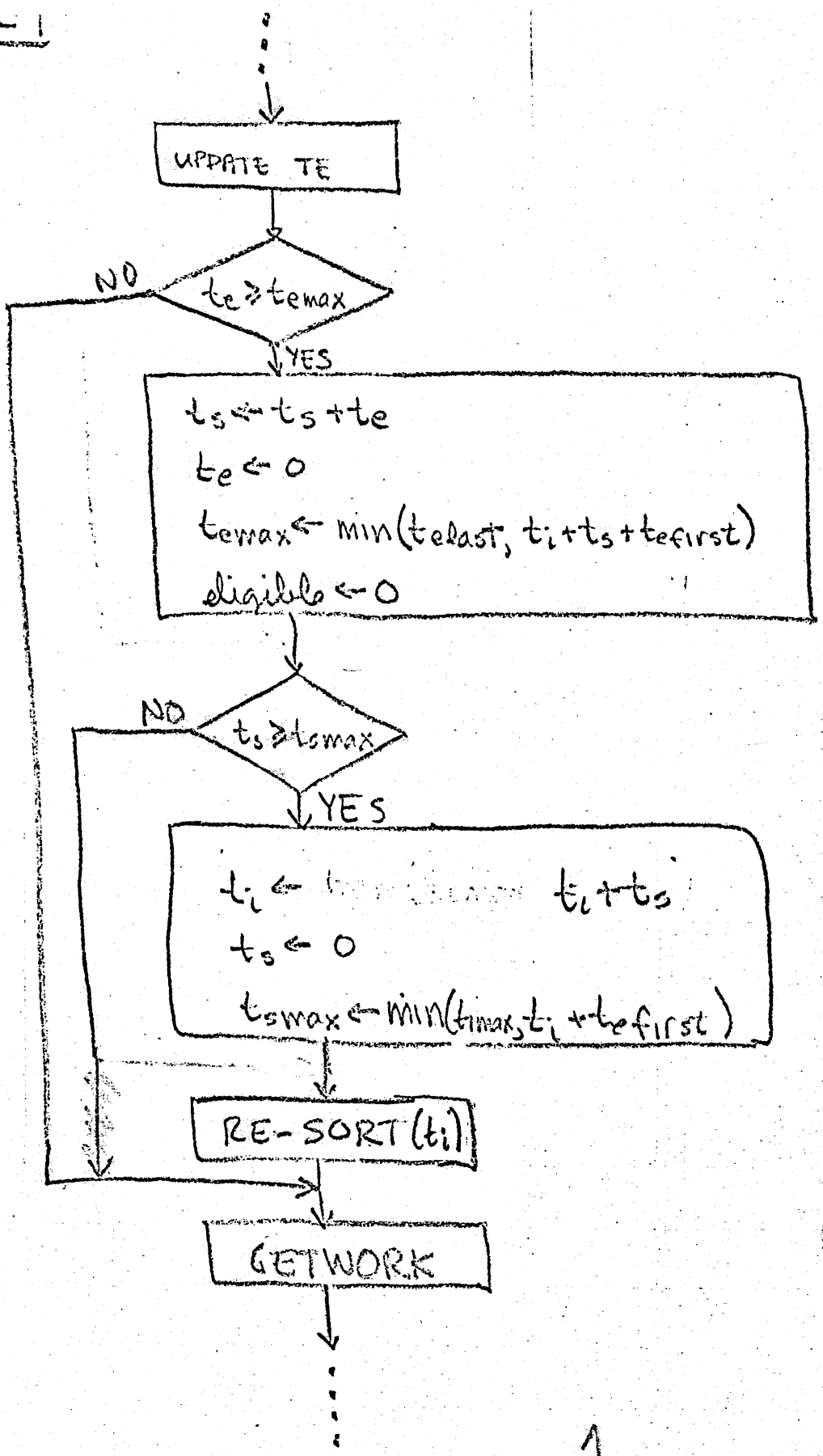

UPDATE TE
NO
te ≥ temax
YES
ts ← ts + te
te ← 0
temax ← min(telast, ti + ts + tefirst)
eligible ← 0
NO
ts ≥ tsmax
YES
ti ← ti + ts
ts ← 0
tsmax ← min(timax, ti + tefirst)
RE-SORT (ti)
GETWORK

FIGURE 1

the eligible queue. The effect of this algorithm is to give the processes which have run the most (since interacting) the lowest priority.

