Lab 05 – Signals

Objectives

1. Learn about signals.
2. Learn about kernel timers.
3. Write C programs that utilize timers and signals.

Pre-Reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

Signals

Signals are a ***limited*** form of **inter-process communication** (IPC) used in Unix-like operating systems. A signal is an asynchronous notification sent to a process (or to a specific thread within the same process) in order to notify it of an event that occurred.

When a signal is sent, the operating system ***interrupts*** the target process's normal flow of execution to deliver the signal. Execution can be interrupted during any **non-atomic instruction**. If the process has previously registered a **signal handler**, that function is executed. Otherwise, the default signal handler (defined generically by the operating system) is executed.

Signals can be used as a means of communication between any two processes. For example, the kernel process will often send signals to application-level processes to inform them of a hardware related or OS related event. Every signal has a **name** and an **integer value**, either of which can be used to refer to the given signal. Most UNIX-like operating systems tend to define the same 31 signals; however their associated names and integer values may vary between operating system families.

Most signals are meant to be used in a particular situation. You are not allowed to create new signals; however, most UNIX-like operating systems include the signals SIGUSR1 and SIGURSR2, which can be used for general application-to-application signaling. That is, they have no OS pre-defined purpose.

To send a signal to another process via the command line, use the **`kill`** command. For example, the command

```
$ kill -2 13
```

sends the SIGINT command (integer value 2) to the process with pid 13. The **pid** of a process is its process ID, and is used to identify the process. Use the **`ps`** command to view the current list of processes on your system, as well as their associated pids.

Signals in C

A signal is raised by calling the **`kill()`** function and identifying the process to receive the signal and the signal type. The receiving process can cause the signal to be handled in the default way, to be ignored, or to be caught by user-defined code. The **`signal()`** function is called to specify how a given signal is to be handled. For example, to ignore the SIGALRM signal, the process must execute the system call

```
signal(SIGALRM, SIG_IGN);
```

where **`SIG_IGN`** means *ignore the signal*.

The default handling can be re-enabled by calling **`signal()`** again with the **`SIG_DFL`** value. The application can process the alarm signal with its own code by supplying a function (that takes an integer argument and returns a void) as the second argument to **`signal()`**.

The signal names are defined as constants (using the preprocessor directive **#define**) in the header file **signal.h**. For example,

```
#define SIGINT 2
```

defines a constant for the **`SIGINT`** signal. Hence, you can refer to this signal by name in you C code (recommended), and when you build the program, the preprocessor replaces every instance of **`SIGINT`** with its associated integer value (2).

Another method for installing a signal handler that offers more flexibility is by using the **`sigaction()`** system call. This system call tells the kernel that this program wants to be notified via a software interrupt when the given event occurs. There are user defined signals that you can use, or you can register with system events. The most common is SIGINT, which is raised when you type Control + C on your keyboard. Your program can optionally catch this and do something useful before closing. More powerful is **`SIGKILL`**; you can't catch this one because it immediately ends the process.

The **`sigaction`** structure looks like:

```
struct sigaction {
      void (*sa_handler)(int);
      void (*sa_sigaction)(int, siginfo_t *, void *);
      sigset_t sa_mask;
      int sa_flags;
      void (*sa_restorer)(void);
}
```

and you **install** (i.e. register) a handler in your code like this:

```
struct sigaction sa;
memset (&sa, 0, sizeof (sa));
sa.sa_handler = &handler;
sigaction (SIGUSR1, &sa, NULL);
```

where **handler** is the name of the function you want to associate with the **SIGUSR1** signal. The above four lines do the same thing as:

```
signal(SIGUSR1, handler);
```

but in a manner that's portable to different UNIX versions.

After this, you can go on with your code and have one function that will run in response to that signal/event. But be careful; you must keep that function to a minimum because it could be called often, and each time it is called, it is *literally* interrupting your program.

Note: if a global variable can be modified by both your code and your handler code, then you may want to look up the variable type **sig_atomic_t**. If the signal interrupts your program just after the global variable has been, say, momentarily set to a temporary value, and your signal handler changes the value, then unpredictable results can occur.

Sample C Program with Signal Handler

The following complete program illustrates how a signal handler is registered with the **signal()** function call and how the whole mechanism operates, as well as the form of the signal handler routine itself.

