CSE 2431 **LAB 2** **Autumn 2018**

Due: Friday, Sep. 28, 11:59 p.m.
5 pts

The source code and Makefile must be packed as a single zip file and submitted electronically online in Carmen Dropbox by 11:59 pm Friday 09/28/18. Late penalty will be 25% per day. Please complete your solution in the virtual machine you installed in Lab 0. The grader will compile and test your solution under the same environment. Any program that does not compile will receive a zero. The grader will NOT spend any time fixing your code. It is your responsibility to leave yourself enough time to ensure that your code can be compiled, run, and tested well before the due date.

1. Goal

Understanding the Linux *signal* mechanisms.

2. Introduction

2.1. What are signals?

Signal is a notification, a message sent by either operating system or some applications to a process (or to one of its threads).

Each signal is identified by a number, from 1 to 31. Signals don't carry any argument and their names are mostly self explanatory. For instance, SIGKILL (signal number 9) tells the program that someone tries to kill it.

Signals is one type of Inter Process Communication (IPC) mechanisms. It has several different usages. For instance, debuggers rely on signals to receive events about programs that are being debugged. A signal can also be used by a program to handle the exceptions generated by itself.

2.2. Types of signals

- **SIGHUP**

This signal indicates that someone has killed the controlling terminal. When someone kills the terminal window, without killing applications running inside the terminal, the operating system sends SIGHUP to the program. Default handler for this signal will terminate the program.

- **SIGINT**

This is the signal that being sent to an application when it is running in a foreground in a terminal and someone presses CTRL-C. Default handler of this signal will quietly terminate the program.

- **SIGILL**

Illegal instruction signal. This is an exception signal, sent to the application by the operating system when the CPU encounters an illegal instruction inside the program.

- **SIGABRT**

Abort signal means the program used abort() API inside of the program. It is yet another method to terminate a program.

- **SIGFPE**

Floating point exception. This is another exception signal, issued by operating system when the application caused an exception.

- **SIGSEGV**

This is an exception signal as well. Operating system sends a program this signal when it tries to access memory that does not belong to it.

- **SIGPIPE**

Broken pipe. As documentation states, this signal sent to a program when it tries to write into pipe (another IPC) with no readers on the other side.

- **SIGALRM**

Alarm signal. Sent to a program using alarm() system call. The alarm() system call is basically a timer that allows a program to receive SIGALRM in preconfigured number of seconds. This can be handy, although there are more accurate timer API out there.

- **SIGTERM**

This signal tells a program to terminate itself. Consider this as a signal to cleanly shut down while SIGKILL is an abnormal termination signal.

- **SIGCHLD**

Tells the process that a child process of the program has stopped or terminated. This is handy when you wish to synchronize a process with its child process.

- **SIGUSR1 and SIGUSR2**

SIGUSR1 and SIGUSR2 are two signals that have no predefined meaning and are left for the developer's consideration.

2.3. Registering signal handler

Many of these signals have their default handler. But there are several interfaces that allow you to register your own signal handler.

sigaction() is a system call that manipulates signal handler.

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

Its first argument specifies a signal number. Second and third arguments are pointers to structure called *sigaction*. This structure specifies how process should handle given signal.

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

sa_handler is a pointer to the signal handler routine. The routine accepts single integer number containing signal number that it handles and returns void. If needed, *sa_sigaction* pointer should point to the advanced signal handler routine. This one receives much more information about the origin of the signal.

To use *sa_sigaction* routine, make sure to set *SA_SIGINFO* flag in *sa_flags* member of struct *sigaction*. Similar to *sa_handler*, *sa_sigaction* receives an integer telling it what signal has been triggered. In addition, it receives a pointer to a structure called *siginfo_t*, which describes the origin of the signal. For instance, the *si_pid* member of *siginfo_t* holds the process ID of the process that has sent the signal. There are several other fields that tell you lots of useful information about the signal. You can find all the details on sigaction's manual page (*man sigaction*).

Last argument received by *sa_sigaction* handler is a pointer to *ucontext_t*. This type different from architecture to architecture.

One additional advantage of *sigaction*() is that it allows you to tell operating system what signals can be handled inside the signal you are registering. That is, it gives you full control over what signals can arrive, while your program handling another signal.

To tell this, you should manipulate *sa_mask* member of the *struct sigaction*. Note that is a *sigset_t* field. *sigset_t* type represents signal masks. To manipulate signal masks, use one of the following functions:

*int sigemptyset(sigset_t **) – to clear the mask.
*int sigfillset(sigset_t *)* – to set all bits in the mask.
*int sigaddset(sigset_t *, int signum)* – to set bit that represents certain signal.
*int sigdelset(sigset_t *, int signum*) – to clear bit that represents certain signal.
*int sigismember(sigset_t *, int signum)* – to check status of certain signal in a mask.

2.4. Examples

Let’s show a small program that demonstrates *sigaction()* in use.

```
/* receiver.c */
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static void hdl (int sig, siginfo_t *siginfo, void *context)
{
        printf ("Sending PID: %ld, UID: %ld\n",
                        (long)siginfo->si_pid, (long)siginfo->si_uid);
}

int main (int argc, char *argv[])
{
        struct sigaction act;

        memset (&act, '\0', sizeof(act));

        /* Use the sa_sigaction field because the handle has two additional parameters */
        act.sa_sigaction = &hdl;

        /* The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field, not sa_handler. */
        act.sa_flags = SA_SIGINFO;

        if (sigaction(SIGUSR1, &act, NULL) < 0) {
          perror ("sigaction");
          return 1;
         }

        printf("The PID is %d\n", getpid());

        while (1)
                sleep (10);

        return 0;
}

/* sender.c */
#include  <stdio.h>
#include  <signal.h>
#include <stdlib.h>

void main(int argc, void**argv)
```

```
{
    if (argc != 2) {
        printf("argument number incorrect!\n");
        exit(1);
    }

    int pid = atoi(argv[1]);
    kill(pid, SIGUSR1);
}
```

The source code is provided to you. You can run *receiver* first, which will print the PID in the terminal. Then run *./sender PID* to send the signal to the receiver process.

3. Assignment

Write a countdown clock, which reflects the number of days, hours, minutes, seconds before your final exam (of this course) starts. The clock should be refreshed after every 5 seconds. You should use the signal handler and the *alarm*() POSIX API. *The countdown clock needs to reflect the actual time, not a simulated value.* An example output is:

```
1 day 4 hours 8 minutes 45 seconds
1 day 4 hours 8 minutes 40 seconds
1 day 4 hours 8 minutes 35 seconds
1 day 4 hours 8 minutes 30 seconds
…
```

4. Extra credit (1pt)

Write a report to discuss how signals can be handled by a particular thread in a process. Write code that can compile and run to assist your discussion.

Hit: You don't need to continue with your code in this assignment. You can start a new program from scratch, if you feel it is easier. The purpose is to ask you to learn programming by yourself---and most importantly, to have a good understanding of what you've learned through practice and writing.