CSE410 Operating Systems Spring 2014
Laboratory 1: Introduction to Unix/Linux Signals
Due: 23:59 Tuesday, February 18, 2014

1 Overview and Background

In this lab assignment you will gain experience with Unix/Linux signals by applying them to a process monitoring application, including writing code to send signals to processes and code for handling signals. This lab assignment will also help you to gain basic knowledge about the `ps` (i.e., *process status*) command in Unix/Linux. You may write your solution in either C or C++. The skeleton code provided to you is written in C++ (you are not required to use it).

You are asked to implement a simple monitor program that periodically checks for certain processes and kills them if they are present. The first type of process is identified by its name, "spinner"; here we assume it has been previously identified as a malicious or otherwise undesirable process. The second type is a zombie process. The monitor periodically checks for these processes by executing the `ps` command. If the spinner is detected, it is killed. If a zombie is detected, its parent process is killed, as discussed below.

Your monitor program will use signals in multiple ways. So that you can focus primarily on the details of using signals, the code for obtaining process status information (executing the command and parsing the results) is provided to you as part of the skeleton code. [link] (Using the skeleton code is not mandatory.)

IMPORTANT NOTE: You should develop and test your code on machines in 3353 EB, **NOT** on the department servers.

1.1 Unix/Linux Signals [man page]

Signals are one of the principle operating system mechanisms for interprocess communication. A signal is a software mechanism for informing a process of the occurrence of an asynchronous event. Some signals (e.g., `SIGINT`, `SIGBUS`) are associated with hardware interrupts, while other signals are initiated from software. Unlike hardware interrupts, signals are not prioritized and have no concept of ordering.

Processes may send each other signals as a (limited) form of interprocess communication. The kernel may also send signals to processes (e.g., `SIGBUS`, `SIGSEGV`). To send a signal to a process A, the kernel updates a field of A's process control block. The signal is said to be "posted" against the target process. Each signal is represented as a single bit flag, and thus signals of the same type cannot be queued. When a process is scheduled for

execution, it checks these signal flags and performs corresponding actions for posted signals before returning to user level. The signals can be handled by predefined (default) actions (e.g., termination for `SIGTERM`), can be "caught" by user-defined signal handlers, or (in some cases) can be ignored by the process. User-defined signal handler functions execute at user level and in the context of the target process.

Some characteristics of signals and signal handlers are noteworthy and summarized as follows.

- Not every signal can have a user-defined signal handler. For example, the default action of a `SIGKILL` signal is terminating the process and cannot be overridden.
- To make a signal handler operational, it must be registered with the kernel by using the `signal()` or `sigaction()` system call. Please see the example below.
- A signal handler takes *exactly one* parameter, the signal number. Any additional arguments need to be passed to the signal handler through some other mechanisms.
- Signals can be sent from one process to another using the `kill()` system call. (Typically, the processes must be owned by the same user for the signal to take effect.)
- A process can also send signals to itself using `raise()` library routine, which invokes `kill()`.

The following is a simple example C program that handles the SIGALRM signal with a user-defined signal handler.

`alarm_example.c`

```
 1 #include<stdio.h>
 2 #include<signal.h>
 3 #include<unistd.h>
 4 // user-defined signal handler for alarm.
 5 void alarm_handler(int signo)
 6 {
 7     if (signo == SIGALRM)
 8     {
 9         printf("alarm goes off\n");
10     }
11 }
12
13 int main(void)
14 {
15     // register the signal handler
16     if (signal(SIGALRM, alarm_handler) == SIG_ERR){
```

```
17          printf("failed to register alarm handler.");
18          exit(1);
19      }
20
21      alarm(3);               // set alarm to fire in 3 seconds.
22      while(1){ sleep(10) }; // wait until alarm goes off
23
24 }
```

1.2 The ps Command [man page]

The ps command reports the status of a subset of active processes in the system. By default, ps reports on all processes owned by the user invoking the command. For example, the invocation of ps displays the following to the terminal.

```
ned:~ >ps
  PID TTY          TIME CMD
19414 pts/0    00:00:00 tcsh
27494 pts/0    00:00:00 ps
```

In this example, the user opened a terminal, logged into ned.cse.msu.edu and executed ps. The output is in four columns: the process identifier (PID), the terminal (TTY) associated with the process, the CPU time consumed by the process (TIME), and the name of the executable command (CMD).

