Computer Science Department

CSCI 247 Computer Systems I
Assignment Exercise 2

Objectives

Gain familiarity with system timer applications. In particular, understand how to schedule periodic activity.

Submitting Your Work

Submit your C program files as the Assignment Exercise 2 Submission item on the course web site. You must submit your program along with an explanation of the results by the deadline in 2 weeks.

Background

All robust software must be able to field notifications graciously. Sometimes notifications are associated with unexpected and unintended circumstances like a user hitting ctrl+c because the system is unresponsive. At other times notifications is designed to fulfill the very mission of the software such as fielding a hardware interrupt that signals the need to read some I/O data. In both these cases, the software developer must be familiar with the interfaces available in a computer system to program and field notifications.

All POSIX (Portable Operating System Interface – IEEE family of standards for OS compatibility) complaint Operating Systems support "**Signals**" as a standard form of notifications. These notification indicate that some "event" has occurred. A "Signal" is similar to an interrupt in that it is asynchronous in nature, but differs from an interrupt in that it is initiated by the Kernel as opposed to the CPU (or hardware). This is subject to a nuanced definition, but will suffice in this context.

Signals may be classified as "Standard" or "Real-time" (see here for a list of predefined signals). "SIGSEGV" (Segmentation fault) is one you have likely encountered previously. The SIGSEGV signal is generated when you access an invalid location in memory.

Standard Signals refer to predefined events, while Real-time Signals allow the software developer to associate application specific meaning to them. Real-time Signals are guaranteed to be delivered in the same order in which they were sent. The Linux OS supports approximately 30 different real-time signals ranging from SIGRTMIN to SIGRTMAX (note that Linux actually supports 33, but 2 or 3 of these are reserved).

Every thread has a mask defining which Signals it wishes to block. A thread inherits this mask from its parent but can change its mask at any time by calling "**pthread_sigmask**" (For non-multithreaded applications, use "**sigprocmask**"). Blocking a Signal, just means that the Signal will not be delivered to the thread that has blocked it. It does not stop a sender from sending the Signal. A signal that is generated but not delivered is said to be in the "Pending" state.

A thread may synchronously wait on a Signal using the "**sigwait**" API even if it had previously blocked the Signal in its mask. The blocking only impacts the delivery of the Signal asynchronously. It does not stop the thread from actually checking on and receiving the signal synchronously.

"**timer_create**" is an API that allows you to create a timer and associate it with any Real-time Signal. Below is an example of how to use this API assuming you have a "threadInfo" structure that defines a variable "signalValue" of type "int" and a variable "timerID" for type "timer_t".

```
struct sigevent mySignalEvent;

// Create a timer associated with real-time signal
mySignalEvent.sigev_notify = SIGEV_SIGNAL;
mySignalEvent.sigev_signo = threadInfo->signalValue; //A value b/t SIGRTMIN and SIGRTMAX
mySignalEvent.sigev_value.sival_ptr = (void *)&(threadInfo->timerId);
ret = timer_create (CLOCK_MONOTONIC, &mySignalEvent, &threadInfo ->timerId);
```

Note that a newly created timer is disarmed. To arm a timer, you have call the "**timer_settime**" API. Here is an example of a call to timer_settime.

```
struct itimerspec timerSpec;

// Arm the timer for a period defined in microseconds
seconds = period/1000000;
nanoseconds = (period - (sec * 1000000)) * 1000;
timerSpec.it_interval.tv_sec = seconds;
timerSpec.it_interval.tv_nsec = nanoseconds;
timerSpec.it_value.tv_sec = seconds;
timerSpec.it_value.tv_nsec = nanoseconds;
ret = timer_settime (threadInfo ->timer_id, 0, & timerSpec, NULL);
```

Goal

In this assignment you will create 3 FIFO threads from your main thread and each of these 3 threads will create a periodic timer that will trigger a Real-time Signal that only the thread that created the timer receives synchronously using "sigwait". The period of the timers in each of the 3 threads will be 1s, 2s and 4s respectively. Each thread will wait for a timer notification 5 times and you will output the actual time taken to receive the notification in each thread as follows:

```
Thread[0]    Timer Delta [1000009]us          Jitter [9]us
```

```
Thread[0]   Timer Delta [999946]us    Jitter [-54]us
Thread[1]   Timer Delta [2000009]us   Jitter [9]us
Thread[0]   Timer Delta [999897]us    Jitter [-103]us
Thread[0]   Timer Delta [999971]us    Jitter [-29]us
Thread[1]   Timer Delta [1999974]us   Jitter [-26]us
Thread[2]   Timer Delta [4000004]us   Jitter [4]us
Thread[0]   Timer Delta [999897]us    Jitter [-103]us
Thread[1]   Timer Delta [2000057]us   Jitter [57]us
Thread[1]   Timer Delta [1999993]us   Jitter [-7]us
Thread[2]   Timer Delta [4000037]us   Jitter [37]us
Thread[1]   Timer Delta [1999953]us   Jitter [-47]us
Thread[2]   Timer Delta [3999969]us   Jitter [-31]us
Thread[2]   Timer Delta [4000076]us   Jitter [76]us
Thread[2]   Timer Delta [3999847]us   Jitter [-153]us
```

The "Timer Delta" field records the Delta between going to wait for the Signal and receiving the signal. The "Jitter" field records the difference between the defined period for a given thread and the actual period observed.

Assignment Hints

Task 1: Refactor the code in Assignment 1

Since this assignment requires you to spin 3 FIFO threads, you can use this opportunity to refactor the code in Assignment 1 and organize your code in a more easily scalable form.

Task 2: Create a generic "CreateAndArmTimer" function

This is a function that can be called from any thread to create and arm a timer associated with the next available Real-time Signal.

Task 3: Create a generic "WaitForTimer" function

This is a function that can be called from any thread to wait for the timer created by this thread

Task 4: Modify your thread function

Modify your thread function call the new "CreateAndArmTimer" function and then call "WaitForTimer" in a loop 5 times.

Pseudo code

```
//Add following elements to Thread structure in Assignment 1

        int         signal_number;
        sigset_t    timer_signal;
        int         missed_signal_count;
        timer_t     timer_Id;
        int         timer_Period;

//Add following new functions

CreateAndArmTimer(int unsigned period, ThreadArgs* threadInfo)
{
        //Create a static int variable to keep track of the next available signal number

        //Initialize the thread structure elements

        //Assign the next available Real-time signal to thread "signal_number"

        //Create the signal mask corresponding to the chosen signal_number in "timer_signal"
        //Use "sigemptyset" and "sigaddset" for this

        //Use timer_Create to create a timer – See code in background section

        //Arm Timer – See code in background section
}

WaitForTimer(ThreadArgs* threadInfo)
{
        //Use sigwait function to wait on the "timer_signal"

        //update missed_signal_count by calling "timer_getoverrun"
}
```

Modify your main thread to block all real-time signals by using "sigemptyset" and "sigaddset" (add all signals between SIGRTMIN and SIGRTMAX) to create a signal set and then call "pthread_sigmask" with SIG_BLOCK.

Modify your thread function to call the above two functions and keep tabs on how long you wait for the timer each time. Each time to come out of the Wait, calculate the time spent in the Wait and the Jitter.

Grading guidelines

- Working code (4 points)
- Conforming to good coding practice (4 points)
- Analysis of jitter output (2 points)