File: sighandlerex.c

```
#include  <signal.h>

static void sig_handler(int);

int main (void) {
      int i, parent_pid, child_pid, status;

      /*Prepare the sig_handler routine to catch SIGUUSR1 and SIGUSR2 */
      if (signal(SIGUSR1, sig_handler)==SIG_ERR)
            printf("parent:Unable to create handler for SIGUSR1\n");
      if (signal(SIGUSR2, sig_handler)==SIG_ERR
                  printf("Parent: unable to create handler for SIGUSR2\n");

      parent_pid = getpid();

      if ((child_pid = fork())==0) {
            /*Raise the SIGUSR1 signal */
            kill(parent_pid, SIGUSR1);
            /*Child process begins busy-wait for a signal */
            for (;;) pause ();
      }
      else {
                  /*Parent raising SIGUSR2 signal*/
                  kill(child_pid, SIGUSR2);
                  printf("Parent: Terminating child ...");
                  /*Parent raising SIGTERM signal */
                  kill(child_pid, SIGTERM);
                  /* Parent waiting for child termination */
                  wait(&status);
                  printf("done\n");
      }
}
```

```
static void sig_handler(int signo) {
      switch (signo) {

            /*Incoming SIGUSR1 signal */
            case SIGUSR1:
                        printf("Parent: Received SIGUSR1\n");
                        break;

            /*Incoming SIGUSR2 signal */
            case SIGUSR2:
                  printf("Child: Received SIGUSR2\n");
                  break;

            /* Should never get this case */
            default: break;

      }
      return;
}
```

This code segment illustrates how signals are raised and caught but does not implement any useful function. It creates a single handler, **`sig_handler`**, that is used by a parent process and a child process with the two calls to **`signal()`**. Next, it determines its own process identifier by using the **`getpid()`** system call. It then creates a child so that the parent knows both the parent and child process identifiers but the child knows only the parent's identifier. The child process sends a **`SIGUUSR1`** to the parent and then enters a busy-wait so that it will still exist when the parent sends signals to it. The parent sends a **`SIGUSR2`** to the child, followed by a termination signal (**`SIGTERM`**), and then calls **`wait()`** to obtain the report that the child has been terminated. The child and parent use the same signal handler definition, though child will never see the **`SIGUSR1`** and the parent will never see a **`SIGUSR2`** signal.

Kernel Timers

The Kernel keeps the current time by reading a clock device and maintaining a kernel variable with the current time. The current time is accessible to user mode programs via system calls.

The function **`gettimeofday()`** is the usual interface to the current system time. The OS uses this call frequently for scheduling purposes. Such as deciding if a running process should be removed from the CPU so that another process can use it, or to keep track of the amount of time that a process executed in user mode or supervisor mode. Time stamping events and calculating elapsed time is something the operating system and user level programs must do to ensure they are responsive and/or measure performance.

Time is stated relative to some important epoch. For example, time in the United States is calculated by using the Gregorian calendar, which is based on a time of zero to be about 2,000 years ago. When you type the **`date`** command to a shell, the command will read the kernel variable to determine the time, say **Mon Jun 21 09:01:28 MDT 2001**, which can be interpreted to mean that about 2001.5 years have elapsed since the beginning of epoch.

UNIX systems were not around before about 1970, so they avoid representing time before 1970. This is accomplished by beginning the time epoch at 12 a.m. January 1, 1970 (00:00:00 Greenwich Meridian Time (GMT)). Two long **int** kernel variables keep track of the number of seconds and microseconds, respectively, that have passed since the beginning of the UNIX time epoch. A user-space program can read the system time as follows:

```
#include <sys/time.h>
...
struct timeval theTime;
...
gettimeofday(&theTime, NULL);
...
```

Where the **timeval** structure is defined by:

```
struct timeval {
  long tv_sec;
  long tv_usec;
};
```

When the code fragment completes, the **timeval** structure will have the variable **theTime.tv_sec** set to the (long) integer number of seconds that have passed since 12 a.m. January 1, 1970. Another variable, **theTime.tv_usec**, is also of type long and provides the number of microseconds that have elapsed since the last second began. The date command translates the kernel time to the local time and to the Gregorian calendar time epoch before it produces a result on stdout.

For the **tv_usec** value to be correct at all times, it must be changed exactly once every microsecond (a millionth of a second). All modern computers use the same basic approach for keeping track of time. The hardware includes a programmable timer device that can be set to issue an interrupt every K time units; in the case of Linux machines, K is 4 milliseconds.

The system can track the passage of time by counting the number of interrupts that have occurred since the system was booted. If the time that the machine was last booted is known, then the current time (more accurately, the time that has elapsed since the machine was booted) can be computed to within 4 milliseconds. In i386 machines, a time-of-day clock runs whether or not the machine is powered up (machines usually have a battery backup that provides power for this clock). The time-of-day clock is not very accurate, but it establishes a base that can be used in conjunction with the 4 millisecond clock.