A user can execute ps with different options to obtain other types of information and targeting other sets of processes. For example, ps aux reports on the processes owned by all users (option a), displays the username of the owner of each process (option u) and shows processes that are not attached to any terminal (option x). An example is shown below. (the ******** pattern is used here to mask actual usernames.)

```
ned:~ >ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
....
root     23788  0.0  0.0  81520  3648 ?        Ss   19:05   0:00 sshd: ******** [priv]
******** 23792  0.0  0.0  81520  1832 ?        S    19:05   0:00 sshd: ********@pts/6
******** 23793  0.0  0.0  18612  2804 pts/6    Ss+  19:05   0:00 -tcsh
root     25001  0.0  0.0      0     0 ?        S    Jan25   0:07 [kworker/5:0]
root     25245  0.0  0.0      0     0 ?        S    Jan23   0:14 [kworker/2:2]
root     26085  0.0  0.0  81512  3616 ?        Ss   20:11   0:00 sshd: ******** [priv]
******** 26090  0.0  0.0  81512  1828 ?        S    20:11   0:00 sshd: ********@pts/2
....
```

In this assignment, we will use `ps uf` to detect spinners and zombies. An invocation of `ps uf` displays information as follows.

```
ned:~ >ps uf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
******** 23793  0.0  0.0  18612  2804 pts/6    Ss+  19:05   0:00 -tcsh
******** 19414  0.0  0.0  18664  2864 pts/0    Ss   17:36   0:00 -tcsh
******** 28064  0.0  0.0  15300  1220 pts/0    R+   21:02   0:00  \_ ps uf
```

This user has two active sessions associated with two respective terminals, `pts/6` and `pts/0`. One of them (`pts/0`) is executing `ps uf`. The option `u` displays user-oriented format the the option `f` displays process hierarchy. In this project, we are concerned only with three columns of the output:

- `PID`: The process ID. The monitor needs to store the process ID properly, so that it can kill the process if necessary.
- `STAT`: The state flags of each process. The monitor checks if the process is labeled as a zombie (`Z` in the `STAT` column). To terminate the zombie, it kills the *parent* of the zombie.
- `COMMAND`: The name of the executable of the process.

Please refer to the man page of `ps` for more information about options and their semantics.

2 Requirements, Deliverables and Grading

You are required to implement a monitor program that periodically executes `ps uf` and displays the output to the terminal. The periodic execution must be implemented with the `SIGALRM` signal and corresponding handler. If any spinner or zombie processes are detected, the monitor should kill them by sending a `SIGKILL` signal to each spinner and the parent of each zombie. Whenever a zombie process is killed, the monitor raises `SIGUSR1` and displays a message indicating success of the action. Whenever a spinner process is killed, the monitor raises `SIGUSR2` and displays a similar message. You are given the code for the *spinner* program (`spinner.cpp`) and the code to create a *zombie* (`zombie.cpp`) as part of the skeleton code, described below.

Moreover, the `SIGINT` signal sends to the monitor program needs to be captured. A confirmation message needs to be shown to the terminal and monitor terminates only if the user confirms exit by entering y/Y. That is, when the user enters `SIGINT (ctrl + C)` and tries to terminate the monitor, the monitor should not be terminated without prompting the user.

Each Linux distribution might be slightly different. It is the students' responsibility to make sure that the lab submissions compile on *at least one* of the following machines in 3353 EB: `carl`, `ned`, `marge` or `skinner`. A statement must be provided in the README file's header. You will **not** be awarded any credit if your lab submission does not compile on any of those machines.

2.1 Project Guidelines

Working alone or in pairs. You may work on this assignment individually or in pairs (not in groups of 3 or more, however). If you prefer to work in a pair, **both** students must submit a copy of the solution and identify their MSU NetID in a README file. If you prefer to work individually, please clearly state that you are working individually and include your MSU NetID in the README file.

Programming Language. You may implement this project using `C` or `C++`. The skeleton code is written in `C++`. Please clearly state the command to compile your project submission in your README file.

Testing your code. Each Linux distribution might be slightly different. It is the students' responsibility to make sure that the lab submissions compile on *at least one* of the following machines in 3353 EB: `carl`, `ned`, `marge` or `skinner`. A statement must be provided in the README file's header. You will **not** be awarded any credit if your lab submission does not compile on any of those machines.

2.2 Deadline and Deliverables

This lab is due no later than 23:59 (11:59 PM) on Tuesday, February 18, 2014. No late submission will be accepted. You must submit your lab using the *handin* utility. (`https://secure.cse.msu.edu/handin/`) Your submission should include:

All source files. Submit all source files in your project directory. If you use the skeleton code, submit all the files, even if some files are not modified.

