TO: MSPM Distribution

FROM: J.H. Saltzer

SUBJ: New BJ.0

This version of BJ.0 removes a number of omissions present in the previous overview and makes several small corrections to minor details. The technical content is unchanged.

To
Corbato
Daley
Padlipsky
Ready to Publish.

Jerry

Insert Q for Page 1

A second M.I.T. thesis, "Implementing Multiprocess Primitives in a Multiplexed Computer System," by R.L. Rappaport, describes in detail the many considerations which went into the actual implementation of the Traffic Controller.

Published: 10/01/68
(Supersedes: BJ.0, 02/23/67)

Identification

Overview of Traffic Control
J. H. Saltzer, R. L. Rappaport, M. J. Spier

Purpose

This section presents a general summary of the procedures of the central supervisor that perform processor multiplexing, interrupt management, and inter-process signalling. The procedures are known collectively as the Traffic Controller.

References

Basic concepts of the Traffic Controller are set forth in the Project MAC Technical Report "Traffic Control in a Multiplexed Computer System", by ~~Jerome~~ H. Saltzer, MAC-TR-30, published July, 1966. This thesis presents the design approach to the Traffic Controller and is useful for background information.

Terminology

A process is basically a program in execution. The tangible evidence of a process is a processor stateword (a set of machine conditions) and an associated two-dimensional address space (a core image). The address space of a process, defined by a Descriptor Segment, determines the region of accessibility of the processor, both in execution of instructions and in obtaining data. A dynamic linking mechanism allows the process to change the contents and extent of its own address space, but this does not alter the fundamental view of a process as the execution of a program contained in the address space.

Within the system every process known to the system is identified by a unique number, its process I.D. This number is a key to a table of all known processes, which contains more information about each process.

Every process is in one of five execution states:

1. running
2. ready
3. waiting
4. blocked
5. stopped

A running process is at this instant executing in some processor. A ready process is one which would be running if a processor were available. A waiting process does not have immediate use for a processor, it is waiting for a system-event (for example, the arrival of a page into core) to happen within a predictable period of time. A blocked process is one which has no use for a processor; it is waiting for some event to happen sometime in the future. The event may be arrival of a signal from elsewhere in the system, or perhaps completion of a computation by another process. A stopped process is a blocked process that does not await events and which is guaranteed to have left its hardcore data bases in a predictable state.

Every process is or is not loaded into core memory. The definition of loaded is entirely an operational one. The "core image" part of a process may be stored in core memory, or in secondary storage, or split between the two. A process is defined as loaded only if enough of it is present in core memory that it may operate within critical supervisor modules.

To control competition for core memory,

An active process is one for which there is sufficient information in core storage to allow it to enter the ready state. The necessary information for an inactive process is stored on secondary storage, and must be retrieved before the process is allowed to enter the ready state. Operationally, an active process is one which appears in the Active Process Table.

~~to control competition for core memory,~~ For ~~administrative reasons~~, the number of processes that are allowed to concurrently participate in the race for a processor at any given time is limited. Such processes are said to be eligible for multiprogramming. ~~The~~ system administrator sets the number of eligible processes in the system. The definition of multiprogramming eligibility is entirely an operational one, and is defined in section BJ.6.00.

In the initial implementation, the

A number of things can happen to divert a process from its programmed course. These diversions have been variously termed traps, interrupts, and faults. We use the term interrupt when referring to hardware signals coming from outside the processor which cause a processor to depart from its normal path of execution. Interrupts are distinguished from faults, which are triggered by hardware signals generated within the processor.

Processor multiplexing includes both the sharing of processors among many users to provide interactive response (sometimes called time-sharing) and the switching of processors among several procedures in response to interrupts so as to keep both processors and I/O devices as efficiently used as possible (sometimes called multiprogramming).

The Traffic Controller

The Traffic Controller is a set of procedures appearing within the address space of a process.

The functions provided by the Traffic Controller are intentionally primitive; it is viewed as the innermost layer of a multilayered supervisor existing within a process. In fact, a user's program is never permitted to call the Traffic Controller entries directly. Instead, the user's program calls some outer supervisor layer which, for example, checks the authority of a call to signal another process.

The rest of this document will describe the Traffic Controller as though it is used directly by some "customer". It is understood, however, that its only "customers" are actually other supervisor procedures.

