Operating Systems **CSI3131** Lab 1

Observation of process behavior - exercise

Goals

1. Study the behavior of process execution by examining its structures through the Linux "/ proc" directory.
2. Gain experience with Linux command line interface (shell) as well as system programs / commands.
3. Try some C programs with fork () and exec () calls.

Guidelines

Context: The Linux OS provides a convenient mechanism for exploring the properties of the system as well as those of the running processes. Several OS structures are visible as files in the pseudo filesystem found in the / proc directory.

Step 1: Start VirtualBox and run the Linux "SiteDev" virtual machine. To start the Linux virtual machine, select SiteDev and click Start. Another window will appear and you will see Linux booting up as if it were the primary OS on a PC. When prompted for a login prompt, use the user name "test1" and password "site" to log on. You are now ready to complete the lab. Note that you are using a CLI to interact with the OS.
The "SiteDev" virtual machine is a Linux environment in which you can complete your laboratory. Please follow the instructions for installing the virtual machine in the guide available on the course website.

Step 2 : Do the command ls /proc (any text with this font represents commands to be executed). The contents of /proc will be displayed. You will see several numeric names (which represent directories associated with processes - for example directory 23406 will contain information about the process with a PID 23406). You will also see filenames containing information (eg "version" and "cpuinfo").

Step 3 : Do the command cat /proc/version. It should display the contents of the "/proc /version" file which contains information about the OS version. Enter the command cat/proc/cpuinfo and examine the information displayed. Take the time to review the contents of other files.

Step 4: Enter the ps command. It will display the processes you have started. At least two processes will be displayed (one running "ps" and one "shell" process - most likely "bash"). Note the PID of the "shell" process.

Step 5 : Do the command cat / proc / XXXX / stat, where XXXX is the PID of your "shell" process from the previous step. You will notice a series of numbers representing the properties of the process. To know the meanings of the numbers with the command man proc (which gives the man page for proc). You can also examine the contents of "/ proc / XXXX / status" which gives similar information but in an easier to read format.

Step 6: We want to explore process state changes, but it is difficult to examine the activity of the "shell" process. In order to be able to examine interesting behavior, we will start and observe other programs created for this purpose.

From your linux, get access to your lab1 files:

1. *procmon*: This program reads every second the file "/ proc / PID / stat" the status of the process and the number of "jiffies" (1/100 of a second) spent in the kernel mode and the used mode. This information will be displayed. The program takes one parameter, the PID of the process which is to be observed. The procmon process terminates when it can no longer open the "/ proc / PID / stat" file - normally because the PID process has terminated.
2. The calcloop and cploop program files. These are two programs with very different behaviors.
    a. The calcloop process does the following:
        i. Start a loop (10 iterations) that sleeps for 3 seconds, and then start another loop that increases a variable 400,000,000 times. The calculation takes about 2 seconds of real time (this varies depending on the load of the computer).
    b. The cploop process does the following:
        i. Create the file 500,000 bytes long (fromfile),
        ii. Start a loop (10 iterations) that sleeps for 3 seconds, and then copy the fromfile to the tofile. The copy operation takes approximately 2 seconds of real time (this varies depending on the system load);
        iii. The program uses two system calls to copy one byte at a time.

    3. The tstcalc and tstcp "shell" programs. The first program starts the calcloop program gets the process PID and starts procmon with that PID to examine its behavior. After 20 seconds the script will terminate the calcloop process (procmon should terminate automatically, but the script will attempt to terminate it as a precaution). The output of procmon will be saved in the calc.log file which can be studied. The tstcp "shell" program does the same with cploop by saving the output of procmon to the cp.log file. To run these scripts, you simply type in their name, ie, tstcalc and tstcp.
4. A template for the C mon.c program. You will have to complete the missing code in this file.
5. A Folder called "**code"** contains: procmon.c, calcloop.c, and cploop.c C programs that you can examine. You can run the script **compile.sh** (./compile.sh) to

compile these files. The scripts will create executable files for these c programs (outside the code folder)

Step 7 : Then executes the script **tstcalc** (./tstcalc). Do the same with **tstcp** (./tstcp).

Step 8: Examine the contents of the "calc.log" and "cp.log" files. Answer the following questions :

1. Explain the changes in state, overhead, and user time, for the calcloop process by relating these changes to the operations completed by calcloop.
2. Explain the changes in state, overhead, and user time, for the cploop process by relating these changes to the operations completed by cploop.
3. Explain the difference between system time and user time between the two processes (ie why one process spends more time in user mode and / or in system mode than the other).

Step 9 : Here is the easy part. Now let's try some programming, using the fork () and exec () system calls we studied in class. Your goal: write a C mon.c program that provides most of the functions of tstcalc and tstcp:

1. *mon.c* uses one parameter: the M name of the program it should run.
2. *mon.c* must start the program M (this program will not take any parameters) and must determine its PID.
3. *mon.c* will need to start procmon with a parameter, the PID of the process running M.
4. *mon.c* falls asleep for 20s, and then ends program M, sleeps another 2 seconds and attempts to terminate procmon.
5. Don't worry about redirecting procmon output to a file (you don't have to worry about C I / O at this time).

Step 10: Compile your C program with the command **gcc mon.c –o mon**. If the compilation is completed without errors, an executable file mon will be created.

Step 11 : run mon calcloop and watch. If everything is working fine, you will see the same output found in the calc.log file. You can redirect this output to a file with the my calcloop> filename command.

Useful information:

- The execution of a C program always begins with the main () function which receives two parameters: integer argc giving the number of parameters of the command line and argv an array (with argc elements) of character strings which refers to the parameters from the command line. The convention states that argv [0] gives the name of the program and argv [1] gives its first parameter.
- The fork () system call creates a new process. The parent and child continue at the same point, ie, returning from the fork () call. But at the parent, fork () returns the

PID of the child while at the child, fork () returns the value 0. For more information about fork (), use man fork.

- The system call execl (path, arg1,…) replaces the image in a process with a program with given parameters. Tap to know the exact meaning of each of the parameters. The execl ("calcloop", "calcloop", NULL) call replaces the code in the process with the calcloop program. The execl ("/ bin / ls", "ls", "-l", NULL) call executes the code of the ls –l program.
- You need to convert the integer PID to a string to send it to procmon. This can be done in C with a library function: sprintf (pidchn, "% d", pid), where pid is an int variable which contains the value of pid, and pidchn is a char * variable which will receive the string of the PID to be sent to procmon.
- The sleep () function which will make the process sleep for the number of seconds specified in its parameter.
- The kill (pid, sig) system call which sends a sig signal to the pid process.
    - Look at signal.h for the different signals. ○ Google is your friend, the following link after a search for "signal.h" may be useful (there are many more) http://www.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html