A makefile. Include a makefile to compile your code. A makefile is provided in the skeleton code, but you may choose to modify it.

A README file. The README file should include a header, sample output and any relevant comments. Please provide the following items in header of the README: whether you are working individually or in a pair, the MSU NetIDs of the submitting students, a list of machines on which you have compiled your code, the command used to compile your code. Two sample README file headers are as follows:

```
Student NetID: alice999, I am working with bob99999.
```

```
Compilation tested on: ned, skinner, marge ...
Command for compile: gcc proj1.c -o proj1


Student NetID: doejohnQ, I work on this project individually.
Compilation tested on: ned
Command for compile: make
```

Your README file should also include example output from your program, which will aid the TA in debugging if he cannot reproduce your results. You are also encouraged to include any comment in the README file. A sample README file is also included in the skeleton code.

2.3 Grading

This project is worth 50 points. You will not be awarded any points if your submission does not compile. The grading rubric is as follows:

```
General requirements: 3 points
_____  1 pts: Coding standard, comments ... etc
_____  1 pts: README file
_____  1 pts: Descriptive messages/outputs

Sending and Handling Signals: 37 points
_____  5 pts: Handling SIGINT correctly, confirm exit.
_____  5 pts: Kill the program if SIGINT is confirmed
_____ 12 pts: Sending and handling SIGALRM
          6 pts: Stop SIGALRM while handling SIGINT
                 Restart SIGALRM if n/N is entered
          6 pts: Send SIGALRM periodically to run and parse ps
_____  5 pts: Correctly kill spinners
_____  5 pts: Correctly kill zombies
______ 5 pts: Correctly use of SIGUSR1 and SIGUSR2

Error checking: 10 points
_____ 10 pts: Check the return values of signal handler registrations
```

3 Skeleton Code

To assist you in this assignment, skeleton code comprising six files is provided. The first is an example README file, as described earlier. The second is a makefile; executing the command `make` will generate the following executables: `zombie`, `spinner` and `proj1`. The remaining four files are described below. Please note that you are not required to use the skeleton code.

3.1 Zombie process (zombie.cpp)

A zombie process is a process that completes its execution before its parent process. The parent process has not yet read the zombie child process's exit status since the parent process is still executing. The zombie process still has an entry in the process table. The provided program forks a child process which terminates immediately, while the parent process is sleeping. Hence the child process becomes a zombie.

Zombies can be detected by examining if the `Z` flag exists in the `STAT` column in the output from `ps` command. The following is an execution result of `ps` command that shows a zombie.

```
ned:~ >ps uf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
******** 23793  0.0  0.0  18616  2808 pts/6    Ss   19:05   0:00 -tcsh
******** 30104  0.0  0.0  11744   980 pts/6    S+   22:10   0:00  \_ zombie
******** 30105  0.0  0.0      0     0 pts/6    Z+   22:10   0:00      \_ [zombie] <defun
******** 19414  0.0  0.0  18664  2864 pts/0    Ss   17:36   0:00 -tcsh
******** 30106  0.0  0.0  15300  1220 pts/0    R+   22:10   0:00  \_ ps uf
```

The user has two active sessions in two terminals connected to the system. One of them (`pts/0`) is executing `ps uf` and the other (`pts/6`) is executing the `zombie` program with one parent process (`PID 30104`) and one zombie child process (`PID 30105`).

Note that since a zombie is already dead, it cannot be killed. If one wishes to remove a zombie process, one will have to kill the zombie's parent process.

3.2 Spinner (spinner.cpp)

The spinner program is assumed to be malicious or unwanted based solely on its name. The code is located in `spinner.cpp`. When a spinner is running, `ps uf` displays the following information.

```
ned:~ >ps uf
```

