Published: 02/05/68

Identification

Quit inhibition
C. Marceau, P. Belmont

Purpose

Certain procedures which execute in the administrative ring of a user's working process may need to be protected from interruption by a quit originated by the user from his console. An example is the Reserver, which must not be "quit out of" while it has its system-wide reservation tables locked. Ring one modules which are sensitive to quits may cause the process in which they execute to be unquittable for short periods of time. Procedures executing in system processes may also call to inhibit quits. For such processes quit_inhibition is meaningless (since there is no console user issuing quits) and the call to inhibit quits returns immediately.

Discussion

Associated with each working process in the Working Process Table of its process-group is a counter called the quit inhibit counter. The entries described in this section increment and decrement the counter respectively. The counter is observed by the stop procedure (see BQ.3.03) and the destroy_wp procedure, which will quit a process only if its quit inhibit counter is zero.

Usage

To inhibit quits for a short period of time, an administrative ring procedure calls

```
call quit_inhibit$on;
```

When it has ceased to be sensitive to quits, the procedure calls

```
call quit_inhibit$off;
```

Restrictions

Quit_inhibit should be used only when it is deleterious to the system that a process be quit. Further, it is essential to ensure that at some later time the process is again quittable, so that we don't end up with a runaway process that cannot be quit.

The necessary safeguards take the form of several programming conventions (see below) and a system-imposed restriction. The programming conventions apply for ring one programs which call quit_inhibit. These programs may call other programs, which must return within a finite length of time. Ring 1 programs are guaranteed to return within a finite length of time (i.e., are in accord with conventions 4 and 5 - see below.). But outer ring programs are not so guaranteed.

Hence the restriction, enforced by the ring crossing mechanism, that <u>no ring 1 procedure may call an outer ring procedure while quits are inhibited</u>.

<u>Conventions</u>

The following conventions are hereby established for administrative ring procedures which call quit_inhibit$on.

1) A procedure should declare itself unquittable only when not doing so would have serious repercussions in the system (i.e., it is not a sufficient reason that a programmer thinks it would be fun to declare the procedure unquittable).

2) A procedure should declare itself unquittable for the smallest necessary period of time. It should probably not, for example, begin with a call to quit_inhibit$on and end with a call to quit_inhibit$off.

3) A procedure which calls quit_inhibit$on must later call quit_inhibit$off.

4) No infinite loops may lie between the two calls to quit_inhibit.

5) No calls to wait for events which may never happen may lie between the calls to quit_inhibit.

6) The procedure may establish a condition handler for the condition "inhibited_ring1_exit" prior to calling quit_inhibit$on (see below).

<u>The "inhibited ring1 exit" condition</u>

Despite careful programming it may happen that one day procedure x calls quit_inhibit$on, then calls procedure a. A calls b calls c calls d, and d is in an outer ring. The gatekeeper catches the attempted ring crossing and signals an "inhibited_ring1_exit" condition.

A procedure may be able to recover from such an event, and has the opportunity to do so if it establishes an appropriate handler.

The default handler first observes whether this is a system process or not. In a system process (one which cannot be quit anyway, and in which the call to quit_inhibit should not have changed the quit_inhibit counter) it makes a record of the event using standard system trouble recording procedures, and returns, allowing the ring crossing. In a user process, the default handler records the error and generates a terminate-process fault. Presumably the error is recorded in such a way as to attract the immediate attention of a systems programmer to correct the error.

