Sleeping and Timers

Advanced Embedded Linux Development

with **Dan Walkes**

University of Colorado **Boulder**

Learning objectives:

Sleeping in your program

Alternatives to Sleeping

Using Timers in your program

Sleeping

```
SLEEP(3)                                  Linux Programmer's Manual

NAME
       sleep - sleep for a specified number of seconds

SYNOPSIS
       #include <unistd.h>

       unsigned int sleep(unsigned int seconds);
```

- Returns number of seconds *not* slept
- Why would this function exit early?
  - signals

```
USLEEP(3)                                 Linux Programmer's Manual

NAME
       usleep - suspend execution for microsecond intervals

SYNOPSIS
       #include <unistd.h>

       int usleep(useconds_t usec);
```

- Sleep with microsecond precision *at least* usec

Sleeping

```
NANOSLEEP(2)                          Linux Programmer's Manual

NAME
       nanosleep - high-resolution sleep

SYNOPSIS
       #include <time.h>

       int nanosleep(const struct timespec *req, struct timespec *rem);
```

- Sleep with nanosecond precision
- Returns remaining time in *rem if interrupted

Sleeping

```
CLOCK_NANOSLEEP(2)                    Linux Programmer's Manual

NAME
       clock_nanosleep - high-resolution sleep with specifiable clock

SYNOPSIS
       #include <time.h>

       int clock_nanosleep(clockid_t clock_id, int flags,
                           const struct timespec *request,
                           struct timespec *remain);
```

- Supports absolute sleeping with TIMER_ABSTIME flag and absolute time in request
  - Can be used for more precise sleep sequences.
  - Avoids inaccuracies in timing between clock_gettime() and clock_nanosleep()

Sleeping

```
/**
* Get the starting time before starting sleep
*/
int rc = clock_gettime(clock_type,&ts_start);
if( rc != 0 ) {
    printf("Error %d (%s) getting CLOCK_MONOTONIC start time\n",
        errno,strerror(errno));
} else {
                .
    unsigned int ret = sleep(sleep_time.tv_sec);
    if (ret != 0) {
        printf("sleep interrupted by signal!\n");
        success = false;
    }
    break;

   rc = clock_gettime(clock_type,&ts_end);
   if (rc != 0) {
        printf("Error %d (%s) getting end time for clock type %d\n",
            errno,strerror(errno),clock_type);
        success = false;
    }
```

- What should I use for clock_type to measure duration?
  - CLOCK_MONOTONIC

https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture9/sleep_functions.c

Sleeping

```
aesd@aesd-VirtualBox:~/aesd-lectures/lecture9$ ./sleep_functions
Sleeping for 1.000000 seconds using sleep type sleep
For sleep type sleep, requested sleep time 1.000000 seconds actual sleep time 1.000326 seconds, difference of 0.000326 seconds
Sleeping for 1.000000 seconds using sleep type usleep
For sleep type usleep, requested sleep time 1.000000 seconds actual sleep time 1.000054 seconds, difference of 0.000054 seconds
Sleeping for 1.000000 seconds using sleep type nanosleep
For sleep type nanosleep, requested sleep time 1.000000 seconds actual sleep time 1.000350 seconds, difference of 0.000350 seconds
Sleeping for 1.000000 seconds using sleep type clock_nanosleep
For sleep type clock_nanosleep, requested sleep time 1.000000 seconds actual sleep time 1.000565 seconds, difference of 0.000565 seconds
```

- Is clock_nanosleep less accurate?
  - No all sleep functions are generally not accurate below 1ms on this system.

https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture9/sleep_functions.c

Sleeping

```
/**
* Get the starting time before starting sleep
*/
int rc = clock_gettime(clock_type,&ts_start);
if( rc != 0 ) {
    printf("Error %d (%s) getting start time for clock type %d\n",
        errno,strerror(errno),clock_type);
} else {
    success = true;
    if( delay_time_ms > 0 ) {
        rc = usleep(delay_time_ms*1000);
        if (rc != 0) {
            printf("usleep interrupted with errno %d in delay (%s)\n",errno,strerror(errno));
            success = false;
        }
    }
```

- Add a 100 ms delay between clock_gettime and actual sleep to show benefit of clock_nanosleep and TIME_ABSTIME

```
aesd@aesd-VirtualBox:~/aesd-lectures/lecture9$ ./sleep_functions --delay 100
Delaying for 100 milliseconds between clock_gettime and sleep call.
Sleeping for 1.000000 seconds using sleep type sleep
For sleep type sleep, requested sleep time 1.000000 seconds actual sleep time 1.100677 seconds, difference of 0.100677 seconds
Sleeping for 1.000000 seconds using sleep type usleep
For sleep type usleep, requested sleep time 1.000000 seconds actual sleep time 1.100448 seconds, difference of 0.100448 seconds
Sleeping for 1.000000 seconds using sleep type nanosleep
For sleep type nanosleep, requested sleep time 1.000000 seconds actual sleep time 1.100849 seconds, difference of 0.100849 seconds
Sleeping for 1.000000 seconds using sleep type clock_nanosleep
For sleep type clock_nanosleep, requested sleep time 1.000000 seconds actual sleep time 1.000840 seconds, difference of 0.000840 seconds
```

https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture9/sleep_functions.c