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
******** 23793  0.0  0.0  18612  2804 pts/6    Ss   19:05   0:00 -tcsh
******** 28425  0.0  0.0  11748   736 pts/6    R+   21:19   0:06  \_ spinner
******** 19414  0.0  0.0  18664  2864 pts/0    Ss   17:36   0:00 -tcsh
******** 28428  0.0  0.0  15300  1220 pts/0    R+   21:19   0:00  \_ ps uf
```

This user has two active sessions in two terminals connected to the system. One of them (`pts/0`) is executing `ps uf` and the other (`pts/6`) is executing the `spinner` program (`PID 28425`).

`proj1.h` is a collection of support functions, including `run_ps`, `parse_ps`, `parse_argv` and `help_message`. `run_ps` and `parse_ps` are explained earlier. The function `parse_argv` is for parsing the argument to this program.

3.3 Support Functions (proj1.h)

This file contains global variables and several functions that will be of use to you. The most important functions are `run_ps` and `parse_ps`, for executing the `ps` command and parsing the output, respectively. The first function makes use of the `popen()` library routine (which uses fork() and exec()) to create a `ps` process and pipe the results back to your program. Specifically, the monitor needs to obtain the `PID`, `CPU`, `STAT` and `COMMAND` of each process. Unlike the related `system()` routine, the result will be stored in a normal standard I/O stream. We can then read the result from the I/O stream using `fget()`. The I/O stream must be closed by `pclose()`; Then, the function `parse_ps` is used to construct four vectors of information for use by the monitor. Please refer to `proj1.h` in the skeleton code for details.

3.4 Monitor Program (proj1.cpp)

This is the file you need to modify. You need to construct the monitor by doing the following:

- Add code for posting and handling signals. You need to register signal handlers with the kernel. Be sure you checked their return values of all system calls.
- Use the `SIGALRM` signal to periodically execute `run_ps` and `parse_ps`. Note that the interval value has been parsed by `parse_argv`.
- Capture `SIGINT` signals and stop/restart `SIGALRM` correctly. That is, when a SIGINT is caught, you should stop the timer. If the user decides to continue, restart the timer.

4 Examples

Follows are examples of output from the monitor program. Your output may differ, depending on how you execute spinners, zombies, the period of the monitor, and the mesages you print.

1. Just the monitor.

```
~ >proj1 -t 5
Checking activity every 5 seconds.
Checking processes...
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
liuchinj 30181  0.0  0.0  18148  2304 pts/4    Ss   22:14   0:00 -tcsh
liuchinj 32324  0.2  0.0  30492  3524 pts/4    S+   23:09   0:00  \_ vim proj1.cpp
liuchinj 23793  0.0  0.0  18624  2816 pts/6    Ss   19:05   0:00 -tcsh
liuchinj 32380  0.0  0.0  11912  1200 pts/6    S+   23:09   0:00  \_ proj1 -k 50 -t
liuchinj 32385  0.0  0.0   4176   576 pts/6    S+   23:09   0:00      \_ sh -c ps u
liuchinj 32386  0.0  0.0  15300  1224 pts/6    R+   23:09   0:00          \_ ps uf
liuchinj 19414  0.0  0.0  18664  2864 pts/0    Ss+  17:36   0:00 -tcsh
^CDo you really want to exit? (Y/N): y
Program terminates.
Killed
```

2. Just the monitor. Double check for `SIGINT`.

```
~ >proj1 -t 5
Checking activity every 5 seconds.
Checking processes...
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
liuchinj 30181  0.0  0.0  18148  2304 pts/4    Ss   22:14   0:00 -tcsh
liuchinj 32324  0.0  0.0  30492  3524 pts/4    S+   23:09   0:00  \_ vim proj1.cpp
liuchinj 23793  0.0  0.0  18628  2816 pts/6    Ss   19:05   0:00 -tcsh
liuchinj 32418  0.0  0.0  11912  1204 pts/6    S+   23:11   0:00  \_ proj1 -k 50 -t
liuchinj 32422  0.0  0.0   4176   580 pts/6    S+   23:11   0:00      \_ sh -c ps u
liuchinj 32423  0.0  0.0  15300  1224 pts/6    R+   23:11   0:00          \_ ps uf
liuchinj 19414  0.0  0.0  18664  2864 pts/0    Ss+  17:36   0:00 -tcsh
^CDo you really want to exit? (Y/N):
n
Program continues.

Checking processes...
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```

```
liuchinj 30181  0.0  0.0  18148  2304 pts/4    Ss   22:14   0:00 -tcsh
liuchinj 32324  0.0  0.0  30492  3524 pts/4    S+   23:09   0:00  \_ vim proj1.cpp
liuchinj 23793  0.0  0.0  18628  2816 pts/6    Ss   19:05   0:00 -tcsh
liuchinj 32418  0.0  0.0  11916  1224 pts/6    S+   23:11   0:00  \_ proj1 -k 50 -t
liuchinj 32437  0.0  0.0   4176   580 pts/6    S+   23:11   0:00      \_ sh -c ps u
liuchinj 32438  0.0  0.0  15300  1220 pts/6    R+   23:11   0:00          \_ ps uf
liuchinj 19414  0.0  0.0  18664  2864 pts/0    Ss+  17:36   0:00 -tcsh
```

3. The monitor detects a spinner and kills it.
   Run a spinner in another terminal first.

   ```
   ~ >spinner.
   ```

   In another terminal.

