Published: 10/01/68
(Supersedes: BJ.4.00 01/30/67)

## Identification

Overview of stop
Robert L. Rappaport, Michael J. Spier

## Purpose

Sometimes a process may wish to halt another process' execution. A typical example is that of a system operator who wants to stop all the processes in the system prior to a general system-shutdown. We name this operation the `stopping' of a process; as a result of it, the target process is put into the `stopped' state.

Technically, a stopped process is similar to a blocked process, the difference being that a process is forced into the stopped state, and that it enters this state without the assurance that it might ever come out of it again.

## Discussion

As mentioned above, a stopped process is not guaranteed to ever run again; indeed, a stopped process is more often than not on its way to being saved or destroyed. Consequently, putting a process into the stopped state must include safeguards to assure that the sudden disappearance of this process from the midst of an interacting multiplexed computer system will not cause any damage to the system.

By definition, a process is the execution of a virtual processor within the boundaries of a private memory space; consequently, a process can cause system-damage only in places where its memory-space overlaps the memory-space of one or more other processes. By convention, all systemwide supervisor data bases are located in the hardcore ring. Rather than try and keep track of a process' execution in order to be able to find out whether or not it is currently manipulating such a data-base, it has been agreed never to stop a process while it executes in the hardcore ring. Thus it is guaranteed that a stopped process always leaves systemwide data-bases in a predictable state.

Another delicate subject is that of restarting a stopped process, for example following a system shutdown, or after a user has been subject to an automatic logout. If a stopped process is "stopped dead in its tracks" then in order to be able to restart it we would have to conserve its stack history symbolically (machine addresses are no good because by the time the process is restarted, one or more of the hardcore procedures which it was executing might have changed).

Rather than go into all that expense, a scheme is used which guarantees a standard (and known) ring 0 history for all stopped processes; thus in restarting a process one can reconstruct the process' history without actually having had to remember its stack.

Stopping strategy

The Traffic Controller entry point `stop' is called whenever a process wishes to halt another process (or possibly itself.)

call stop(B)

where `B' is the target process' ID, sets in that process' Active Process Table (APT) entry a flag known as the `stop_pending' flag, and sends a wakeup to that process. Whenever a process is chosen to run, that flag is placed in the processor's interrupt cell which is assigned to the stop interrupt.

A process is masked against that interrupt for as long as it executes in the hardcore ring. Whenever a process abandons its processor, the stop interrupt cell is remembered in the process' stop-pending flag. This strategy insures that a process does not lose the received stop signal, and that it will not be affected by that signal as long as it is in the hardcore ring.

As soon as the process executes outside of the hardcore ring (as soon as the stop interrupt is unmasked) it gets `hit' by the stop interrupt; it diverts its execution into the stop-interrupt handler which in turn calls the Traffic Controller.

call i_stop

Subroutine i_stop puts the process' APT entry on the list of blocked processes and gives its processor away. Thus the process is made to stop itself and therefore leaves its ring 0 history in a known state.

There is only one case in which a process must be stopped within the hardcore ring. It is the case of a blocked process which happens to be in ring 0 because it called block in behalf of the user. In order to allow the stopping of such a process, subroutine stop always send a wakeup signal to the target process, and whenever a process returns from subroutine block it gets momentarily unmasked against stop interrupts. In this way, if a process is in ring 0 in behalf of the user, it does get `hit' by the stop interrupt which makes it call i_stop.

One of the reasons why a process must not be stopped in the hardcore ring is that a stopped process can be restarted anytime in the future. The hardcore ring is pre-linked and shared by all processes. When a stopped process gets restarted in the future, it is possible that the hardcore ring might have been changed in the meantime (procedure recompiled, segments bound etc). If such as restarted process goes on executing on its old ring-0 stack, it might cause damage to the system. Therefore it is a system rule to never stop a process which has an unpredictable ring-0 stack history.

As explained above, a process may be stopped only if it executes either outside of the hardcore ring, or in the hardcore-ring in the wait coordinator.

When a process is restarted, it can be determined which one of the two possible stack histories it had, and a dummy stack history can be provided which would be sufficient to get the process out of the hardcore ring. A process' actual ring-0 stack must never be used for restart purposes and is therefore never saved.