1094

Bell Laboratories

subject: Cautions on the Use of the Alarm Signal
Case- 39380-6

date: April 26, 1976

from: T. G. Lyons
PY 9144
1A-129 x7346

PROGRAMMER'S NOTES

Two system calls have been installed on the PWB machines operated by Department 9144 which are not documented in the UNIX Programmer's Manual, Sixth Edition. Alarm(n) and pause() were coded at MH and are likely to filter down to standard UNIX, eventually, in their present form.

Alarm takes a single argument, a number of seconds. Alarm(n), when n != 0, arranges for signal number 14, SIGCLK, to be sent to the invoking process after n seconds. Alarm(0) suppresses a pending SIGCLK. In both cases the value returned by the call is the time remaining until any previously scheduled SIGCLK. Only the most recent alarm value is remembered by the operating system. Posting of SIGCLK turns off the alarm clock.

Pause is called with no arguments. Pause() suspends the invoking process indefinitely. A paused process may be reactivated only by the occurrence of a non-ignored signal.

The alarm signal may be used to interrupt the execution of a system call which has led to the prolonged suspension of a process. Alarm and pause together may be used to emulate a sleep system call, with less operating system overhead. Experience with these two mechanisms has uncovered certain dangerous aspects of the implementation of signals in UNIX, which will be described below. The alarm signal may have other applications of which the author is unaware and for which different cautions are appropriate.

Some UNIX system calls may lead to the suspension of the invoking process until a particular condition is satisfied. These conditions typically involve I/O completion, external events and internal synchronization. Whenever a process is suspended, it is assigned a numeric priority. A process suspended at negative priority cannot be reactivated until the associated condition is met. A process suspended at positive priority, however, may be reactivated by the occurrence of a non-ignored signal. In such an event, the partially completed system call is terminated abruptly. If the interrupting signal is not caught, the process to which it is posted is terminated according to the default action for the signal. If the signal is caught, the process is activated at the routine assigned to trap the signal. Return from the signal trapping routine leads to resumption of the main program at a point immediately following the interrupted system call. To the main program it appears that the interrupted system call has returned error number 4, EINTR.

Which system calls lead to suspension at positive priorities, and can therefore be interrupted, can currently be determined only by inspecting the code of the UNIX operating system. Among them, however, is pause(). Suppose that the subroutine null has been defined by `null() {}'. Then the following code should emulate sleep(n):

```
signal(14,null);
alarm(n);
pause();
alarm(0);
```

In fact, something more is required, partly because of timing problems and partly because of interaction with other signals. What is needed will be explained after the next example.

Another system call which it is sometimes desirable to interrupt is the opening of a teletype. An attempt to open a teletype which is not dialed in will lead to the suspension of a process until carrier is reestablished on the line. Most programs, of course, will not attempt to open any teletype explicitly. They will use one of the standard file descriptors 0,1,2 for terminal I/O. Some programs, however, may attempt to obtain direct access to the controling teletype by opening /dev/tty. If a program is being run in background under nohup, /dev/tty may no longer be dialed in. To prevent indefinite suspension at the open call, the following code should suffice:

```
signal(14,null);
alarm(10);
tty = open("/dev/tty",2);
alarm(0);
```

If the terminal has been hung up, tty == -1 should result.

A first modification to the above schemes is required to handle the possibility that the alarm signal may be posted before the process has reached the point of suspension. The alarm clock ticks off real time. In a time-sharing environment it cannot be predicted how long it will take a program to execute even short sections of code. Consequently, a more elaborate scheme is needed:

```
int set14 0;

sig14()
 {signal(14,sig14);
  if (set14) alarm(1);};

main()
 {...
  sig14();
  ...};

sleep(n)
 {alarm(set14=n);
  pause();
  alarm(set14=0);};
```

Sig14() should be called early in the program. It will maintain the trap on SIGCLK thereafter.

Experience has proved the above modification alone to be inadequate. RJE background processes, coded as above, have occasionally been detected suspended in pause() indefinitely. It is suspected that ignored signals issued from the controlling terminal of the otherwise detached process have been masking the alarm signal. An examination of UNIX shows this to be possible. The C program attached to these notes provides a practical example of the masking of signals. A fix for the problem can be devised which should guarantee infallible operation of the alarm mechanism. Unfortunately, the fix has undesirable side-effects. The fix will not be installed in background RJE code. It may never be known, therefore, whether the second modification proposed below is indeed adequate.

