# Lab 6: Signal and Timer

Yu-Lun Huang

2011-07-20

## 1 Objective

- Be familiar with signal, timer and process reaper.

## 2 Prerequisite

- Read man pages of `kill()`, `sigaction()`, `setitimer()`, etc.

## 3 Signal

Signals notify tasks of events that occurred during the execution of other tasks or Interrupt Service Routines (ISRs) . It diverts the notified task from its normal execution path and triggers the associated signal handler. Signal handler is a function run in "asynchronous mode", which gets called when the task receives that signal, no matter which code the task is executed on. This just like the relationship between hardware interrupts and ISRs, which are also asynchronous to the execution of OS and do not occur at any predetermined point of time. The difference between a signal and a normal interrupt is that signals are software interrupts, which are generated by other task or OS to trap normal execution of task. Another difference is that task can only receive a signal when it is running in user mode. If the receiving process is running in kernel mode, the execution of the signal will start only after the process returns to user mode. When the signal handler completes, the normal execution resumes. The task continues execution from wherever it happened to be before the signal was received.

Signals have been used for approximately 30 years without any major modifications. Although we now have advanced synchronization tools and many IPC mechanisms, signals play a vital role in Linux for handling exceptions and interrupts. For example, when child process exits its execution, it sends a `SIGCHLD` signal to parent process and becomes a zombie process. When the parent process catches this signal, it performs `wait()` or `waitpid()` to reclaim the resources used by child process in the signal handler. This design prevents the parent process from blocked in `wait()` or `waitpid()` function calls. (We detail this implementation in later section.)

### 3.1 Sending Signals

Linux supports various types of signal, you should specify the desired type when sending signal to other tasks. Each signal in Linux is identified by an integer value, which is called signal number or vector number. The first 31 signals are standard signals, some of which date back to 1970s UNIX from Bell Labs. The POSIX (Portable Operating Systems and Interface for UNIX) standard introduced a new class of signals designated as real-time signals, with numbers ranging from 32 to 63. Usually, all signal numbers as well as the associated symbolic name are defined in `/usr/include/signal.h`, or you can use the command `'kill -l'` to see a list of signals supported by your Linux system.

Signals can be generated from within the shell using the kill command. (Man the command "kill" to obtain detailed information.) The name of `kill` may seem strange, but actually most signals serve the purpose of terminating processes, so this is not really that unusual.

For example, the following command sends the SIGUSR1 signal to process 3423.

```
SHELL> kill -USR1 3423
```

Another method is to use the kill system call within a C program to send a signal to a process. This call takes a process ID and a signal number as parameters.

```
int kill(pid_t pid, int sig_no);
```

One common use of this mechanism is to end another process by sending it a SIGTERM or SIGKILL signal. Another common use is to send a command to a running program. Two "userdefined" signals are reserved for this purpose: SIGUSR1 and SIGUSR2. The SIGHUP signal is sometimes used for this purpose as well, commonly to wake up an idling program or cause a program to reread its configuration files.

## 3.2 Catching Signals

When a process receives a signal, it may do one of several things, depending on the signal's disposition. For each signal, there is a default disposition, which determines what happens to the task if the program does not specify any signal handler. If a signal handler is used, the currently executing task is paused, the signal handler is executed; and when the signal handler returns, the task resumes.

For most signal types, a program can specify some other behavior – either to ignore the signal or to call a special signal handler function to respond to the signal. We can use `sigaction` system call to register the signal handler or set a signal disposition.

The syntax of `sigaction` is:

```
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact)
```

The first argument, `signum`, is a specified signal. The next two parameters are pointers to `sigaction` structures; the second argument, is used to set the new disposition of the signal `signum`; and the third argument is used to store the previous disposition, usually NULL.

The `sigaction` structure is defined as:

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

The members of the `sigaction` structure are described as follows.

- `sa_hander`:
  `sa_hander` is a pointer pointed to a user-defined signal handler or default signal handler:
  - SIG_DFL, which specifies the default disposition for the signal.
  - SIG_IGN, which specifies that the signal should be ignored.
  - A pointer to a signal-handler function. This function receives the signal number as its only argument

- sa_mask:
  sa_mask gives a mask of signals which should be blocked during execution of the signal handler. The sigset_t structure consists an array of unsigned long, each bit inside controls the block state of a particular signal type. In the the first unsigned long in sigset_$t$, the least significant bit (0) is unused since no signal has the number 0, the other 31 bits represents the 31 standard UNIX signals The bits in the second unsigned long are the real-time signal numbers from 32 to 64.

