Keeping Time--Interval Timers in Amiga UNIX

By David Miller

The Sleep() function

When you want your program to pause for a number of seconds then continue, you will typically use the sleep(3) function (the notation NAME(SECTION#) refers to the manual page NAME in the SECTION# chapter of the *UNIX Reference Manuals*; section 1 is commands, section 2 is system calls, and section 3 is library functions). There is also a sleep(1) program which provides the same function to shell scripts. For example:

In C:

```
main()
{
    printf("Hello world!\n");
    sleep(5);
}
```

Shell:

```
#!/usr/bin/sh

echo "Hello world!"
sleep 5
```

The following example is an implementation of the sleep(3) function.

Example 1: SLEEP() Using ALARM()

```
 1  # include       <signal.h>
 2  static void trap ()
 3  {
 4  }
 5  unsigned sleep ( unsigned duration )
 6  {
 7          void            (*oldsig)();
 8          unsigned        oldtime;
 9          oldsig = signal(SIGALRM, trap);
10          oldtime = alarm(0);
11          if (oldtime && oldtime < duration)
12                  alarm(oldtime);
13          else
14                  alarm(duration);
15          pause();
16          signal(SIGALRM, oldsig);
17          if (oldtime > duration)
18                  alarm(oldtime - duration);
19          if (oldtime && oldtime < duration)
20                  return(duration - oldtime);
21          else
22                  return 0;
23  }
```

Line	Explanation
1	The file signal.h defines the parameters for the signal(2) function.
2-4	trap() is the simple signal handler that just sets a flag that another piece of code will examine.
5	Define the sleep() function.
6-8	oldsig is a variable that will be used to save the previous state of the signal handler; oldtime will be used to save the state of the alarm clock.
9	Establish trap() as the current signal handler for the alarm signal. The previous handler is saved in oldsig.
10	Clear the alarm clock, saving the current state.
11-14	If the previous setting of alarm was sooner than duration, use the old value, otherwise use duration to set the alarm.
15	Pause the process until the alarm goes off.
16	Restore the old signal handler.

17-18 If the old alarm setting was later than duration, reset the alarm with the difference between duration and oldtime (the time remaining until the previous alarm).

19-22 If an existing alarm is making sleep return early, return the time remaining on the requested sleep.

To schedule its ‘‘wake up’’ time, sleep(3) uses the alarm(2) system call. The alarm(2) system call asks the OS to deliver a signal (basically a software interrupt) in some number of seconds according to the system clock. However, setting an alarm for 2 seconds does not mean that you will receive an alarm signal in exactly two seconds.

The OS processes alarm requests once every second. Each time an alarm request is processed, the number of seconds remaining for that alarm is decremented by one. When the number of seconds remaining reaches zero, the OS delivers a signal to the process. Given this, if a process places a 1 second alarm request 1 microsecond before the OS does its alarm processing, the signal will arrive in 1 microsecond, not in 1 second. An alarm set for **N** seconds actually means:

> deliver a signal after **N-**1 seconds, but before **N** seconds.

If you’d like to see for yourself, try running this shell script:

```
$ for i in 1 2 3 4 5 6 7 8 9 10
> do
>  time sleep 2
> done
```

How many times did the process actually take more than 2 seconds? If you try it with both the Korn shell, ksh(1), and the System V shell, sh(1), you’ll find that under ksh(1) it takes about 1.2 seconds and under sh(1) it takes about 1.8 seconds (There is a difference because sh(1) counts the time required to start the sleep process whereas ksh(1) only counts the actual running time). The sleep(1) program rarely, if ever, actually sleeps for 2 seconds.

If you are writing a daemon that checks for some event every 5 minutes, or if you want to pause the output to give the user a chance to read it, alarm’s 1 second granularity is fine. But what about that daemon that needs to wake up every second? Waking up after 1 microsecond could cause the process to run almost continuously. For any sort of realtime processing, one second is a very long time. So how do you sleep for less than one second reliably?

The answer is do not use an alarm, use an *interval timer*.

Interval Timers

Each interval timer has a resolution of 1 tick of the system's clock, or 1 microsecond (whichever is larger). Additionally, you can configure an interval timer to automatically restart itself. The system provides each process with three independent interval timers:

`ITIMER_REAL` This timer will count down in real time. That is, this timer will continue to run when your process is waiting for the OS to perform a system call, or when the OS preempts your process. When the timer expires, the OS will deliver a `SIGALARM` signal.

`ITIMER_VIRTUAL` This timer counts down only when your process is running. If your process makes a system call, or is preempted, this timer will stop counting. The timer will resume when your process resumes execution. When this timer reaches zero, the process will receive a `SIGVTALRM` signal.

Possible uses for this timer include checkpointing (saving data after some period of execution) and multithreading. The virtual timer is more desireable for these applications since it counts only when the process is running; there is no reason to perform a checkpoint or switch threads if the process has been idle.

`ITIMER_PROF` This timer will stop counting any time your process is preempted by the OS, but will not stop when the process is waiting for a system call to return. When it expires the OS generates a `SIGPROF` signal.