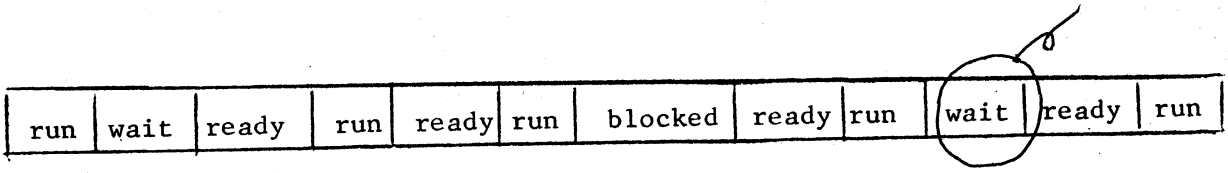
The Traffic Controller can be conveniently broken into two distinct parts which perform its major functions:

1. The system interrupt interception routines
2. The process exchange

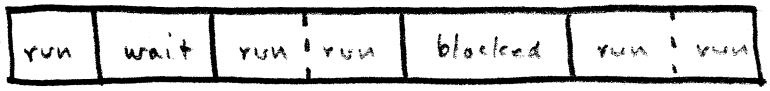
The three major functions of the Traffic Controller are the following:

1. Perform multiplexing of processors among processes
2. Provide an interface with the system interrupt hardware
3. Allow one process to signal another.

An important function of the Traffic Controller is processor multiplexing. To visualize this multiplexing, consider the progress of a process, as seen by the system. As time passes, the process goes back and forth among the running, ready, waiting and blocked states as in the diagram below:



The Traffic Controller has inserted the ready states in order to multiplex, or share, the processor among all the processes demanding service. The process, however, does not normally observe the times spent in "ready" status. From the point of view of this particular process, the above diagram looks like this:



with dotted lines indicating points at which the calendar clock takes a quantum jump. Multiplexing is arranged so that, except for the real time clock jumps, it is basically "invisible" to the affected process. This means that a process can completely ignore the multiplexing being performed by the supervisor. It also means that a process must be substantially independent of timing. A further implication is that service to critically timing-dependent hardware functions must be provided by the Traffic Controller itself.

The Traffic Controller has two interfaces: on the one side with the system interrupt hardware, and on the other with the rest of the supervisor and the user's program. The hardware interface is described in detail in the section on interrupt handling, BK.

The interface with the rest of the system consists primarily of seven calls into the Traffic Controller. (There are also several less important entry points concerned with process synchronization, process creation, and processor-resource management. These entries do not affect the significance of this discussion and can be ignored for the moment.) The seven calls can be classified into three groups as follows:

1. Process Wait and Notify (PWN) calls: wait, notify.
2. Interprocess Communication (IPC) calls: block, wakeup.
3. Process interrupt calls: re-schedule, pre-empt, stop.

The Process Wait and Notify Calls

Every process reaches a point in its execution where it has to have information from some other process; if the information is unavailable, it abandons the processor on which it currently executes until such time as the information will become available, or until that event happens. We name "event" anything that is observed by some other process and which is of interest to our process. We distinguish between two classes of events, system-events and user-events. This distinction is made ~~for reasons that are largely implementation-dependent.~~ (~~Theoretically,~~ In principle, events are all of the same nature and can be handled uniformly.)

System-events are characterized by the fact that they can be observed in the hardcore ring only and that they are guaranteed to happen within a predictable period of time (normally measured in milliseconds.) These include the arrival of a page into core, or the unlocking of a currently-locked system-wide data base.

A process that has to wait for a specific system-event calls

call wait (event);

This call puts it into the waiting state and associates it with "event" so that when some other process observes the occurrence of "event" it calls

call notify (event);

which causes all the processes which are currently waiting for "event" to be restored into the ready, and eventually the running state. As can be seen, the PWN calls are event oriented. PWN is discussed in detail in sections BJ.2.

The Interprocess Communication Calls

A process may wish to give its processor away until it be notified of the occurrence of a user-event. Typical of a user event is that it may happen anytime in the future; also, a user-event is process-oriented (the signalling is done towards a specific process rather than "generally broadcast") and is always associated with some information.

Entry point block of the Traffic Controller is called by a process when that process cannot proceed until a signal in the form of a wakeup from another process arrives. It is the responsibility of the process calling block to insure that some process will indeed wake it up. Block is called with two arguments:

```
call block (interaction_switch, event);
```

The Traffic Controller will place this process in blocked status, where it will remain until some wakeup signal arrives for it. The "interaction_switch" indicates whether or not the process is blocking itself while interacting with a human being, in which case the process will be given a higher-than-usual priority in its race for a processor, when awakened, to insure quick system response to human requests; "event" is a queue of event-messages, returned by the Traffic Controller.

The entry name wakeup is used whenever a process wishes to wake up a blocked process. The wakeup, by definition, is directed to some named process as a result of the observation of some user-event. A typical call from within process "Q" to wake up process "B" and inform it that event "E" has happened would be

```
call wakeup (B,E,B-state);
```

Process "B" may be running, ready, waiting, blocked or stopped at this time. Although the information associated with event "E" will not be lost to process "B", the call will have effect only if "B" is blocked, in which case it will be restored to the ready state, or awakened. Return argument `B-state' reflects process B's current execution state.

What happened to the definition of process interrupt and interrupt?