Implementation

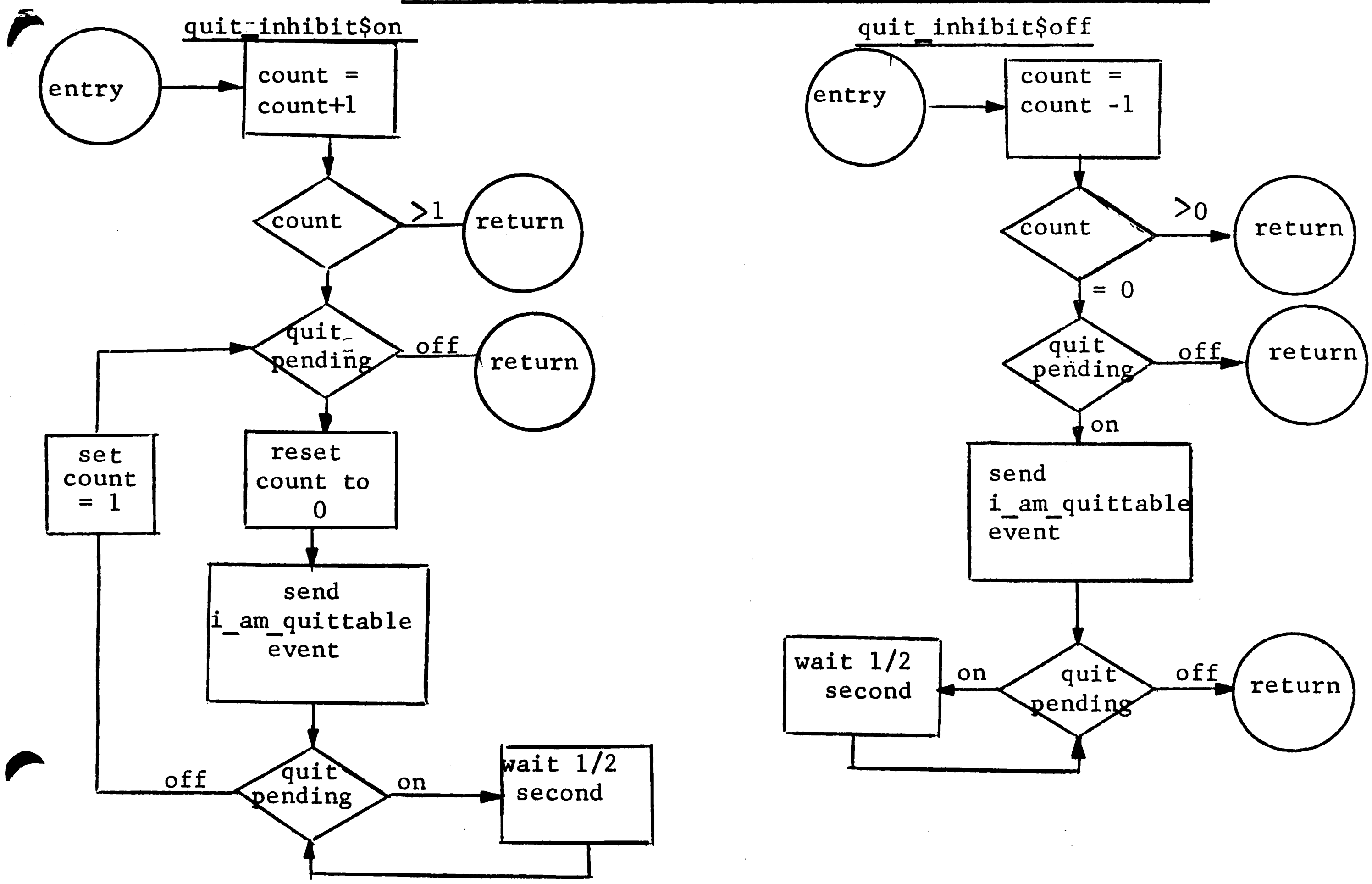
Quit_inhibit$on first increments its quit_inhibit counter by one. If the value of the counter is now one, it checks the quit_pending flag associated with its process in the working process table (see BQ.3.01). If the flag is up (the Overseer wishes to quit the process) then quit_inhibit$on resets the counter to zero, sends an i_am_quittable event to the Overseer Process, and begins a loop on reading the quit_pending flag. (That is, it waits 1/2 second and it reads the quit_pending flag. If the flag is up quit_inhibit loops again). At some time while it is looping, the process is quit. If the process is ever restarted, it will read the quit_pending flag and discover that the flag is down (the Overseer does not want to quit the process). Quit_inhibit then increments the quit_inhibit counter by one, checks the quit_pending flag again and returns to its caller.

Quit_inhibit$off first decrements the quit_inhibit counter by one. Then, if the counter = 0, it checks the quit_pending flag associated with its process in the working process table. If the flag is up (the Overseer would like to quit the process) then quit_inhibit$off sends an i_am_quittable event to the Overseer Process and begins to loop on reading the quit_pending flag.

Meanwhile the Overseer Process wakes up because of the i_am_quittable event and the stop procedure sees that the process is quittable and quits it. At some later time the Overseer may start up the process again. When this happens the process continues what it was doing, namely looping on the quit_pending flag. But now it discovers that the flag is down and returns to its caller.

Should quit_inhibit$on ever be called when the value of the quit_inhibit counter <0, or quit_inhibit$off when the value is <1, quit_inhibit notes the "impossible" occurrence by calling a system error handling procedure. If the error procedure returns, quit_inhibit$on sets the quit_inhibit counter to 1, and quit_inhibit$off sets the counter to 0, and then returns.

Figure 1 Flow of Quit_Inhibit Procedures and Quitting Procedures



How stop and destroy_wp quit a moving process.