High-Resolution Timers —Part II

This article focuses on software timers and their role in real time applications.

A timer is similar to a clock. We can make use of timers to control any event. There are different types -- mechanical timers, digital timers and computer timers. In the category of computer timers, there are hardware and software timers. The former are operated according to every clock jiffy, whereas, software timers are operated according to few internal routines that may not be called at every clock jiffy. A clock jiffy is the measure of one clock cycle. The high-resolution timers discussed here belong to the category of software timers.

For real-time applications, timers are very much essential. Application developers have to understand the timers that are supported by the operating system in order to develop applications efficiently. Linux supports three interval timers, namely *ITIMER_REAL*, *ITIMER_VIRTUAL* and *ITIMER_PROFILE*. However, these timers work in units of microseconds, which may not be suitable for most real-time applications. As we know that Linux supports soft real-time applications, it has to support timers for real-time applications also. Hence, Linux supports high-resolution timers for real-time applications according to the POSIX standard.

High-resolution timer APIs are of two categories—timer management APIs (namely, *timer_create*, *timer_delete*, *timer_settime*, *timer_gettime* and *timer_getoverrun*) and clock management APIs (namely, *clock_getres*, *clock_settime*, *clock_gettime*). A few basic differences between these two categories are:

1. Using timer functions, we can set the time in units of seconds and nanoseconds and we can also make it to work as a one-shot or periodic timer. Time can be counted according to *CLOCK_*

REALTIME or *CLOCK_MONOTONIC.*

2. The *clock_settime* function can also be used to set *CLOCK_REALTIME* or *CLOCK_MONOTONIC*, but in normal conditions, it may not be needed. It is most suitable in the application that monitors time over the network and to synchronise with other systems.
3. Using the *timer_settime* function, we can even do the signal settings, which is not possible with clock functions.

Timer management APIs, except for *timer_getoverrun,* have already been discussed in the article titled 'High Resolution Timers—A prime feature of real-time applications' published in July 2007 edition of *LFY*. So in this article, we are trying to explore and explain other system calls: *clock_settime, clock_gettime, clock_getres* and *timer_getoverrun.*

1. timer_getoverrun( ): The *timer_getoverrun* system call helps us to know how many times a timer was expired when the process is busy. To illustrate the usage of this system call with an example, a few signal-related system calls are used. The interested reader can get more information about Linux signals at *www.linuxjournal.com/article.php?sid=6483*. The *Timer_getoverrun* system call accepts only one argument and the syntax is as follows:

```
int   timer_getoverrun (timer_t timer_id);
```

The application of this API is explained with the following example program:

```
#include<stdio.h>
#include<time.h>
#include<signal.h>
void sigfn() {
        printf("Signal is caught \n");   }
int main()  {
        long int ret;
        long long int i;
        timer_t id;
        struct sigevent s;
        struct itimerspec it;
        struct sigaction act;
        sigset_t pro_set;
        s.sigev_signo=SIGUSR1;
        s.sigev_notify=SIGEV_SIGNAL;
        act.sa_handler=sigfn;
        act.sa_flags=0;
        ret=sigaction(SIGUSR1,&act,0);
        it.it_value.tv_sec=1;
        it.it_value.tv_nsec=0;
        it.it_interval.tv_sec=1;
        it.it_interval.tv_nsec=0;
        ret=timer_create(CLOCK_REALTIME,&s,&id);
        ret=timer_settime(id,0,&it,NULL);
        ret=sigemptyset(&pro_set);
        ret=sigaddset(&pro_set,SIGUSR1);
        ret=sigprocmask(SIG_SETMASK,&pro_set,NULL);
        sleep(15);
        ret=sigprocmask(SIG_UNBLOCK,&pro_set,NULL);
        ret=0;
        ret=timer_getoverrun(id);
        printf("overruns=%ld\n",ret);   return 0;   }
```

The SIGUSR1 signal is used to notify the process when the timer expires and the same has been registered using the sigaction system call. The timer has been created with *CLOCK_REALTIME,* and has been set to 1 sec as the *it_value* and 1 sec as *it_interval.*

sigemptyset API is used to initialise the *sigset_t pro_set.* Generally, before using *pro_set* in other APIs, it will be initialised using the sigemptyset or sigfillset API. This is necessary so that *pro_set* does not hold any other signal related information.

sigaddset API is used to add the SIGUSR1 signal to the set pointed by *pro_set*. To make the *pro_set* point to the current signal mask for the process, we are using the sigprocmask API with *SIG_SETMASK* as the first argument; the address of *pro_set* as the second argument and the third argument as *NULL*. The third argument can be used to know the old value of the signal mask, which is of type *sigset_t*.

The application has to set the signal and the timer before starting the specific task, so that the process will be busy. Then, sigprocmask is called again by passing *SIG_UNBLOCK* as the first argument. This is to know the number of times the signal was generated when the process is busy. In the end, if we use the *timer_getoverrun* system call by passing the *timer_id* as the only argument, it will return the number of times the timer had expired when the process was busy. To compile the applications that use high resolution timers, the *lrt* option is required with the cc or gcc compiler.

