Over and over again: periodic tasks in Linux

Fri, 09/25/2009 - 17:03 — csimmonds

It is very common for real-time applications to have tasks that need to run periodically, for example to scan inputs or to generate regular outputs. A crude solution is to use a processing loop with a sleep at the end, but the periodicity will vary as the execution time varies. To create accurate periodic tasks you need to use timers. In this article I will show how timers work in Linux, especially with regard to multi-threaded applications.

Linux has several different timer interfaces, acquired over many years. Which to use depends on the versions of the kernel and C library you have. If you are using GNU libc 2.8 and kernel 2.6.25 or later, the timerfd interface is the best. If you are using GNU libc 2.3 and any version of the 2.6 kernel, the POSIX timers interface works well. If you are using uClibc you should use setitimer. I have examples of all three below.

In the examples I have separated out the timer code into two functions, *make_periodic* and *wait_period*:

```
struct periodic_info
{
        /* Opaque data */
};

int make_periodic (unsigned int period, struct periodic_info *info);
void wait_period (struct periodic_info *info);
```

You call *make_periodic* at the start of the thread giving the period in microseconds and then call *wait_period* when execution is complete. This is inspired by RTAI [1] which has functions rt_task_make_periodic and rt_task_wait_period to do the same thing. To show you what I mean, here is an example of a thread with a period of 10 ms:

```
void *thread_1 (void *arg)
{
        struct periodic_info info;

        make_periodic (10000, &info);
        while (1)
        {
                /* Do useful work */
                wait_period (&info);
        }
        return NULL;
}
```

Using timerfd

The timerfd interface is a Linux-specific set of functions that present POSIX timers as file descriptors (hence the fd) rather than signals thus avoiding all that tedious messing about with signal handlers. It was first implemented in GNU libc 2.8 and kernel 2.6.25: if you have them I highly recommend this approach.

You create a timer by calling timerfd_create() giving the POSIX clock id CLOCK_REALTIME or CLOCK_MONOTONIC. For periodic timers

such as we are creating it does not matter which you choose. For absolute timers the expiry time is changed if the system clock is changed and the clock is CLOCK_REALTIME. In almost all cases, CLOCK_MONOTONIC is the one to use. timerfd_create returns a file descriptor for the timer.

To set the timer running, call timerfd_settime() giving flag = TFD_TIMER_ABSTIME for an absolute timer or 0 for relative, as we want here, and the period in seconds and nanoseconds. To wait for the timer to expire, read from its file descriptor. It always returns an unsigned long long (8 byte unsigned integer) representing the number of timer events since the last read, which should be one if all is going well. If it is more than one then some events have been missed. In my example below I keep a record in "wakeups_missed".

```
#include

struct periodic_info
{
        int timer_fd;
        unsigned long long wakeups_missed;
};

static int make_periodic (unsigned int period, struct periodic_info *info)
{
        int ret;
        unsigned int ns;
        unsigned int sec;
        int fd;
        struct itimerspec itval;

        /* Create the timer */
        fd = timerfd_create (CLOCK_MONOTONIC, 0);
        info->wakeups_missed = 0;
        info->timer_fd = fd;
        if (fd == -1)
                return fd;

        /* Make the timer periodic */
        sec = period/1000000;
        ns = (period - (sec * 1000000)) * 1000;
        itval.it_interval.tv_sec = sec;
        itval.it_interval.tv_nsec = ns;
        itval.it_value.tv_sec = sec;
        itval.it_value.tv_nsec = ns;
        ret = timerfd_settime (fd, 0, &itval, NULL);
        return ret;
}

static void wait_period (struct periodic_info *info)
{
        unsigned long long missed;
        int ret;

        /* Wait for the next timer event. If we have missed any the
           number is written to "missed" */
        ret = read (info->timer_fd, &missed, sizeof (missed));
        if (ret == -1)
        {
                perror ("read timer");
                return;
        }
```

```
        /* "missed" should always be >= 1, but just to be sure, check it is not 0 anyway */
        if (missed > 0)
                info->wakeups_missed += (missed - 1);
}
```

Using POSIX timers

If your glibc or kernel doesn't support timerfd, then you will have to use POSIX timers to generate signals and wait for the signal to arrive to indicate the start of the next period. This causes problems because signals are sent to the process not the thread. If you have several periodic threads, and therefore several timers, in a process each one must use a different signal to tell them apart. The obvious signals to use are the real time signals from SIGRTMIN (33) to SIGRTMAX (64), so you cannot have more than 32 timers per process. Note *per process*: it is perfectly acceptable to have 32 other timers in another process.