- sa_flags:
  sa_flags specifies the action of signal. Sets of flags are available for controlling the signal in a different manner. More than one flag can be used by OR operation:

  - SA_NOCLDWAIT:
    If signum is SIGCHLD, do not transform child processes into zombies when they terminate.
  - SA_RESETHAND:
    SA_RESETHAND restores the default action of the signal after the user-defined signal handler has been executed.
  - SA_NODEFER:
    The signal which triggered the handler will be blocked. Set the SA_NODEFER allows signal can be received from within its own signal handler.
  - SA_SIGINFO:
    When SA_SIGINFO is set, the sa_sigaction should be used, instead of specifying the signal handler in sa_handler.

  For more flags information, please refer the sigaction manpages.

- sa_sigaction:
  If the SA_SIGINFO flag is set in sa_flags, sa_sigaction should be used. sa_sigaction is a pointer to a function that takes three arguments, for example:

```
void my_handler (int signo, siginfo_t *info, void *context);
```

  Here, signo is the signal number, and info is a pointer to the structure of type siginfo_t, which specifies the signal-related information; and context is a pointer to an object of type ucontext_t, which refers to the receiving process context that was interrupted with the delivered signal. For the detail structure of siginfo_t and ucontext_t, please refer the manpage of sigaction and getcontext.

  The following example uses SA_SIGINFO and sa_sigaction to extract information from a signal. Use the kill command to send a SIGUSR1 signal to catch program.

```
1  /*
2   * sig_catch.c
3   */
4
5  #include <signal.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10
11 void handler (int signo, siginfo_t *info, void *context)
12 {
13    /* show the process ID sent signal */
```

```
14      printf ("Process (%d) sent SIGUSR1.n", info->si_pid);
15  }
16
17  int main (int argc, char *argv[])
18  {
19      struct sigaction my_action;
20
21      /* register handler to SIGUSR1 */
22      memset(&my_action, 0, sizeof (struct sigaction));
23      my_action.sa_flags = SA_SIGINFO;
24      my_action.sa_sigaction = handler;
25
26      sigaction(SIGUSR1, &my_action, NULL);
27
28      printf("Process (%d) is catching SIGUSR1 ...\n", getpid());
29      sleep(10);
30      printf("Done.\n");
31
32      return 0;
33  }
```

Remember that some OSs implement `sa_handler` and `sa_sigaction` as a union, do not assign values to these two arguments at same time.

### 3.3 Atomic Access in Signal Handler

Assigning a value to a global variable can be dangerous because the assignment may actually be carried out in two or more machine instructions, and a second signal may occur between them, leaving the variable in a corrupted state. If you use a global variable to flag a signal from a signal handler function, it should be of the special type `sig_atomic_t`. In Linux, `sig_atomic_t` is an ordinary integer data type, but which one it is, and how many bits it contains, may vary from machine to machine. In practice, you can also use `sig_atomic_t` as a pointer. If you want to write a program which is portable to any standard UNIX system, though, use `sig_atomic_t` for these global variables. Reading and writing `sig_atomic_t` is guaranteed to happen in a single instruction, so there's no way for a handler to run "in the middle" of an access.

The following program listing uses a signal-handler function to count the number of times that the program receives SIGUSR1, one of the signals reserved for application use.

```
1   /*
2    * sig_count.c
3    */
4
5   #include <signal.h>
6   #include <stdio.h>
7   #include <string.h>
8   #include <sys/types.h>
9   #include <time.h>
10  #include <unistd.h>
11
12  sig_atomic_t sigusr1_count = 0;
13
14  void handler (int signal_number)
```

```
15 {
16    ++sigusr1_count;  /* add one, protected atomic action */
17 }
18
19 int main ()
20 {
21    struct sigaction sa;
22    struct timespec req;
23    int retval;
24
25    /* set the sleep time to 10 sec */
26    memset(&req, 0, sizeof(struct timespec));
27    req.tv_sec = 10;
28    req.tv_nsec = 0;
29
30    /* register handler to SIGUSR1 */
31    memset(&sa, 0, sizeof (sa));
32    sa.sa_handler = handler;
33
34    sigaction (SIGUSR1, &sa, NULL);
35
36    printf("Process (%d) is catching SIGUSR1 ...\n", getpid());
37
38    /* sleep 10 sec */
39    do{
40       retval = nanosleep(&req, &req);
41    } while(retval);
42
43    printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
44
45    return 0;
46 }
```