If a process completes its computation in its first eligible time it will be awarded relatively high priority. This is because the process will go blocked waiting for typewriter I/O which is considered an "interaction". This in turn causes the interaction indicator to be turned ON which will cause the traffic controller to set that process's ti to 0. When the process is later scheduled (after a wakeup) he will be sorted into the priority queue with a ti of 0 which results in high priority. In fact only the eligible processes and other processes already in the queue with a ti of 0 will have a higher priority.

To illustrate the algorithm consider the following two representations of the queues seperated by a small amount of time.

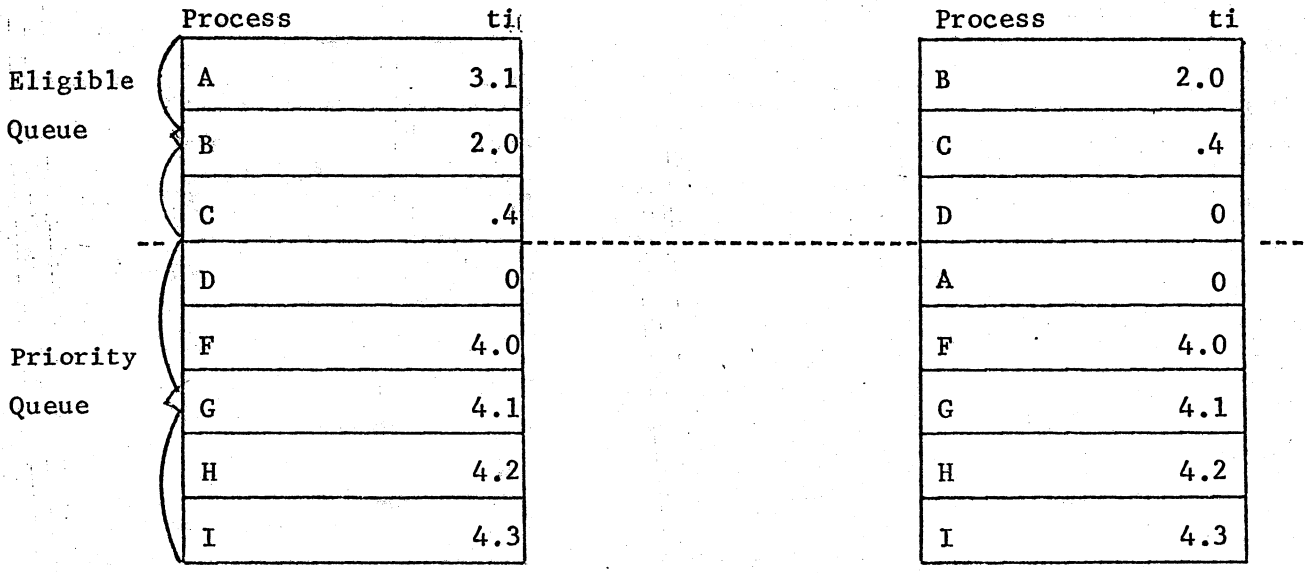


Figure 2.

Process A had highest priority, finished its computation in its first eligible time, and then immediately requested the processor again. Since process A was re-sorted with a ti of 0 it was given fairly high priority. Process A will run again before any of F, G, H, or I, and hence will have better response. Process I in fact may not run for quite a while. When process I does run, however, it will be given a temax (and tsmax) large enough so that a substantial amount of work can be done before again losing eligibility and the processor to the interactive users.

As can be determined from the flow chart the following values of temax and tsmax will be used for compute bound processes.

Scheduling	temax	tsmax
1	tefirst	tefirst
2	min(telast, 2*tefirst)	min(timax, 2*tefirst)
3	min(telast, 4*tefirst)	min(timax, 4*tefirst)
4	min(telast, 8*tefirst)	min(timax, 8*tefirst)
.	.	.
.	.	.
.	.	.

If tefirst is 1 second, telast is 4 seconds and timax is 8 seconds this

would give

Scheduling	temax	tsmax	ti
1	1	1	0
2	2	2	1
3	4	4	3
4	4	8	7
5	4	8	8
6	4	8	8
.	.		
.	.		
.	.		

Note there are effectively 5 queues with these settings. The queues correspond to the following values of ti: 0, 1, 3, 7, 8.

If tefirst, telast, and timax had the values of 4,4 and 8 seconds respectively this would give.