Alternatives to Sleeping

- Sleep is appropriate for short (<1 sec) infrequent events.
- Code that is laced with sleeps in order to “busy-wait” for events is usually of poor design.
- Code that blocks on a file descriptor, allowing the kernel to handle the sleep and wake up the process, is better.

Alternatives to Sleeping

- Avoid the urge to use as band-aids for some poorly understood problem.
  - “I got an error once, then I added a sleep and it disappeared.”
- Consider using Timers instead.

Timers - alarm

```
ALARM(2)          Linux Programmer's Manual          ALARM(2)

NAME
       alarm - set an alarm clock for delivery of a signal

SYNOPSIS
       #include <unistd.h>

       unsigned int alarm(unsigned int seconds);

DESCRIPTION
       alarm()  arranges for a SIGALRM signal to be delivered
       to the calling process in seconds seconds.
```

```c
int main ( int argc, char **argv )
{
    struct sigaction new_action;
    bool success = true;
    memset(&new_action,0,sizeof(struct sigaction));
    new_action.sa_handler=signal_handler;
    if( sigaction(SIGALRM, &new_action, NULL) != 0 ) {
        printf("Error %d (%s) registering for SIGALRM",errno,strerror(errno));
        success = false;
    }
    if( success ) {
        alarm(2);
        printf("Waiting 2 seconds for alarm signal\n");
        pause();
        if( caught_sigalarm ) {
            printf("Caught SIGALARM!\n");
        } else {
            printf("Error: Did not catch SIGALARM!\n");
            success = false;
        }
    }
```

- Simplest interface
- Delivers SIGALRM to the process after **seconds** of time have expired.
- SIGALRM can be sent outside the process

Linux System Programming Chapter 11
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture9/timer_simple.c

alarm concerns

- Signal reentrancy limitations
- SIGALRM can be sent by sleep(), usleep() setitimer()
- SIGALRM can be sent outside the process

```c
bool caught_sigalarm = false;

static void signal_handler ( int signal_number )
{
    int errno_saved = errno;
    if ( signal_number == SIGALRM ) {
        caught_sigalarm = true;
    }
    errno = errno_saved;
}

int main ( int argc, char **argv )
{
    struct sigaction new_action;
    bool success = true;
    memset(&new_action,0,sizeof(struct sigaction));
    new_action.sa_handler=signal_handler;
    if( sigaction(SIGALRM, &new_action, NULL) != 0 ) {
        printf("Error %d (%s) registering for SIGALRM",errno,strerror(errno));
        success = false;
    }
    if( success ) {
        alarm(2);
        printf("Waiting 2 seconds for alarm signal\n");
        pause();
        if( caught_sigalarm ) {
            printf("Caught SIGALARM!\n");
        } else {
            printf("Error: Did not catch SIGALARM!\n");
            success = false;
        }
    }
```

Linux System Programming Chapter 11
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture9/timer_simple.c

Timers - Interval Timers

```
GETITIMER(2)                          Linux Programmer's Manual

NAME
       getitimer, setitimer - get or set value of an interval timer

SYNOPSIS
       #include <sys/time.h>

       int getitimer(int which, struct itimerval *curr_value);
       int setitimer(int which, const struct itimerval *new_value,
                     struct itimerval *old_value);
```

```
struct itimerval {
    struct timeval it_interval; /* Interval for periodic timer */
    struct timeval it_value;    /* Time until next expiration */
};
```

```
struct timeval {
    time_t      tv_sec;         /* seconds */
    suseconds_t tv_usec;        /* microseconds */
};
```

- Delivers SIGALRM to the process, optionally re-arming itself with it_interval after it_value
- Similar issues with signal handlers

