使用定时器的目的无非是为了周期性的执行某一任务，或者是到了一个指定时间去执行某一个任务。要达到这一目的，一般有两个常见的比较有效的方法。一个是用 Linux 内部的三个定时器；另一个是用 sleep 或 usleep 函数让进程睡眠一段时间；其实，还有一个方法，那就是用 gettimeofday、difftime 等自己来计算时间间隔，然后时间到了就执行某一任务，但是这种方法效率低，所以不常用。

1、alarm

如果不要求很精确的话，用 alarm() 和 signal() 就够了

```
unsigned int alarm(unsigned int seconds)
```

- 1

专门为SIGALRM信号而设，在指定的时间seconds秒后，将向进程本身发送SIGALRM信号，又称为闹钟时间。进程调用alarm后，任何以前的alarm()调用都将无效。如果参数seconds为零，那么进程内将不再包含任何闹钟时间。如果调用alarm（）前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回0。

示例：

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void sigalrm_fn(int sig)
{
    printf("alarm!\n");
    alarm(2);
    return;
}

int main(void)
{
    signal(SIGALRM, sigalrm_fn);
    alarm(2);

    while(1) pause();
}
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

2、setitimer

```
int setitimer(int which, const struct itimerval *value, struct itimerval
*ovalue));
int getitimer(int which, struct itimerval *value);

strcut timeval
{
   long tv_sec; /*秒*/
   long tv_usec; /*微秒*/
};

struct itimerval
{
   struct timeval it_interval; /*时间间隔*/
   struct timeval it_value; /*当前时间计数*/
};
```

```
1
2
3
4
5
6
```

- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

setitimer() 比 alarm() 功能强大，支持3种类型的定时器：

① ITIMER_REAL：给一个指定的时间间隔，按照实际的时间来减少这个计数，当时间间隔为0的时候发出SIGALRM信号。

② ITIMER_VIRTUAL：给定一个时间间隔，当进程执行的时候才减少计数，时间间隔为0的时候发出SIGVTALRM信号。

③ ITIMER_PROF：给定一个时间间隔，当进程执行或者是系统为进程调度的时候，减少计数，时间到了，发出SIGPROF信号。

setitimer() 第一个参数 which 指定定时器类型（上面三种之一）；第二个参数是结构 itimerval 的一个实例；第三个参数可不做处理。

下面是关于setitimer调用的一个简单示范，在该例子中，每隔一秒发出一个SIGALRM，每隔0.5秒发出一个SIGVTALRM信号：：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>

int sec;

void sigroutine(int signo){
   switch (signo){
   case SIGALRM:
       printf("Catch a signal -- SIGALRM \n");
       signal(SIGALRM, sigroutine);
       break;
```

```
    case SIGVTALRM:
        printf("Catch a signal -- SIGVTALRM \n");
        signal(SIGVTALRM, sigroutine);
        break;
    }
    return;
}

int main()
{
    struct itimerval value, ovalue, value2;

    sec = 5;
    printf("process id is %d ", getpid());
    signal(SIGALRM, sigroutine);
    signal(SIGVTALRM, sigroutine);
    value.it_value.tv_sec = 1;
    value.it_value.tv_usec = 0;
    value.it_interval.tv_sec = 1;
    value.it_interval.tv_usec = 0;
    setitimer(ITIMER_REAL, &value, &ovalue);
    value2.it_value.tv_sec = 0;
    value2.it_value.tv_usec = 500000;
    value2.it_interval.tv_sec = 0;
    value2.it_interval.tv_usec = 500000;
    setitimer(ITIMER_VIRTUAL, &value2, &ovalue);
    for(;;);
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

```
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
```

该例子的执行结果如下：

```
localhost:~$ ./timer_test
process id is 579
Catch a signal  -  SIGVTALRM
Catch a signal  -  SIGALRM
Catch a signal  -  SIGVTALRM
Catch a signal  -  SIGVTALRM
Catch a signal  -  SIGALRM
Catch a signal  -GVTALRM
```

```
1
2
```