Scheduling

Whenever a process gives up a processor, it first establishes a time-allotment and priority level number to be used when the process is next to ~~XX~~ compete for a processor. This advance establishment of conditions of the next running is known by the (misleading) name of scheduling. ~~XXXX~~ The priority number so assigned remains valid for the duration

of the ~~current~~ next time-allotment. Whenever a process is to wait~~ing~~ for a human response (interaction), it gives itself the highest-possible priority; as time passes (with each timer-runout), its priority ~~decreases~~ until it reaches and maintains the lowest possible priority.

Interrupt Handling

The underlying philosophy of interrupt handling is that interrupts are signals similar in nature to wakeup calls, but originating on external hardware equipment. Thus, the sole function of the interrupt handling routines is to transform an interrupt into appropriate calls to the Process Exchange. As an example, for an interrupt representing the completion of a write operation on a typewriter, the interrupt handler would call wakeup for the process which originated output to the typewriter, signalling an event-name which is associated with that typewriter. No other computation is done at the instant of the interrupt. The process "responsible" for the interrupt (in the example above, the process initiating I/O on that typewriter) is restored into the ready state by the wakeup call; computation in response to the signal (data transformation, etc.) is not accomplished until the responsible process begins execution.

There are two categories of interrupts. Those arising from external equipment, as just described, are called system interrupts. Those originated inside the system itself, by the Traffic Controller, are known as process interrupts. The process interrupts generally call for more drastic action then ~~working~~ waking up some process; they are orders to the process currently running that it should change its execution state.

Process Interrupt Calls

there are three types of process interrupts:

1. the timer runout ~~interr~~ interrupt
2. the pre-emption interrupt
3. the stop interrupt.

(A) Each one of the three process interrupts causes the target process to divert its execution into the Traffic Controller, where it gives its processor away. The timer-runout interrupt is generated by the hardware whenever a process' time-allotment runs out. The other two interrupts are software-initiated. The pre-emption interrupt is set in behalf of a ready high-priority process when it is observed that a lower-priority process is currently executing on a processor; the running process is thus forced to abandon the processor in favor of the ready high-priority process. The stop interrupt is set by a process when it wishes to halt the execution of some other process.

When a process is initially made to run, it is given a certain time allotment which the hardware keeps track of. When this time has been used up, a process-interrupt is generated which diverts the process' execution into an interrupt handler which then calls the Traffic Controller's entry point

(or a pre-emption interrupt occurs)

```
call reschedule;~~(interaction_sw);~~
```

to reschedule the process, give it a fresh time allotment and put it into the ready state. A process always reschedules itself (when calling "reschedule" or "block(1)") before giving its processor away rather than when being put on the ready list. In fact, a process that is rescheduling itself is computing at the time it abandons its processor what its priority will be when it is put on the ready list once more. ~~(Interaction_sw is the above-mentioned argument to block)~~.

A currently executing process may sometimes have to abandon its processor, even though its time-allotment has not yet run out. It is the scheduler which decides whether or not to pre-empt a running process when a higher-priority process enters the ready state. The decision is based upon the length of time that the current process has already run. For reasons of efficiency, there are two different algorithms in the scheduler; one is used when the ready process is a system process, the other when it is not.

However, regardless of how the decision was reached, when it is decided to pre-empt a running process then a pre-emption interrupt is set which causes the target process to stop its execution and to release the processor.

~~call pre-empt (processor);~~

~~sends a pre-emption interrupt to the process which is currently executing on `processor'.~~

Sometimes, a process may wish to halt another process' execution. If process "A" wants to stop process "B", it calls

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

which will put process "B" into the stopped state. By convention, process "B", if currently executing in behalf of the system, is allowed to finish its current system task before it is stopped. This is done in order to insure that a stopped process always leaves hardcore databases in a predictable state.

Entry point reschedule and the the scheduler are discussed in section BJ.5; stop is discussed in section BJ.4.

Interaction with the File System

The operations of processor multiplexing interact with those of core memory multiplexing. The multi-programming control ("eligibility") mechanism guarantees that the Traffic Controller will not attempt to multiplex processor capacity among so many processes that memory becomes too crowded. To this end, a little-used ineligible process may be unloaded by the traffic controller system process (see BJ.6) if space becomes too tight; when an unloaded process comes to the top of the ready list it will not be reloaded until adequate space is available for it to run efficiently. Unloading is accomplished by paging out the remainder of its descriptor segment and other segments needed to enter the running state; the process is remembered only by its entry in the Active Process Table. Loading and unloading is done by a special (and never unloaded) system process known as the Traffic Controller System Process (TCSP).

Multi-programming control and the TCSP are described in section BJ.6.

Process Control

Process Control

In addition to the Process Exchange and the interrupt handling procedures, the Traffic Controller contains a "housekeeping" module, known as Process Control. This module provides entries to

~~1. Initialize the rest of the Traffic Controller and the processor hardware.~~

1. 2. Create new processes.

2. 3. Delete old processes.

~~4. Interface with the basic file system to perform core memory multiplexing.~~

3. 5. Simulate an execution meter (processor usage meter) for each process.