# 4 Timer

The timer schedules an event according to a predefined time value in the future. When the timer expired, it deliveries a signal to the calling process. Timers are used everywhere in Unix-like systems, from basic delay function implementation to network transmission and performance monitoring. Linux provides two basic POSIX standard timer functions, `alarm` and `setitimer`. The `alarm` system call arranges an alarm clock for delivering a `SIGALRM` signal after the specified time elapsed. The `setitimer` system call is a generalization of the alarm call, it provides three different types of timer for counting the elapsed time. The syntax of `setitimer` is shown as following:

```
int setitimer(int which, const struct itimerval *new_val, struct itimerval *old_val);
```

The first argument `which` is the timer code, specifying which timer to set.

- If the timer code is `ITIMER_REAL`, the process is sent a `SIGALRM` signal after the specified wall-clock time has elapsed.
- If the timer code is `ITIMER_VIRTUAL`, the process is sent a `SIGVTALRM` signal after the process has

executed for the specified time. Time in which the process is not executing (that is, when the kernel or another process is running) is not counted.

- If the timer code is `ITIMER_PROF`, the process is sent a `SIGPROF` signal when the specified time has elapsed either during the process's own execution or the execution of a system call on behalf of the process.

The second argument is a pointer to a `itimerval` structure specifying the new settings for that timer. The third argument, if not null, is a pointer to another `itimerval` structure which receives the old timer settings. The struct `itimerval` variable has two fields:

- `it_value` is a struct `timeval` field that contains the time until the timer next expires and a signal is sent. If this is 0, the timer is disabled.
- `it_interval` is another struct `timeval` field containing the value to which the timer will be reset after it expires. If this is 0, the timer will be disabled after it expires. If this is nonzero, the timer is set to expire repeatedly after this interval.

And the struct `timeval` has the following two members:

- `tv_sec` represents the number of whole seconds of elapsed time.
- `tv_usec` is the rest of the elapsed time (a fraction of a second), represented as the number of microseconds. It is always less than one million.

The following program illustrates the use of `setitimer` to track the execution time of a program. A timer is configured to be expired every 250 milliseconds and send a `SIGVTALRM` signal.

```
1  /*
2   * timer.c
3   */
4
5  #include <signal.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <sys/time.h>
9
10 void timer_handler (int signum)
11 {
12     static int count = 0;
13
14     printf ("timer expired %d times\n", ++count);
15 }
16
17 int main (int argc, char **argv)
18 {
19     struct sigaction sa;
20     struct itimerval timer;
21
22     /* Install timer_handler as the signal handler for SIGVTALRM */
23     memset (&sa, 0, sizeof (sa));
24     sa.sa_handler = &timer_handler;
25     sigaction (SIGVTALRM, &sa, NULL);
26
27     /* Configure the timer to expire after 250 msec */
```

```
28      timer.it_value.tv_sec = 0;
29      timer.it_value.tv_usec = 250000;
30
31      /* Reset the timer back to 250 msec after expired */
32      timer.it_interval.tv_sec = 0;
33      timer.it_interval.tv_usec = 250000;
34
35      /* Start a virtual timer */
36      setitimer (ITIMER_VIRTUAL, &timer, NULL);
37
38      /* Do busy work */
39      while (1);
40
41      return 0;
42  }
```