- 3
- 4
- 5
- 6
- 7
- 8

注意：Linux信号机制基本上是从Unix系统中继承过来的。早期Unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，因此，把那些建立在早期机制上的信号叫做”不可靠信号”，信号值小于SIGRTMIN(Red hat 7.2中，SIGRTMIN=32，SIGRTMAX=63)的信号都是不可靠信号。这就是”不可靠信号”的来源。它的主要问题是：进程每次处理信号后，就将对信号的响应设置为默认动作。在某些情况下，将导致对信号的错误处理；因此，用户如果不希望这样的操作，那么就要在信号处理函数结尾再一次调用 signal()，重新安装该信号。

3、用 sleep 以及 usleep 实现定时执行任务

```
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>

static char msg[] = "I received a msg.\n";
int len;

void show_msg(int signo)
{
    write(STDERR_FILENO, msg, len);
}

int main()
{
    struct sigaction act;
    union sigval tsval;
    act.sa_handler = show_msg;
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    sigaction(50, &act, NULL);
    len = strlen(msg);
```

```
    while ( 1 )
    {
        sleep(2); /*睡眠2秒*/
        /*向主进程发送信号，实际上是自己给自己发信号*/
        sigqueue(getpid(), 50, tsval);
    }
    return 0;
}
```

看到了吧，这个要比上面的简单多了，而且你用秒表测一下，时间很准，指定2秒到了就给你输出一个字符串。所以，如果你只做一般的定时，到了时间去执行一个任务，这种方法是最简单的。

4、通过自己计算时间差的方法来定时

```
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

static char msg[] = "I received a msg.\n";
int len;
static time_t lasttime;

void show_msg(int signo)
{
    write(STDERR_FILENO, msg, len);
}

int main()
{
    struct sigaction act;
    union sigval tsval;
    act.sa_handler = show_msg;
    act.sa_flags = 0;
    sigemptyset(&act.sa_mask);
    sigaction(50, &act, NULL);
    len = strlen(msg);
    time(&lasttime);
    while ( 1 )
    {
        time_t nowtime;
        /*获取当前时间*/
        time(&nowtime);
        /*和上一次的时间做比较，如果大于等于2秒，则立刻发送信号*/
        if (nowtime - lasttime >= 2)
        {
            /*向主进程发送信号，实际上是自己给自己发信号*/
            sigqueue(getpid(), 50, tsval);
            lasttime = nowtime;
```

```
        }
    }
    return 0;
}
```

```
39
40
```

这个和上面不同之处在于，是自己手工计算时间差的，如果你想更精确的计算时间差，你可以把 time 函数换成 gettimeofday，这个可以精确到微妙。

5、使用 select 来提供精确定时和休眠

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

```
1
```

n 指监视的文件描述符范围，通常设为所要select的fd＋1；readfds，writefds 和 exceptfds分别是读，写和异常文件描述符集；timeout 为超时时间。

可能用到的关于文件描述符集操作的宏有：

```
FD_CLR(int fd, fd_set *set);   // 清除fd
FD_ISSET(int fd, fd_set *set); // 测试fd是否设置
FD_SET(int fd, fd_set *set);   //设置fd
FD_ZERO(fd_set *set);          //清空描述符集
```

```
1
2
3
4
```

我们此时用不到这些宏，因为我们并不关心文件描述符的状态，我们关心的是select超时。所以我们需要把 readfds，writefds 和 exceptfds 都设为 NULL，只指定 timeout 时间就行了。至于 n 我们可以不关心，所以你可以把它设为任何非负值。实现代码如下：

```
int msSleep(long ms)
{
    struct timeval tv;
    tv.tv_sec = 0;
    tv.tv_usec = ms;

    return select(0, NULL, NULL, NULL, &tv);
}
```

怎么样，是不是很简单？ setitimer 和 select 都能实现进程的精确休眠，这里给出了一个简单的基于 select 的实现。我不推荐使用 setitimer，因为 Linux 系统提供的 timer 有限（每个进程至多能设3个不同类型的 timer），而且 setitimer 实现起来没有 select 简单。

6、高精度硬件中断定时器 hrtimer

需要在 kernel 中打开 “high resolution Timer support”，驱动程序中 hrtimer 的初始化如下：

```
hrtimer_init(&m_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL_PINNED);
m_timer.function = vibrator_timer_func;
hrtimer_start(&m_timer, ktime_set(0, 62500), HRTIMER_MODE_REL_PINNED);
```