Per-Process Timers

The kernel accumulates time and manages various timers for each process. For example, the scheduling strategy depends on each process's having a record of the amount of CPU time that it has accrued since it last acquired the CPU. Because these time values are associated with each process, they are saved in the process's descriptor. When the kernel creates a task, it allocates a new task descriptor of type 'struct task_struct' from the kernel's heap space (i.e., it uses the kernel 'kmalloc()' call). This structure contains more than 75 fields. Next, you will consider the ones related to time values.

```
Struct task_struct {
      ...
      long counter;
      ...
      unsigned long policy, rt_priority;
      unsigned long it_real_value, it_prof_value, it_virt_value;
      unsigned long it_real_incr, it_prof_incr, it_virt_incr;
      struct timer_list real_timer;
      //contains tms_utime, tms_stime, tms_cutime, and tms_cstime
      struct tms;
      unsigned long start_time;
      long per_cpu_time[NR_CPUS], per_cpu_stime[], cutime, cstime;
      ...
};
```

The counter field is used to determine whether the process needs scheduling attention. The interval timers (**`it_XXX_value`** / **`it_XXX_incr`**) use the kernel time to keep track of the following three intervals of time relevant to every process.

- **`ITIMER_REAL`**: Reflects the passage of real time and is implemented by using the it_real_value and it_real_incr fields.

- **`ITIMER_VIRTUAL`**: Reflects the passage of virtual time. This time is incremented only when the corresponding process is executing.

- **`ITIMBERE_PROF`**: Reflects the passage of time during which the process is active (virtual time) plus the time that the kernel is doing work on behalf of the corresponding process (for example, reading a timer).

Each of these timers is actually a countdown timer. That is, it is periodically initialized to some prescribed value and then reflects the passage of time by counting down toward zero. When the timer reaches zero, it raises a signal to notify another part of the system (in the OS or a user-space program) that the counter has reached zero. Then it resets the value and begins counting down again.

Each timer is initialized with the **setitimer()** system call.

```
#include  <sys/time.h>
...
setitimer(
      int timerType,
      const struct timerval*value,
      struct itimerval *oldValue
);
```

The struct **itimerval** includes the following fields.

```
struct itimerval {
      struct timeval it_interval;
      struct timeval it_value;
);
```

To learn more about the parameters, read the man page for **setitimer()**. The general idea is that the **timerType** parameter is set to one of **ITIMER_REAL**, **ITIMER_VIRTUAL**, or **ITIMER_PROF**, which are constants defined in the sys/time.h header file. The **value** parameter is used to initialize second and microsecond fields of the given timer. The **it_value** field defines the current value for the time. The **it_interval** field defines the value that should be used to reset the timer when it reaches zero.

A timer is read with the **getitimer()** system call, as follows. In this case, the value parameter is used to return the value of the kernel's clock.

```
#include  <sys/time.h>
...
getitimer(
      int timerType,
      const struct timerval *value,
);
```

The following code fragment sets **ITIMER_REAL** and then reads it.

```
#include  <sys/time.h>
...
struct itimerval v;
...
v.it_interval.tv_sec = 9;
v.it_interval.tv_usec = 999999;
v.it_value.tv_sec = 9;
v.it_value.tv_usec = 999999;
setitimer(ITIMER_REAL, &&v, NULL);
...
getitimer(ITIMER_REAL, &v);
printf(“...%ld seconds, %ld microseconds ...”,
           ...,
           v.it_value.tv_sec,
           v.it_value.tv_usec,
           ...
      );
...
```

When **`ITIMER_REAL`** reaches zero, it is reset to (9, 999999) again. However, this code segment does not define any other processing that could be done when the corresponding signal is raised.

Procedure

1. Use **`ITIMER_REAL`** to implement our version of **`gettimerofday()`**. Set it so that it raises a signal once per second. Use the signal facility to determine when **`ITIMER_REAL`** has been decremented to zero and to count the number of seconds that have elapsed.

2. Design and implement facilities that use the **`ITIMER_VIRTUAL`** and **`ITIMER_PROF`** interval timers to profile a process. The profile should provide actual time of execution (by using the time from step 1), a CPU time (the time that the process is actually running), user-space time, and kernel-space time. Your program should use **`ITIMER_VIRTUAL`** and **`ITIMER_PROF`** to report CPU time, time spent in kernel space, and time spend in user space.