The following program demonstrates the difference among three different timers. Write a simple "while(1) loop" program first and execute several instances of this program as the number of cores in your machine. For example, if you have a dual core machine, then run two "while(1) loop" programs simultaneously. After occupy all computing resources, execute the `timer2.c` program and observe the results of three counters.

```
1   /*
2    * timer_diff.c
3    */
4
5   #include <fcntl.h>
6   #include <signal.h>
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <string.h>
10  #include <sys/stat.h>
11  #include <sys/time.h>
12  #include <sys/types.h>
13  #include <unistd.h>
14
15  /* counter */
16  int SIGALRM_count = 0;
17  int SIGVTALRM_count = 0;
18  int SIGPROF_count = 0;
19
20  /* handler of SIGALRM */
21  void SIGALRM_handler (int signum)
22  {
23      SIGALRM_count++;
24  }
25
26  /* handler of SIGVTALRM */
27  void SIGVTALRM_handler (int signum)
28  {
29      SIGVTALRM_count++;
30  }
31
```

```
32 /* handler of SIGPROF */
33 void SIGPROF_handler (int signum)
34 {
35    SIGPROF_count++;
36 }
37
38 void IO_WORKS();
39
40 int main (int argc, char **argv)
41 {
42    struct sigaction SA_SIGALRM, SA_SIGVTALRM, SA_SIGPROF;
43    struct itimerval timer;
44
45    /* Install SIGALRM_handler as the signal handler for SIGALRM */
46    memset (&SA_SIGALRM, 0, sizeof (SA_SIGALRM));
47    SA_SIGALRM.sa_handler = &SIGALRM_handler;
48    sigaction (SIGALRM, &SA_SIGALRM, NULL);
49
50    /* Install SIGVTALRM_handler as the signal handler for SIGVTALRM */
51    memset (&SA_SIGVTALRM, 0, sizeof (SA_SIGVTALRM));
52    SA_SIGVTALRM.sa_handler = &SIGVTALRM_handler;
53    sigaction (SIGVTALRM, &SA_SIGVTALRM, NULL);
54
55    /* Install SIGPROF_handler as the signal handler for SIGPROF */
56    memset (&SA_SIGPROF, 0, sizeof (SA_SIGPROF));
57    SA_SIGPROF.sa_handler = &SIGPROF_handler;
58    sigaction (SIGPROF, &SA_SIGPROF, NULL);
59
60    /* Configure the timer to expire after 100 msec */
61    timer.it_value.tv_sec = 0;
62    timer.it_value.tv_usec = 100000;
63
64    /* Reset the timer back to 100 msec after expired */
65    timer.it_interval.tv_sec = 0;
66    timer.it_interval.tv_usec = 100000;
67
68    /* Start timer */
69    setitimer(ITIMER_REAL, &timer, NULL);
70    setitimer(ITIMER_VIRTUAL, &timer, NULL);
71    setitimer(ITIMER_PROF, &timer, NULL);
72
73    /* Do some I/O operations */
74    IO_WORKS();
75
76    printf("SIGALRM_count___=_%d\n", SIGALRM_count);
77    printf("SIGVTALRM_count_=_%d\n", SIGVTALRM_count);
78    printf("SIGPROF_count___=_%d\n", SIGPROF_count);
79
80    return 0;
81 }
82
```

```
83  void IO_WORKS()
84  {
85      int fd, ret;
86      char buffer[100];
87      int i;
88
89      /* Open/Read/Close file 300000 times */
90      for (i = 0; i < 300000; i++) {
91          if ((fd = open("/etc/init.d/networking", O_RDONLY)) < 0) {
92              perror("Open /etc/init.d/networking");
93              exit(EXIT_FAILURE);
94          }
95
96          do {
97              ret = read(fd, buffer, 100);
98          }while(ret);
99
100         close(fd);
101     }
102 }
```

# 5 Process Reaper

When a child process completes its execution, rather than reclaims all memory resources associated with it, the OS puts this process into a zombie state and allows the parent process read the exit status of child process. In Lab 3, we let the parent process executes the `wait` and `waitpid` system call to 'reap" (remove) the zombie process. But executing `wait` and `waitpid` blocks the normal execution of parent process. Even configures the `waitpid` with `WNOHANG` option, the parent should perform periodic checking on status of child processes.

One solution is let parent process explicitly ignores `SIGCHLD` by setting its handler to `SIG_IGN` (rather than simply ignoring the signal by default) or has the `SA_NOCLDWAIT` flag set, all child exit status information will be discarded and no zombie processes will be left. But sometimes, we need to perform specific operation when child process end its execution, e.g. check the exit status of child processes. In such case, we register the signal handler with `SIGCHLD` to reap the child process manually, it is also known as the `process reaper`.