This timer is designed to be used for execution profiling by interpreters. By having a profiling timer send a signal every second, or fraction of a second, and examining the current position in the interpreted code, the process can determine where the most execution time is being spent.

All three timers operate on the following structure:

```
struct itimerval
{
        struct timeval it_interval;
        struct timeval it_value;
}
```

The timeval structure looks like this:

```
struct timeval
{
      long tv_sec;
      long tv_usec;
}
```

Both of these structures are defined in the *<sys/time.h>* include file.

Note that System V Release 4.0 does not guarantee that these are the only members of these structures, nor that they will occur in this order. You must initialize the members individually. This can be annoying and tedious, but it allows the structure to be expanded in future releases.

You set and examine timers using these two functions:

```
int getitimer(int which, struct itimerval *myvalue)
```

places the current timer setting into myvalue

```
int setitimer(int which, struct itimerval *myvalue, struct itimerval *myovalue);
```

sets the timer. It gets the new time from myvalue and places the previous setting in myovalue.

Now, let us take a look at the sleep function again. The code in example 2 creates a new version of sleep that will sleep an exact number of seconds.

Example 2: SLEEP() Using an Interval Timer

```
# include       <signal.h>
# include       <sys/time.h>
static void trap ()
{
}
unsigned mysleep ( unsigned duration )
{
       void                    (*oldsig)();
       struct itimerval        oldtime;
       struct itimerval        newtime;
       oldsig = signal(SIGALRM, trap);
       getitimer(ITIMER_REAL, &oldtime);
       if (oldtime.it_value.tv_sec == 0
                && oldtime.it_value.tv_usec == 0
                || oldtime.it_value.tv_sec >= duration)
       {
               newtime.it_interval.tv_sec = 0;
               newtime.it_interval.tv_usec = 0;
               newtime.it_value.tv_sec = duration;
               newtime.it_value.tv_usec = 0;
               setitimer(ITIMER_REAL, &newtime, NULL);
       }
       pause();
       signal(SIGALRM, oldsig);
       if (oldtime.it_value.tv_sec > duration)
       {
               oldtime.it_value.tv_sec -= duration;
               setitimer(ITIMER_REAL, &newtime, NULL);
       }
       if (oldtime && oldtime < n)
               return(n - oldtime);
       else
               return 0;
}
```

Line	Explanation
1-2	The file *signal.h* defines the parameters of the signal(2) function. The file <*sys/time.h*> defines `ITIMER_REAL` and the structures used by getitimer(3) and setitimer(3).
3-5	trap() is the simple signal handler that just sets a flag that another piece of code will examine.
6-7	Define the mysleep() function.
8-10	oldsig will hold the previous state of the signal handler; oldtime will hold the state of the timer; and newtime will be used to set the new timer parameters.
11	Establish trap() as the current signal handler for the alarm signal. The previous handler is saved in oldsig.
12	Fetch the current settings of the timer.
13-20	If the timer was idle (both parts of it_value are zero) or it is set to go off later than duration (it_value.tv_sec is greater than duration), set the timer to go off in duration seconds.
21	pause(2) the process until the timer expires.
22	Restore the old signal handler.
23-27	If the old timer setting was later than duration, reset the timer with the difference between duration and oldtime.it_value.tv_sec (the time remaining until expiration of the previous setting).
28-32	If an existing timer is making mysleep() return early, return the time remaining on the requested mysleep().

This example uses what is called a ‘‘one-shot’’ timer. The timer goes off once, and then stops. By supplying an interval setting, the timer becomes a clock, generating alarm signals on a regular basis.

The code fragment in Example 3 shows how to set the timer to produce a 1.5 second clock that will start ‘‘ticking’’ in 1 minute.

Example 3: Using an Interval Timer as a Clock

```
1  # include       <sys/time.h>
2  ...
3          {
4                  struct itimerval        newtime;
5                  newtime.it_value.tv_sec = 60;
6                  newtime.it_value.tv_usec = 0;
7                  newtime.it_interval.tv_sec = 5;
8                  newtime.it_interval.tv_usec = 500000;
9                  setitimer(ITIMER_REAL, &newtime, 0);
10         }
11 ...
```

Line	Explanation
1	The file *<sys/time.h>* defines **`ITIMER_REAL`** and the structures used by getitimer(3) and setitimer(3).
2	Other code.
3	Start of block.
4	Structure newtime will hold the setting for the timer.
5-6	Set it_value to expire in 1 minute.
7-8	Set it_interval to reload it_value with 5 seconds and 500,000. microseconds (one half of a second).
9	Load the new settings into the **`ITIMER_REAL`** timer.
10	End of block.
11	Other code.

When the time interval specified by it_value expires, the contents of it_interval is copied into it_value and the timer is restarted. If the interval specified by it_interval is zero, the timer stops. Timers can be stopped at anytime by calling setitimer(3) with the members of it_value set to zero.

So there you have the realtime interval timer. The other timers work exactly the same way, varying only in when the timer is running. For more information on interval timers see *UNIX System V Release 4 - Programmer's Reference Manual*, published by Prentice Hall. Next time, we'll explore context switching, multithreading, and light weight processes.