The problem alluded to above derives from the fact that UNIX remembers only the most recent signal directed to a process. Signals are not posted to processes immediately. A process looks for pending signals requiring action only at certain points during the processing of a system call and when strobed by a clock interrupt. One pending signal may be overwritten by another before the process detects it. A signal to be caught, such as SIGCLK, may be overwritten by another which is to be ignored, such as SIGINT. In such an event, the process will not be notified that the alarm clock has gone off.

The moral of this is that you cannot ignore any expected signals if you wish the alarm mechanism to work properly. Asynchronous signals generated from a terminal, such as SIGHUP, SIGINT and SIGQIT, are especially dangerous. A user may spawn a nohup background process and then proceed on to other work. Each time the user hits the DEL key thereafter, the user is potentially interfering with the background alarm clock. A program which is sometimes run in background may protect itself as follows:

```
int set14 0;

xalarm(n)
 {return (alarm(set14=n));};

sig14()
 {signal(14,sig14);
  if (set14) alarm(1);};

sigx(n)
 {printf("\nSignal #%d.\n",n);
  exit(~0);};

sigz(n)
 {register int t;
  signal(n,sigz);
  if (set14)
   {t=alarm(1);
    if (t) alarm(t);};};
```

```
main()
 {...
  for (t=1;t<4;t++)
   {if (!signal(t,sigz)) signal(t,sigx);};
  for (t=4;t<20;t++) signal(t,sigx);
  sig14();
  ...};

sleep(n)
 {xalarm(n);
  pause();
  xalarm(0);};

  ...
  xalarm(10);
  tty=open("/dev/tty",2);
  xalarm(0);
  if (tty<0) {/*error*/ ...
```

One example cannot indicate all possible treatments of signals. In the above, all signals which are not to be ignored lead to a diagnostic message followed by exit. The (pseudo-)ignored signals are trapped through sigz() to keep the alarm clock running.

Some methods for handling signals do not affect the alarm mechanism. Signals for which the default action is desired need not be trapped. One must simply insure that a parent process has not set them to be ignored. Signals which will invoke a reset or something similar probably do not need to inspect the alarm setting. It is primarily dangerous to ignore the three teletype signals. It is also dangerous to trap the teletype signals, in order to set a flag and return, without checking the alarm setting.

The undesirable side-effect of the last method proposed for insuring the integrity of the alarm mechanism is that it makes a process vulnerable to rapidly repeated signals. UNIX resets the trap for most caught signals to default action at the time they are posted to a process. Before the process can reestablish the associated trap, a second instance of the same signal may occur. This will invoke the default action, which will probably cause termination of the process. Such events are rare because they depend on the occurrence of a clock interrupt within a critical interval. They may be effectively non-repeatable. Nonetheless, such events can occur. Background processes can only protect themselves by ignoring signals entirely. Such protection is incompatible with guaranteed operation of the alarm mechanism.

The ability to remember multiple pending signals appears to be a most desirable enhancement to the UNIX operating system. It could be accomplished without too much trouble. Instead of storing the number of the most recent signal, UNIX could update a bit array of signals generated but not yet posted. This would remove the undesirable masking of one signal by another. It would allow the alarm signal to be treated independently of the others. In the presence of such an enhancement, the only special consideration in dealing with the alarm signal would be to recycle it if it went off too soon.

Protection against rapidly repeated trapped signals is a second desirable enhancement to UNIX. However, this problem has been recognized for a long time, and people have either adapted to it or become reconciled to it. It is not clear is anything can be done about it now.

Finally, the implementation of sleep() in UNIX might be made more efficient, perhaps by implementing a second internal alarm clock transparent to user processes. This would relieve any pressure to convert to the alarm/pause mechanism for the sake of efficiency. In the presence of such an enhancement, most programmers would not need to concern themselves with the two new system calls.

Terry G. Lyons

PY-9144-TGL

T. G. Lyons

Att.
alarm_bug.c

Copy (with att.) to
9144 Supervision
A. P. Boysen, Jr.
J. F. Maranzano
D. M. Ritchie

```
/* alarm bug - tty signal masks out SIGCLK */

main()
 {sig14();
  if (fork())
   {for (;;)
     {write(1,".",1); sleep(1);};};
  signal(2,0);
  for (;;)
   {kill(0,2); sleep(1);};};

int set14;

sig14()
 {signal(14,sig14);
  if (set14) alarm(1);};

sleep(n)
 {int i;
  alarm(set14=n);
  for (i=0;i<30000;i++);    /*simulated swap-out*/
  pause();
  alarm(set14=0);};

/*end*/

/* usage:  a.out &
   when `.' stops typing, hit break.
   `ps l' will show process wchan = 140000 */
```