   All times should have millisecond accuracy, to the extent that your hardware can actually support this. That is, code it for millisecond accuracy, even though you might not get this level of accuracy in reality. Use the signal facility to create signal handlers to keep track of the number of seconds of virtual and profile time (raise a signal once per second).

3. Write a program to spawn two children, with each child recursively computing a Fibonacci sequence for N = 30, 34, and 40. The code skeleton for this program is provided below. Note **`Fibonacci(40)`** might take a couple of minutes to compute. Use the facilities that you implemented in steps 1 and 2 to determine the real time, virtual time, and profile time for each of the three processes.

Code Skeleton for Procedure Step 3 (Fibonacci Program)

As with all programs, you have many different ways to organize your solution. This section provides a skeleton of a solution for step 3. The skeleton uses signals to notify the user processes about time values. Your program must incorporate your own signal handler routines.

Skeleton for Fibonacci Program

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

long unsigned int fibonacci(unsigned int n);

static long p_realt_secs = 0, c1_realt_secs = 0, c2_realt_secs = 0;
static long p_virtt_secs = 0, c1_virtt_secs = 0, c2_virtt_secs = 0;
static long p_proft_secs = 0, c1_proft_secs = 0, c2_proft_secs = 0;
static struct itimerval p_realt, c1_realt, c2_realt;
static struct itimerval p_virtt, c1_virtt, c2_virtt;
static struct itimerval p_proft, c1_proft, c2_proft;

main(int argc, char**argv) {

      long unsigned fib = 0;
      int pid1, pid2;
      unsigned int fibarg;
      int status;

      // Get command line arugment, fibarg
      ...
      // Initialize parent, cild1, and child 2 timer values
      ...
      // Enable your signal handlers for the parent
      signal(SIGALRM, ...);
      signal(SIGVTALRM, ...);
      signal(SIGPROF, ...);

      // Set the parent's itimers
      ...
      pid1 = fork();

      if(pid1 == 0) {
            // Enable child 1 signal handlers (disable parent handlers)

            // Set the child 1 itimers

            // Start child 1 on the Fibonacci program
            fib = fibonacci)fibarg);

            // Read the child 1 itimer values, and report them
            getitimer(ITIMER_PROF, ...);
            getitimer(ITIMER_REAL, ...);
            getitimer(ITIMER_VIRTUAL, ...);
            printf("\n");
            printf("Child 1 fib = %ld, real time = %ld sec, %ld msec\n",
                        fib, c1_realt_secs,
                        elasped_usecs(c1_realt.it_value.tv_sec,
                                    c1_realt.it_value.tv_usec) / 1000);
            printf("Child 1 fib = %ld, cpu time = %ld sec, %ld msec\n",
                        fib, c1_proft_secs,
                        elapsed_usecs(c1_proft.it_value.tv_sec,
                                    c1_proft.it_value.tv_usec) / 1000);
            printf("Child 1 fib = =%ld, user time = %ld sec, %ld msec\n",
                        fib, c1_virtt_secs,
                        elapsed_usecs(c1_virtt.it_value.tv_sec,
                                    c1_virtt.it_value.tv_usec) / 1000);
            printf("Child 1 ib  = %ld, kernel time = %ld sec, %ld msec\n",
                        fib, c1_proft_secs – c1_virtt_secs,
                        (elaspsed_usecs(c1_proft.it_value.tv_sec,
                                    c1_proft.it_value.tv_usec) / 1000) -
                                    (elapsed_usecs(c1-virtt.it_value.tv_sec,
                                          c1_virtt.it_value.tv_usec) / 1000));
            fflush(stdout);
            exit(0);
      }
      else {
            pid2 = fork();
            if(pid2 == 0) {
                  // Enable the child 2 signal handlers (disable parent handlers)
                  ...
                  // Set the child 2 itimers
                  ...
                  // Start the child 2 on the Fibonacci program
                  fib = fibonacci(fibarg);
                  // Read the child 2 itimer values and report them
                  ...
            }
            else { /* This is the parent */
                  //Start the parent on the Fibonacci program
                  fib = fibonacci(fibarg);

                  // Wait for the children to terminate
                  waitpid(0, &status, 0);
                  waitpid(0, &status, 0);

                  // Read the parent itimer values, and report them
                  ...
            }
            printf("this line should never be printed\n");
      }
}

long unsigned int fibonacci(unsigned int n) {
      if(n ==0)
            return 0;
      else if (n == 1 || n ==2)
            return 1;
      else
            return (fibonacci(n-1_ + fibonacci(n-2))
}
```

Deliverables

All source, header, and make files.