Timers - POSIX timer

- Use threads instead of signals with SIGEV_THREAD
- Adds support for TIMER_ABSTIME

```
TIMER_CREATE(2)                          Linux Programmer's Manual

NAME
       timer_create - create a POSIX per-process timer

SYNOPSIS
       #include <signal.h>
       #include <time.h>

       int timer_create(clockid_t clockid, struct sigevent *sevp,
                        timer_t *timerid);
```

```
TIMER_SETTIME(2)

NAME
       timer_settime, timer_gettime - arm/disarm and fetch state of POSIX per-process timer

SYNOPSIS
       #include <time.h>

       int timer_settime(timer_t timerid, int flags,
                         const struct itimerspec *new_value,
                         struct itimerspec *old_value);
```

```
TIMER_DELETE(2)                                          Linux I

NAME
       timer_delete - delete a POSIX per-process timer

SYNOPSIS
       #include <time.h>

       int timer_delete(timer_t timerid);
```

```
Li  struct timespec {
        time_t tv_sec;                /* Seconds */
        long   tv_nsec;               /* Nanoseconds */
    };

    struct itimerspec {
        struct timespec it_interval;  /* Timer interval */
        struct timespec it_value;     /* Initial expiration */
    };
```

Timers - POSIX timer example

```c
int clock_id = CLOCK_MONOTONIC;
memset(&sev,0,sizeof(struct sigevent));
/**
* Setup a call to timer_thread passing in the td structure as the sigev_value
* argument
*/
sev.sigev_notify = SIGEV_THREAD;
sev.sigev_value.sival_ptr = &td;
sev.sigev_notify_function = timer_thread;
if ( timer_create(clock_id,&sev,&timerid) != 0 ) {
    printf("Error %d (%s) creating timer!\n",errno,strerror(errno));
} else {
    struct timespec sleep_time;
    /**
    * Set sleep time to 2.001 s to ensure the last ms aligned timer event is fired
    */
    sleep_time.tv_sec = 2;
    sleep_time.tv_nsec = 1000000;
    success = run_timer_test(clock_id,timerid,10,&sleep_time,&td);
    if (timer_delete(timerid) != 0) {
        printf("Error %d (%s) deleting timer!\n",errno,strerror(errno));
        success = false;
    }
}
```

```c
/**
* A thread which runs every timer_period_ms milliseconds
* Assumes timer_create has configured for sigval.sival_ptr to point to the
* thread data used for the timer
*/
static void timer_thread ( union sigval sigval )
{
    struct thread_data *td = (struct thread_data*) sigval.sival_ptr;
    if ( pthread_mutex_lock(&td->lock) != 0 ) {
        printf("Error %d (%s) locking thread data!\n",errno,strerror(errno));
    } else {
        td->timer_count++;
        if ( pthread_mutex_unlock(&td->lock) != 0 ) {
            printf("Error %d (%s) unlocking thread data!\n",errno,strerror(errno));
        }
    }
}
```

```
aesd@aesd-VirtualBox:~/aesd-lectures/lecture9$ ./timer_thread
Sleeping for 2.001000 seconds while timer is running once every 10 ms
Timer count 200 matches expected value for sleep time 2.001000 seconds and timer period 10 ms
```

Linux System Programming Chapter 11

https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture9/timer_thread.c

Timers - POSIX timer example

```c
int clock_id = CLOCK_MONOTONIC;
memset(&sev,0,sizeof(struct sigevent));
/**
* Setup a call to timer_thread passing in the td structure as the sigev_value
* argument
*/
sev.sigev_notify = SIGEV_THREAD;
sev.sigev_value.sival_ptr = &td;
sev.sigev_notify_function = timer_thread;
if ( timer_create(clock_id,&sev,&timerid) != 0 ) {
    printf("Error %d (%s) creating timer!\n",errno,strerror(errno));
} else {
    struct timespec sleep_time;
    /**
    * Set sleep time to 2.001 s to ensure the last ms aligned timer event is fired
    */
    sleep_time.tv_sec = 2;
    sleep_time.tv_nsec = 1000000;
    success = run_timer_test(clock_id,timerid,10,&sleep_time,&td);
    if (timer_delete(timerid) != 0) {
        printf("Error %d (%s) deleting timer!\n",errno,strerror(errno));
        success = false;
    }
}
```

```c
/**
* A thread which runs every timer_period_ms milliseconds
* Assumes timer_create has configured for sigval.sival_ptr to point to the
* thread data used for the timer
*/
static void timer_thread ( union sigval sigval )
{
    struct thread_data *td = (struct thread_data*) sigval.sival_ptr;
    if ( pthread_mutex_lock(&td->lock) != 0 ) {
        printf("Error %d (%s) locking thread data!\n",errno,strerror(errno));
    } else {
        td->timer_count++;
        if ( pthread_mutex_unlock(&td->lock) != 0 ) {
            printf("Error %d (%s) unlocking thread data!\n",errno,strerror(errno));
        }
    }
}
```

- Can use functions which aren't signal safe in timer_thread

Linux System Programming Chapter 11
https://github.com/cu-ecen-5013/aesd-lectures/blob/master/lecture9/timer_thread.c