Now, we will look into clock management system calls like *clock_getres*, *clock_gettime* and *clock_settime*. The *clock_getres* API is used to know the resolution of the *CLOCK_REALTIME*, *CLOCK_MONOTONIC*, *CLOCK_PROCESS_CPUTIME_ID*, *CLOCK_THREAD_CPUTIME_ID* and other clocks, if supported by the system. The mentioned clock related functions return 0 for a successful execution and -1 for failure.

Different clock types are briefly described here; *CLOCK_REALTIME* can be used to set the timer to count according to the wall clock time and this clock type is visible to all the processes in the system. *CLOCK_REALTIME* measures the time since Epoch, i.e., the1st January 1970 0 hours according to UTC time zone. The second type of clock is *CLOCK_MONOTONIC*, where the timer has to count with reference to some starting point, which is system dependent. The third type of clock is *CLOCK_PROCESS_CPUTIME_ID*, which means that the timer will be incrementing/decrementing only when the calling process is executing in the CPU (*TASK_RUNNING* state of the process). This clock is also system dependent. The fourth type of clock is *CLOCK_THREAD_CPUTIME_ID*. Here, the timer will perform increment/decrement operations only when the calling thread is in the running state and it is also system dependent.

2. clock_getres( ): In the example of Program 2, we use only *CLOCK_REALTIME,* which is a system-wide realtime clock. The *clock_getres* API accepts two arguments, the first of which is the clock ID and the second argument is a structure variable of type timespec. The syntax of *clock_getres* is as follows:

```
int   clock_getres (clockid_t clockid, struct timespec *res);
```

The following program illustrates the use of the *clock_getres* API:

```
#include<stdio.h>
#include<time.h>
int main()
{
         int ret;
         struct timespec t;
         ret=clock_getres(CLOCK_REALTIME,&t);
         if(ret==-1)
                   perror("clock_getres error");
         printf("Resolution of realtime clock=%d secs %ld nsecs\
n",t.tv_sec,t.tv_nsec);
         return 0;
}
```

By passing a variable of type struct timespec to the clock_getres API, we can specify (or declare) the resolution of the clock in terms of seconds and nanoseconds.

3. clock_gettime( ): The *clock_gettime* API can be used to figure out how much time a particular clock has spent since the Epoch. The Epoch can be briefly explained as a reference date that is considered for time measurements. In Linux, the Epoch date is 1st January 1970, according to the UTC timezone. *clock_gettime* takes two arguments as the input. First, on which clock do we want to know the time (for example, *CLOCK_REALTIME)*, and the second argument is a struct timespec variable. By displaying the members of the timespec structure, we can know the time spent in terms of seconds and nanoseconds. The *clock_gettime* API syntax is as follows:

```
int   clock_gettime (clockid_t cid, struct timespec *res);
```

The following program shows the use of the *clock_gettime* API.

```
#include<stdio.h>
#include<time.h>
int main()
{
         long int ret,i,j;
         struct timespec t,ot,t1;
         clockid_t cid;
         ret=clock_gettime(CLOCK_REALTIME,&ot);
         printf("%d secs \t %ld nsecs\n",(ot.tv_sec),(ot.
tv_nsec));
         sleep(5);
         ret=clock_gettime(CLOCK_REALTIME,&ot);
         printf("%d secs \t %ld nsecs after the process is busy\
n",(ot.tv_sec),(ot.tv_nsec));
         return 0;
}
```

4. clock_settime( ): The *clock_settime* API is used to set a particular clock with some seconds and nanoseconds. Again, the time that we are setting will be referred to according to the Epoch. So, like the *timer_settime* API, the process will not receive any signal when the set time expires. By only using the *clock_gettime* API before and after setting the clock, we can measure the difference or the amount of time consumed by the process. The *clock_settime* API takes two arguments and the syntax is shown below:

```
         int    clock_settime  (clockid_t cid, const struct
timespec *res);
```

The following program illustrates the use of the *clock_settime* with the *clock_gettime* API.

```
#include<stdio.h>
#include<time.h>
#include<signal.h>
int main()
{
         long int ret,i,j;
         struct timespec t,ot,t1;
         clockid_t cid=CLOCK_REALTIME;
         t.tv_sec=1;
         t.tv_nsec=1;
         ret=clock_gettime(cid,&ot);
         printf("%d secs \t %ld nsecs before setting the timer\
n",(ot.tv_sec),(ot.tv_nsec));
         ret=clock_settime(CLOCK_REALTIME,&t);
         for(i=0;i<1000000000;i++)
                   getpid();
         sleep(2);
         ret=clock_gettime(CLOCK_REALTIME,&ot);
         printf("gettime sec=%d\n",(ot.tv_sec));
         printf("gettime nsec=%d\n",(ot.tv_nsec));
         return 0;
}
```

Timers and clock functions are best suited for real time applications since they have nanosecond resolutions. Here, the example programs are assumed to be real time processes. END

By: Parimala S. and Dr B. Thangaraju. *The authors work with Talent Transformation, Wipro Technologies, Bangalore. They can be reached at parimala.sathyapramodha@wipro.com and balat.raju@wipro.com, respectively.*