The way to wait for a signal to arrive is to block it and then call sigwait(). Here is another complication: although signals are sent to the parent process, each thread has its own signal mask. I will write another article on the reasons it is done this way, but in this case it has the implication that all the real time signals must be blocked before creating any threads so that they all inherit the same mask. Doing it any other way risks the race condition where the signal is delivered before all threads have blocked it, resulting in the process being killed.

You can detect missed timer events using the function timer_getoverrun (), which returns zero if none were missed. Here is the code:

```
struct periodic_info
{
        int sig;
        sigset_t alarm_sig;
        int wakeups_missed;
};

static int make_periodic (int unsigned period, struct periodic_info *info)
{
        static int next_sig;
        int ret;
        unsigned int ns;
        unsigned int sec;
        struct sigevent sigev;
        timer_t timer_id;
        struct itimerspec itval;

        /* Initialise next_sig first time through. We can't use static
           initialisation because SIGRTMIN is a function call, not a constant */
        if (next_sig == 0)
                next_sig = SIGRTMIN;
        /* Check that we have not run out of signals */
        if (next_sig > SIGRTMAX)
                return -1;
        info->sig = next_sig;
        next_sig++;

        info->wakeups_missed = 0;

        /* Create the signal mask that will be used in wait_period */
        sigemptyset (&(info->alarm_sig));
        sigaddset (&(info->alarm_sig), info->sig);

        /* Create a timer that will generate the signal we have chosen */
```

```
        sigev.sigev_notify = SIGEV_SIGNAL;
        sigev.sigev_signo = info->sig;
        sigev.sigev_value.sival_ptr = (void *) &timer_id;
        ret = timer_create (CLOCK_MONOTONIC, &sigev, &timer_id);
        if (ret == -1)
                return ret;

        /* Make the timer periodic */
        sec = period/1000000;
        ns = (period - (sec * 1000000)) * 1000;
        itval.it_interval.tv_sec = sec;
        itval.it_interval.tv_nsec = ns;
        itval.it_value.tv_sec = sec;
        itval.it_value.tv_nsec = ns;
        ret = timer_settime (timer_id, 0, &itval, NULL);
        return ret;
}

static void wait_period (struct periodic_info *info)
{
        int sig;
        sigwait (&(info->alarm_sig), &sig);
        info->wakeups_missed += timer_getoverrun (info->timer_id);
}

int main(int argc, char *argv[])
{
        sigset_t alarm_sig;
        int i;

        /* Block all real time signals so they can be used for the timers.
           Note: this has to be done in main() before any threads are created
           so they all inherit the same mask. Doing it later is subject to
           race conditions */
        sigemptyset (&alarm_sig);
        for (i = SIGRTMIN; i <= SIGRTMAX; i++)
                sigaddset (&alarm_sig, i);
        sigprocmask (SIG_BLOCK, &alarm_sig, NULL);
```

Using setitimer

This ONLY works if you are using uClibc. Actually the real determinant is that you are using the Linux Threads library rather than the Native POSIX Threads Library. With very few exceptions, uClibc uses Linux Threads, glibc uses NPTL.

Using setitimer is somewhat similar to POSIX clocks except that it is hard coded to deliver a SIGALRM at the end of each period. Using NPTL that means that you can only have one periodic task per process, but with Linux Threads each thread IS a process so that is fine: you can have as many periodic threads as you like.

Setitimer is part of the base POSIX specification and has been present in Linux since the year dot. The time out is passed in a struct itimerval which contains an initial time out, it_value, and a periodic time out in it_interval which is reloaded into it_value every time it expires. At each expiry it sends a SIGALRM. The times are given in microseconds which will be rounded up the the granularity of your timers if they are greater than 1 us. The best way to handle the signal is to block it and then wait for the next one with sigwait() as shown below. There is no easy way to detect missed timer events.