Scheduling	temax	tsmax	ti
1	4	4	0
2	4	8	4
3	4	8	8
4	4	8	8
.	.		
.	.		
.	.		

Note there are effectively 3 queues in this case corresponding to the following values of ti: 0, 4, 8.

Under either of these schemes, once a process falls to the lowest queue it will remain in the priority queue for 8 seconds without being rescheduled. It will however use this 8 seconds up in two 4-second eligibility times.

The lowest priority queue mentioned here is called the "last scheduling queue" and is determined by the per-process variable tlast

Timer Runouts and Pre-empts Note that all process with the same tim which have exceeded more than tlmax seconds w be scheduled in a round robin manner the

The schedule/reschedule function is invoked by any process which must give away the processor. This is done by explicit calls into the traffic controller. When a process's eligible time is up, however, the process must be forced to give up the processor. This is done with a hardware interrupt which is set to go off while the process which has exceeded its time allotment is actually executing. The interrupt causes control to be directed straight to the schedule/reschedule code and then to getwork. Note that if no higher-priority process exists the interrupted process will be chosen to run again immediately.

In a like manner a pre-empt interrupt is used to award the processor to a high priority process when its pages arrives in core. The lower priority process currently executing receives a pre-empt interrupt and control is channeled exactly as for timer runout interrupts.