   ```
   ~ >proj1 -t 10
   Checking activity every 10 seconds.
   Checking processes...
   USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
   liuchinj 30181  0.0  0.0  18148  2304 pts/4    Ss   22:14   0:00 -tcsh
   liuchinj 32324  0.0  0.0  30492  3524 pts/4    S+   23:09   0:00  \_ vim proj1.cpp
   liuchinj 23793  0.0  0.0  18632  2820 pts/6    Ss   19:05   0:00 -tcsh
   liuchinj 32721  0.0  0.0  11912  1204 pts/6    S+   23:15   0:00  \_ proj1 -k 50 -t
   liuchinj 32725  0.0  0.0   4176   580 pts/6    S+   23:15   0:00      \_ sh -c ps u
   liuchinj 32726  0.0  0.0  15300  1224 pts/6    R+   23:15   0:00          \_ ps uf
   liuchinj 19414  0.0  0.0  18664  2864 pts/0    Ss   17:36   0:00 -tcsh
   liuchinj 32720  0.0  0.0  11748   736 pts/0    R+   23:15   0:07  \_ spinner
   -tcsh
    \_ spinner is consuming too much CPU time.
   Attempting to kill process 32720...
   Successfully killed spinner program.
   ```

4. The monitor detects a zombie and kills its parent. Run a zombie in another terminal first.

   ```
   ~ >zombie
   ```

   In another terminal.

   ```
   ~ >proj1 -t 3
   Checking activity every 3 seconds.
   Checking processes...
   ```

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
liuchinj 30181  0.0  0.0  18148  2304 pts/4    Ss   22:14   0:00 -tcsh
liuchinj 32324  0.0  0.0  30492  3524 pts/4    S+   23:09   0:00  \_ vim proj1.cpp
liuchinj 23793  0.0  0.0  18632  2820 pts/6    Ss   19:05   0:00 -tcsh
liuchinj 32721  0.0  0.0  11916  1260 pts/6    S+   23:15   0:00  \_ proj1 -k 50 -t
liuchinj   378  0.0  0.0   4176   580 pts/6    S+   23:17   0:00      \_ sh -c ps u
liuchinj   379  0.0  0.0  15300  1212 pts/6    R+   23:17   0:00          \_ ps uf
liuchinj 19414  0.0  0.0  18664  2864 pts/0    Ss   17:36   0:00 -tcsh
liuchinj   361  0.0  0.0  11744   980 pts/0    S+   23:16   0:00  \_ zombie
liuchinj   362  0.0  0.0      0     0 pts/0    Z+   23:16   0:00      \_ [zombie] <
zombie process found:
-tcsh
 \_ zombie
     \_ [zombie] <defunct>
Attempting to kill process 361...
Successfully killed zombie program.
```

5. The monitor detects both zombie and spinner and kills both of them. Run a zombie and a spinner first as shown before.

```
~ >proj1 -t 10
Checking activity every 10 seconds.
Checking processes...
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
liuchinj 30181  0.0  0.0  18148  2304 pts/4    Ss   22:14   0:00 -tcsh
liuchinj   610  0.0  0.0  11748   736 pts/4    R+   23:20   0:07  \_ spinner
liuchinj 23793  0.0  0.0  18632  2824 pts/6    Ss   19:05   0:00 -tcsh
liuchinj   582  0.0  0.0  11912  1204 pts/6    S+   23:20   0:00  \_ proj1 -k 50 -t
liuchinj   617  0.0  0.0   4176   580 pts/6    S+   23:20   0:00      \_ sh -c ps u
liuchinj   618  0.0  0.0  15300  1212 pts/6    R+   23:20   0:00          \_ ps uf
liuchinj 19414  0.0  0.0  18664  2864 pts/0    Ss   17:36   0:00 -tcsh
liuchinj   606  0.0  0.0  11744   984 pts/0    S+   23:20   0:00  \_ zombie
liuchinj   607  0.0  0.0      0     0 pts/0    Z+   23:20   0:00      \_ [zombie] <
-tcsh
 \_ spinner is consuming too much CPU time.
Attempting to kill process 610...
Successfully killed spinner program.
zombie process found:
-tcsh
 \_ zombie
     \_ [zombie] <defunct>
Attempting to kill process 606...
Successfully killed zombie program.
```