The following program illustrates the basic operation of `process reaper`. This program uses another system call `siganl` to register the `reaper` function for `SIGCHLD`. (Please refer the manpage to see the difference between `siganl` and `sigaction`.) When the parent process traps into the `reaper` function, it performs a nonblocking `waitpid` call to reclaim zombie processes. The `reaper` function checks the return value of `waitpid` to determine any zombie process to reap. If the return value is equal to zero, which indicates no zombie exists, we exit the handler to resume normal execution.

```
1   /*
2    * reaper.c
3    * Demonstrate the work of process reaper
4    */
5
6   #include <signal.h>
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <sys/types.h>
10  #include <sys/wait.h>
```

```
11 #include <time.h>
12 #include <unistd.h>
13
14 #define FORKCHILD 5
15
16 volatile sig_atomic_t reaper_count = 0;
17
18 /* signal handler for SIGCHLD */
19 void Reaper(int sig)
20 {
21     pid_t pid;
22     int status;
23
24     while((pid = waitpid(-1, &status, WNOHANG)) > 0)
25     {
26         printf("Child %d is terminated.\n", pid);
27         reaper_count++;
28     }
29 }
30
31 void ChildProcess()
32 {
33     int rand;
34
35     /* rand a sleep time */
36     srand(time(NULL));
37     rand = random() % 10 + 2;
38
39     printf("Child %d sleep %d sec.\n", getpid(), rand);
40     sleep(rand);
41     printf("Child %d exit.\n", getpid());
42
43     exit(0);
44 }
45
46 int main(int argc, char *argv[])
47 {
48     int cpid;
49     int i;
50
51     /* regist signal handler */
52     signal(SIGCHLD, Reaper);
53
54
55     /* fork child processes */
56     for (i = 0; i < FORKCHILD; i++)
57     {
58         if ( (cpid = fork()) > 0)  /* parent */
59             printf("Parent fork child process %d.\n", cpid);
60         else  /* child */
61             ChildProcess();
```

```
62
63          sleep(1);
64      }
65
66      /* wait all child exit */
67      while(reaper_count != FORKCHILD)
68          sleep(1);
69
70      return 0;
71  }
```

# 6 PXA I/O

- Keypad I/O
  - Header files: linux-2.4.x/include/asm-armnommu/arch-creator/lib/creator_s3c4510_lcd.h
  - Function: ioctl(fd, command, data)
    - Command:
      KEY_IOCTL_GET_CHAR: unsigned short, get its ASCII value.
      KEY_IOCTL_WAIT_CHAR: wait until get a character.
      KEY_IOCTL_CHECK_EMPTY
      KEY_IOCTL_CLEAR
      KEY_IOCTL_CANCEL_WAIT_CHAR

    - Definition

```
1  #define VK_S2  1     /* ASCII = '1' */
2  #define VK_S3  2     /* ASCII = '2' */
3  #define VK_S4  3     /* ASCII = '3' */
4  #define VK_S5  10    /* ASCII = 'A' */
5  #define VK_S6  4
6  #define VK_S7  5
7  #define VK_S8  6
8  #define VK_S9  11
9  #define VK_S10 7
10 #define VK_S11 8
11 #define VK_S12 9
12 #define VK_S13 12
13 #define VK_S14 14    /* ASCII = '*' */
14 #define VK_S15 0
15 #define VK_S16 15    /* ASCII = '#' */
16 #define VK_S17 13
```

    - Sample

```
1  unsigned short     key;
2  int           fd, ret;
3
4  if ((fd = open("/dev/lcd", O_RDWR)) < 0)  return (-1);
5
6  ioctl(fd, KEY_IOCTL_CLEAR, key);
```

```
7  while(1) {
8    ret = ioctl(fd, KEY_IOCTL_CHECK_EMPTY, &key)
9    if (ret < 0) {
10     sleep(1);
11     continue;
12   }
13   ret = ioctl(fd, KEY_IOCTL_GET_CHAR, &key)
14   if (key & 0xff) == '#') break;    /* terminate */
15 }
16 close(fd);
```

# Lab 6 Assignments

· Write a program that implements a stop timer up to the 0.1 second accuracy. Use the 7-segment LEDs for the clock display. (You should display second, 1/10 second and decimal point.) Use the keypad keys as the control: '∗' means "start", '#' means "pause", and '0' means "reset". The timer cannot be reset during counting. Explain your design to TAs.