Here is the code:

```
struct periodic_info
{
        sigset_t alarm_sig;
};

static int make_periodic (unsigned int period, struct periodic_info *info)
{
        int ret;
        struct itimerval value;

        /* Block SIGALRM in this thread */
        sigemptyset (&(info->alarm_sig));
        sigaddset (&(info->alarm_sig), SIGALRM);
        pthread_sigmask (SIG_BLOCK, &(info->alarm_sig), NULL);

        /* Set the timer to go off after the first period and then
           repetitively */
        value.it_value.tv_sec = period/1000000;
        value.it_value.tv_usec = period%1000000;
        value.it_interval.tv_sec = period/1000000;
        value.it_interval.tv_usec = period%1000000;
        ret = setitimer (ITIMER_REAL, &value, NULL);
        if (ret != 0)
                perror ("Failed to set timer");
        return ret;
}

static void wait_period (struct periodic_info *info)
{
        int sig;

        /* Wait for the next SIGALRM */
        sigwait (&(info->alarm_sig), &sig);
}
```

Conclusion

The accuracy of the timers will depend on the your kernel and scheduling policy and priority you use for the threads. By default all time-outs will be rounded up to the nearest 10 ms (actually 1/HZ, but in most cases HZ = 100). If your board support package supports High Resolution Timers (most do) enabling CONFIG_HIGH_RES_TIMERS will give you accuracy to a few microseconds. Periodic threads are almost by definition real-time, so you probably want to give them a real-time policy such as SCHED_FIFO (in a follow-up article I will look into the implications of real-time periodic threads). Finally, if you want to reduce jitter to sub millisecond, you should enable kernel pre-emption (CONFIG_PREEMPT) or for jitter in the 10's to 100's microsecond region you should apply the PREEMPT_RT patch [2].

You can download demonstrations of all three techniques from here.

References

[1] RTAI: the RealTime Application Interface, https://www.rtai.org/

[2] The PREEMPT_RT real time patch series, http://www.kernel.org/pub/linux/kernel/projects/rt/

Pthreads

Comments

Comment viewing options

[redacted] [redacted] [redacted] Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

dream hosting

Wed, 11/28/2012 - 21:08 — TrergienI (not verified)

this website

Finding Your Path Round The Web Hosting Entire world

When designing a site, you'll ought to choose a web hosting company sooner or later in time. Although you may don't know very much about website hosting, there are many quick questions you may request to make sure you get what you require at a cost you really can afford. Keep reading for what you must watch out for throughout your hunt.

The down time of your web host ought to be carefully scrutinized. The time period accustomed to conduct hosting company servicing, along with the time the maintenance is carried out, must be thought about. If they seem to be off-line during optimum hrs or regularly through the entire month, you will need a better number.

Support all of your current information and facts, don't depend on any web host to accomplish this. It's up to you to ensure you back your blog often. This really is the only method to make certain a challenge doesn't damage your data. If your internet site is Search engine optimisation intense, it can be specially significant not to take a chance on losing everything that operate.

Be sure that there are actually no costs for cancellation. You could decide to end your assistance after few weeks. When you visit stop, you might find out your company has a big cancellation payment. It is a standard exercise, specifically for web hosting solutions which can be affordable. Ensure you understand what the results will likely be of finishing a contract early on.

Low-cost web hosts will not be the most effective solution. When you'll certainly be tempted by their the best prices, you must also know that they generally convert to poor quality providers. They either possess a unsafe business model, or these are cutting edges in a fashion that will end up impacting you and your website.

Read about your potential hold to see what kind of internet sites they handle. Plenty of totally free internet sites offer you only stationary webpages, which means you can't put vocabulary scripts of your personal. If you discover your self requiring a dynamic scripting site, you might need to find an cost-effective pay host rather.

Do not let the amount of alternatives available with web hosting companies to overwhelm you. More than the recent years, countless new website hosting providers have came into the marketplace several present rock-base prices. Keep in mind that in internet hosting, as with most things, you will get the things you pay for. You may thin your options straight down by trying to find your leading tastes inside a web host, and comparing prices and offerings properly.