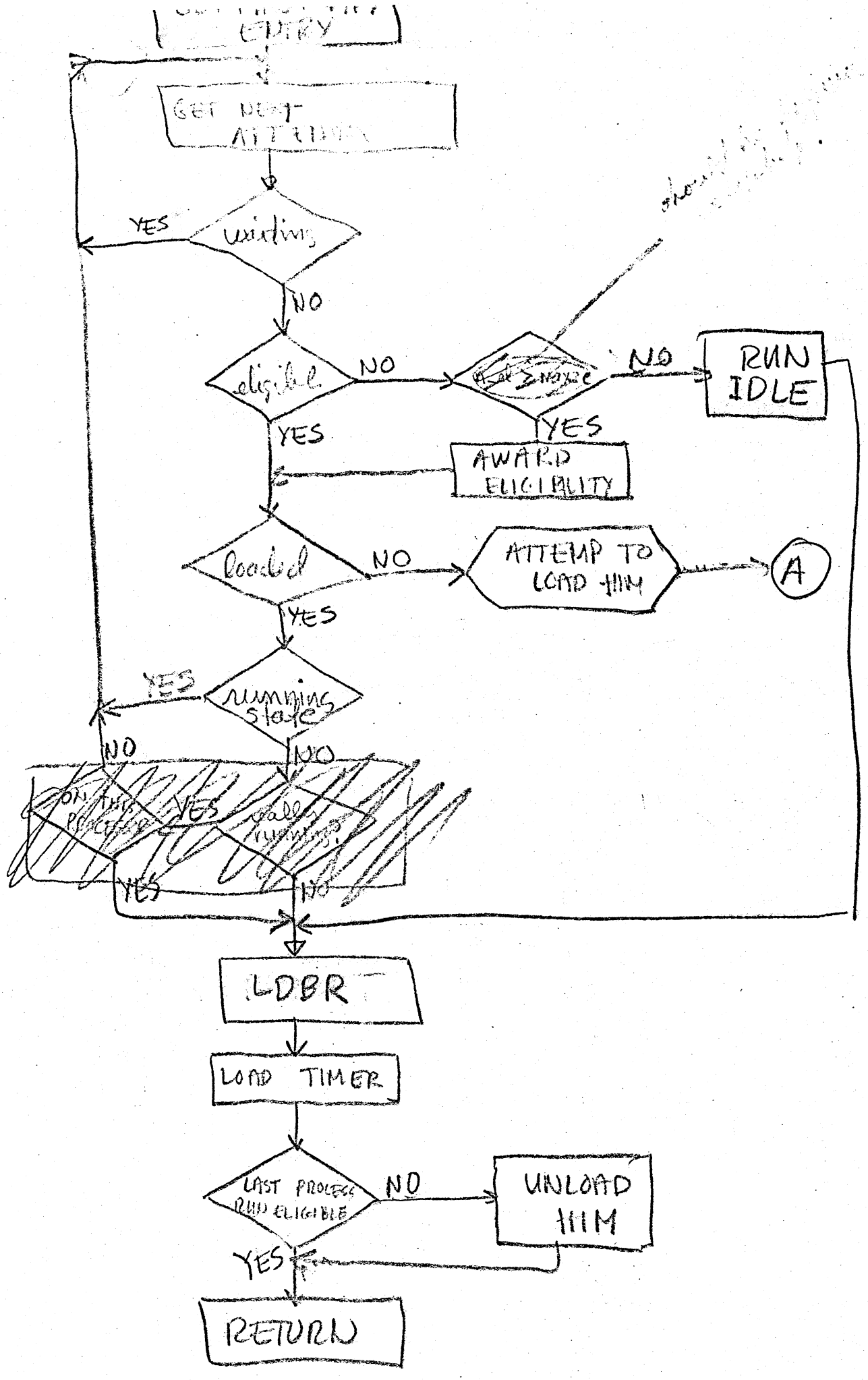
The Getwork function

After a process which is giving the processor away has invoked the schedule/reschedule function, the queues are correctly sorted by priority. It is the job of the getwork function to run the highest priority process

it can. The algorithm is simply to search first the eligible queue and then the priority queue.

There are several reasons (not loaded, waiting, not correct processor, etc.) why a process which has relatively high priority can not be run. If getwork finds one of these it skips the process and goes to the next process in the queue. There are other processes which need some function performed by getwork before they can be run, e.g., the process may need to be loaded. If getwork is successful in satisfying such a process's need the process will be run, otherwise it is skipped and the next process is looked at. If getwork can find no process to run, either because there are none wishing to run or because those which do want to run can't, it runs an idle process.

The flow chart (Figure 3) describes in more detail which tests getwork uses in deciding which process to run.


ENTRY
GET NEXT ATTEMPT
waiting
YES
NO
eligible
NO
YES
AWARD ELIGIBILITY
RUN IDLE
loaded
NO
ATTEMP TO LOAD HIM
A
YES
running state
YES
NO
NO
ON THIS PROCESSOR
YES
call running
YES
NO
LDBR
LOAD TIMER
LAST PROCESS RUN ELIGIBLE
NO
UNLOAD HIM
YES
RETURN

Note that if getwork fails to award eligibility to a process (because the maximum number are already eligible) it has exhausted all chances of finding a process to run and an idle process must be run. The attempt to load a process may or may not succeed. If it does not succeed on one pass it will on some other pass at a later time because a notify will occur for each page needed by the process being loaded. (The attempt to load a process checks to see if all pages needed are in core and if not it puts the process in the wait state where it will remain until all notifies occur.)

The Wait and Notify Functions

The wait and notify functions are provided for the use of the basic file system and other hardcore modules. The wait function is invoked when a process cannot proceed, mainly for either of two reasons: first, a page must be brought into core, and second, a hardcore data base must be unlocked. The notify function is invoked when the page arrives or when the data base is unlocked. When a process is notified of the event it is waiting for -- it must be eligible -- it is placed in the ready state and a check is made to see if another process should be pre-empted for its processor. If so, the pre-empt is sent and the highest priority process which can run is given the processor when the interrupt occurs.

Several processes could be waiting for the same event. When this occurs the notify places all such processes in the ready state and the highest priority process in the eligible queue is run.

The Block and Wakeup Functions

During the 'life' of a Multics process, the need may arise for this process to have some information furnished by some other process; we say that this process is engaged in "interprocess communication". Interprocess communication implies a synchronization of processes; a process might have to 'pause' (idle) for the other process to communicate the information. By convention, for reasons of efficiency, such a process gives its processor away, or blocks itself, until the awaited information has been communicated, or until that specific "event" has occurred. It is then taken out of the blocked state and put into the ready state, i.e. "awakened". The Traffic Controller functions block and wakeup provide these basic functions.

3

An event is anything that is observed by a process and which might be of interest to another process (or maybe another procedure of the same process). An event is always associated with some information to be communicated to the interested (receiving) process. Examples of events are: the terminating of a computation, the unlocking of a user-shared data-base or the arrival of new input from an I/O device. These events happen outside of the hardcore ring and are known as "user-events".