定时函数 vibrator_timer_func 如下：

```
static enum hrtimer_restart vibrator_timer_func(struct hrtimer *timer)
{
  gpio_set_value(gpio_test, 1);
  gpio_set_value(gpio_test, 0);
  hrtimer_forward_now(&m_timer,ktime_set(0, 62500));
  return HRTIMER_RESTART;
}
```

- 5
- 6
- 7

其中 gpio_test 为输出引脚，为了方便输出查看。但是用示波器查看引脚波形时，发现虽然设定的周期为62.5us，但是输出总是为72us左右，而且偶尔会有两个波形靠的很近（也就是说周期突然变为10us以下）。我将周期设到40us的话，就会出现72us和10us经常交替出现，无法实现精确的40us的波形，如果设置到100us时，则波形就是100us了，而且貌似没有看到有10us以下的周期出现。

7、高精度定时器 posix_timer

最强大的定时器接口来自POSIX时钟系列，其创建、初始化以及删除一个定时器的行动被分为三个不同的函数：`timer_create()`(创建定时器)、`timer_settime()`(初始化定时器)以及 `timer_delete()`(销毁它)。

创建一个定时器：

```
int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timerid)
```

- 1

进程可以通过调用 `timer_create()` 创建特定的定时器，定时器是每个进程自己的，不是在 fork 时继承的。`clock_id` 说明定时器是基于哪个时钟的，*timerid 装载的是被创建的定时器的 ID。该函数创建了定时器，并将他的 ID 放入timerid指向的位置中。参数evp指定了定时器到期要产生的异步通知。如果evp为 NULL，那么定时器到期会产生默认的信号，对 `CLOCK_REALTIMER`来说，默认信号就是SIGALRM。如果要产生除默认信号之外的其它信号，程序必须将 `evp->sigev_signo`设置为期望的信号码。struct sigevent 结构中的成员 `evp->sigev_notify`说明了定时器到期时应该采取的行动。通常，这个成员的值为`SIGEV_SIGNAL`，这个值说明在定时器到期时，会产生一个信号。程序可以将成员`evp->sigev_notify`设为`SIGEV_NONE`来防止定时器到期时产生信号。

如果几个定时器产生了同一个信号，处理程序可以用 evp->sigev_value来区分是哪个定时器产生了信号。要实现这种功能，程序必须在为信号安装处理程序时，使用struct sigaction的成员sa_flags中的标志符SA_SIGINFO。

clock_id取值为以下：

CLOCK_REALTIME :Systemwide realtime clock.

CLOCK_MONOTONIC:Represents monotonic time. Cannot be set.

CLOCK_PROCESS_CPUTIME_ID :High resolution per-process timer.

CLOCK_THREAD_CPUTIME_ID :Thread-specific timer.

CLOCK_REALTIME_HR :High resolution version of CLOCK_REALTIME.

CLOCK_MONOTONIC_HR :High resolution version of CLOCK_MONOTONIC.

```
struct sigevent
{
int sigev_notify; //notification type
int sigev_signo; //signal number
union sigval   sigev_value; //signal value
void (*sigev_notify_function)(union sigval);
pthread_attr_t *sigev_notify_attributes;
}
union sigval
{
int sival_int; //integer value
void *sival_ptr; //pointer value
}
```

- 12
- 13

通过将evp->sigev_notify设定为如下值来定制定时器到期后的行为：

SIGEV_NONE：什么都不做，只提供通过timer_gettime和timer_getoverrun查询超时信息。

SIGEV_SIGNAL: 当定时器到期，内核会将sigev_signo所指定的信号传送给进程。在信号处理程序中，si_value会被设定会sigev_value。

SIGEV_THREAD: 当定时器到期，内核会(在此进程内)以sigev_notification_attributes为线程属性创建一个线程，并且让它执行sigev_notify_function，传入sigev_value作为为一个参数。

启动一个定时器：

timer_create()所创建的定时器并未启动。要将它关联到一个到期时间以及启动时钟周期，可以使用timer_settime()。

```
int timer_settime(timer_t timerid, int flags, const struct itimerspec *value,
struct itimerspect *ovalue);

struct itimespec{
    struct timespec it_interval;
    struct timespec it_value;
};
```

- 1
- 2
- 3
- 4
- 5
- 6

如同settimer()，it_value用于指定当前的定时器到期时间。当定时器到期，it_value的值会被更新成it_interval 的值。如果it_interval的值为0，则定时器不