An effective web host is communicative. You want a host that communicates having its clients and provide them information on any updates or down time and upkeep. It is additionally significant to possess a hosting provider that can response your questions must any troubles come up.

Discover more about the backdrop of any web hosting support just before investing in a plan. Some companies make outlandish claims or promises that cannot be substantiated. Doing your research is the best way to make the right decisions.

See how big your site is going to be inside the next year and select a internet hosting plan which gives you adequate hard disk drive area. An Web-page coding site takes minimal area, but incorporating pictures or videos will require a lot more space. For internet hosting data files by itself, all around 100MB - 1 GB ought to give a great starting system for your domain.

If you love services or design and style program from one business, you don't must feel you have to utilize the web hosting services they supply also. Most hosts that you deal with will allow you to use various professional services, even if you will likely possess a more difficult time addressing problems by doing this.

Just because an online number company is free, does not always mean you should guideline it all out entirely. You may be wary of cost-free online hosts due to the fact many of them spot banner adverts on top of your websites, producing your site look unprofessional. Even so, some cost-free hosts don't use banner ad adverts, so it's well worth looking into cost-free hosting services that appeal to you

as an alternative to judgment them out instantly. Understand that utilizing a totally free hold will save you lots of money every single 12 months.

Ensure that your website address is listed by you rather than your host company to help you ensure that it stays should you really transform providers. This spots the charge of your site in your hands, as an alternative to your host's.

Deciding on a web host which can be physically found in the exact same country when your visitors will enhance the velocity of your own internet site considerably. As an example, should your company web site is centered on British people, be sure that your computer data server is near to your market.

Usually do not go with a totally free web host even though the services are cost-free. Hosting providers which are free usually force you to have adverts on your internet site. Sometimes, the ads will not be associated with your website and you will probably struggle to management what appears on your own site. Adverts will show up randomly, totally outside your manage. It will not only give your website an less than professional visual appeal, your audiences will more than likely resent the intrusion.

Browse the site of your own potential hosting company. In case the organization includes a doubtful, glitchy or otherwise in question internet site, prevent it. They could be a fresh firm without any expertise or perhaps a rip-off. A well created internet site shows they have a good focus when it comes to detail and so they have good expertise with regards to website design as well as in terms of Web coding.

Prior to signing in the dotted digital range, ensure that you understand the information on your determination by using a number company. You must know about hidden service fees their advertising don't talk about. Certain things, for example concealed service fees, expenses, and penalties when the contract's span isn't implemented by way of, could add up big time when you don't use caution.

Searching for a web host is actually much like looking about for whatever else. You need to understand exactly what you want then evaluate which can be a value that is good for you, then you need to find services which aligns with one of these objectives. With a little luck, with all the info you might have obtained using this post, it will be easy to achieve that.

reply

periodic timers - POSIX example

Mon, 11/26/2012 - 02:34 — Visitor (not verified)

timer_id should be in periodic_info stuct or exist as a global variable

Best regars,
Rafal

reply

periodic timers - POSIX example

Sat, 09/15/2012 - 01:32 — Jim Gibbons (not verified)

It looks like timer_id should have been in periodic_info.

reply

Detecting missed timer events

Tue, 02/23/2010 - 14:56 — csimmonds

One keen reader of my blog pointed out that I had misunderstood they way missed timer events are reported when using timerfd. Just to be clear, when you read, the value is a count of the timer events since the last one, which should be 1, not zero as I stated in an earlier draft. I have updated the article accordingly, and add in a comment about using timer_getoverrun() to do the same thing when you are using timers created with timer_create().

Bye for now,
Chris.

reply

Post new comment

Your name: *

Visitor

E-mail: *

The content of this field is kept private and will not be shown publicly.

Homepage:

Subject:

Comment: *

Web page addresses and e-mail addresses turn into links automatically.

Allowed HTML tags: <a>   <cite> <code>   <dl> <dt> <dd>

Lines and paragraphs break automatically.

More information about formatting options

By submitting this form, you accept the Mollom privacy policy.

Preview