Process A reaches a point in its execution where it cannot proceed until event E has occurred (or in other words, until some information is furnished by some other process.) It therefore invokes function block and abandons the

the processor. Process A is now in the blocked state, ~~which means that it no longer~~ (it loses eligibility) and will remain in that state until awakened by some other process. Process B now executes and observes an event. This could be event E for which process A is waiting, it could also be any other event Q in which process A might be interested some time in the future; the point is, even though process B knows that the observed event is of interest to process A, it has no way of determining what process A's current state is, whether it is waiting for some event or whether it is executing. Consequently, the notification mechanism must be such as to allow the preservation of all communicated information even though it might not be of immediate interest to the receiving process.

Process B invokes the wakeup function specifying that process A should be awakened because event E has occurred. After A wakes up it returns from the traffic controller and finds the information communicated by process B (namely event 'E'). If that information is the one it has waited for, it continues its interrupted execution, otherwise it stores that information somewhere in its memory-space, and invokes block again.

Both block and wakeup manipulate the Active Process Table (APT); normally, block puts the APT entry of its own process into the blocked state and removes the entry from the queues, wakeup finds the APT entry of the target process and restores it into the priority queue. However, it is not guaranteed that a call to wakeup in behalf of

some process will actually find that process in the blocked state; also, it is not guaranteed that if a process calls block because it is waiting for some event to happen that this event will happen in the future, it might already have happened in the past. Evidently, some further interaction is needed between functions block and wakeup to insure that event signals do not get lost, and that a process will not mistakenly block itself, never again to be awakened.

This interaction is provided by the process's 'wakeup-waiting' flag in the process's APT entry. A call to wakeup always sets this flag ON. Then, if the process is blocked, it will be put into the priority queue, else it is left in whatever state it is in. A call to block will actually cause the process to abandon its processor only if its wakeup-waiting flag is OFF; the flag's ON state indicates that an event signal (which might be the one awaited) has already occurred, and consequently block returns to its caller. Upon returning, subrouting block always resets its wakeup-waiting flag to OFF.

Associated with block and wakeup is a system-wide data-base known as the Interprocess Transmission Table (ITT). This table contains as many event queues as there are receiving processes in the system. Every receiving process has in its APT entry the head of its ITT event queue.

Invoking block, before returning, detaches the ITT queue from the process's

APT entry (providing the process with a fresh, zero-length queue) and returns the detached queue to its caller, which then copies the queue's contents into its own memory space and frees the ITT space for future use.

The Start and Stop Functions

The purpose of the stop function is to bring a process to an orderly stop and leave it in a consistent state. Once in the stopped state the process can be destroyed, started, or saved for later restarting. Since much time may pass between the time a process was stopped and the time it is restarted a stopped process must leave all supervisor data bases in a well-defined, consistent state. This means that a process can not be stopped while in the hardcore ring. To guarantee this a special hardware interrupt is used to force a process to invoke the traffic controller code to stop itself. This interrupt is masked (prevented from going off) in the hardcore ring and hence any process receiving the interrupt is in a satisfactory state to be stopped. Note if a process is blocked it is necessary to awaken it so that it will return to the procedure calling block. Block consequently can not be called in the hardcore ring.

The start function is used to restart a stopped process. It is invoked by an explicit call to the traffic controller with the name (processid) of the process to be started passed as an argument.

Process Creation and Destruction

Process creation consists of two steps; First creating those per-process segments needed by the process in order to handle segment, page, and linkage faults, and second, to enter the process into the traffic control data bases. The per-process segments consist of a hardcore descriptor segment for the process as well as the following segments, place in the "process directory" for the process:

1) pdf — a segment used to store pageable information about the process and also used as the ring 0 stack for the process. This stack is used in handling segment and linkage faults.

2) pds — a wired down segment (wired when the process is loaded) containing information about the process which is needed at page fault time. In addition the pds contains the stack used by the traffic controller when executing in this process.

3) kst — (known segment table) this segment contains information about segments needed to handle linkage and segment faults. It contains entries (indexed by segment number) which have information about segment names and access.

To enter the process in the traffic control data bases a processid must be created and an APT entry must be allotted. The APT entry is then initialized for the process and the execution state set to "blocked".

After the entire environment is created for the process it is sent a wakeup which causes it to be placed in the ready state in the priority queue. When the process is subsequently run it finishes initializing itself having the capability to create a user ring (containing stack, combined linkage segment and descriptor segment).

Note that when a process is loaded under normal running the hardcore descriptor segment, user ring descriptor segment and pds must all be wired down. The process can then take page faults to bring in any segments needed to handle segment and linkage faults.