是一个时间间隔定时器，一旦it_value到期就会回到未启动状态。timespec的结构提供了纳秒级分辨率：

```
struct timespec{
    time_t tv_sec;
    long tv_nsec;
};
```

如果flags的值为TIMER_ABSTIME，则value所指定的时间值会被解读成绝对值(此值的默认的解读方式为相对于当前的时间)。这个经修改的行为可避免取得当前时间、计算“该时间”与“所期望的未来时间”的相对差额以及启动定时器期间造成竞争条件。

如果ovalue的值不是NULL，则之前的定时器到期时间会被存入其所提供的itimerspec。如果定时器之前处在未启动状态，则此结构的成员全都会被设定成0。

获得一个活动定时器的剩余时间：

```
int timer_gettime(timer_t timerid,struct itimerspec *value);
```

取得一个定时器的超限运行次数：

有可能一个定时器到期了，而同一定时器上一次到期时产生的信号还处于挂起状态。在这种情况下，其中的一个信号可能会丢失。这就是定时器超限。程序可以通过调用timer_getoverrun来确定一个特定的定时器出现这种超限的次数。定时器超限只能发生在同一个定时器产生的信号上。由多个定时器，甚至是那些使用相同的时钟和信号的定时器，所产生的信号都会排队而不会丢失。

```
int timer_getoverrun(timer_t timerid);
```

执行成功时，timer_getoverrun()会返回定时器初次到期与通知进程(例如通过信号)定时器已到期之间额外发生的定时器到期次数。举例来说，在我们之前的例子中，一个1ms的定时器运行了10ms，则此调用会返回9。如果超限运行的次数等于或大于DELAYTIMER_MAX，则此调用会返回DELAYTIMER_MAX。

执行失败时，此函数会返回-1并将errno设定会EINVAL，这个唯一的错误情况代表timerid指定了无效的定时器。

删除一个定时器：

```
int timer_delete (timer_t timerid);
```

- 1

一次成功的timer_delete()调用会销毁关联到timerid的定时器并且返回0。执行失败时，此调用会返回-1并将errno设定会 EINVAL，这个唯一的错误情况代表timerid不是一个有效的定时器。

例1：

```
void  handle()
{
 time_t t;
 char p[32];
 time(&t);
 strftime(p, sizeof(p), "%T", localtime(&t));
 printf("time is %s\n", p);
}

int main()
{
 struct sigevent evp;
 struct itimerspec ts;
 timer_t timer;
 int ret;
 evp.sigev_value.sival_ptr = &timer;
 evp.sigev_notify = SIGEV_SIGNAL;
 evp.sigev_signo = SIGUSR1;
 signal(SIGUSR1, handle);
 ret = timer_create(CLOCK_REALTIME, &evp, &timer);
```

```
 if( ret )
  perror("timer_create");
 ts.it_interval.tv_sec = 1;
 ts.it_interval.tv_nsec = 0;
 ts.it_value.tv_sec = 3;
 ts.it_value.tv_nsec = 0;
 ret = timer_settime(timer, 0, &ts, NULL);
 if( ret )
  perror("timer_settime");
 while(1);
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31

例2：

```
void  handle(union sigval v)
{
 time_t t;
 char p[32];
 time(&t);
 strftime(p, sizeof(p), "%T", localtime(&t));
 printf("%s thread %lu, val = %d, signal captured.\n", p, pthread_self(),
v.sival_int);
 return;
}

int main()
{
 struct sigevent evp;
 struct itimerspec ts;
 timer_t timer;
 int ret;
 memset   (&evp,   0,   sizeof   (evp));
 evp.sigev_value.sival_ptr = &timer;
 evp.sigev_notify = SIGEV_THREAD;
 evp.sigev_notify_function = handle;
 evp.sigev_value.sival_int = 3;   //作为handle()的参数
 ret = timer_create(CLOCK_REALTIME, &evp, &timer);
 if( ret)
  perror("timer_create");
 ts.it_interval.tv_sec = 1;
 ts.it_interval.tv_nsec = 0;
 ts.it_value.tv_sec = 3;
 ts.it_value.tv_nsec = 0;
 ret = timer_settime(timer, TIMER_ABSTIME, &ts, NULL);
 if( ret )
  perror("timer_settime");
 while(1);
}
```