Operating Systems **CSC465/665 *Lab*** *Spring 2019*

Observing Process Behavior - Exercise

Goals

1. Study the behavior of process execution by monitoring the process structures via the Linux "/proc" directory.
2. Get some experience with command line interface (shell) in Linux; learn some system commands/programs.
3. Get some experience running C programs
.

Guideline

Background: The Linux OS provides a convenient way to explore the system properties, as well as properties and states of the currently running processes. Many of the OS structures are visible as files in the pseudo-file system under the directory /proc.

Step 1: Start Linux on your Virtual Machine (as per the instructions I posted in the file named "Getting Set Up").

Step 2: Enter command `ps`. This will list the processes you have launched. At least two processes will be listed: 'ps' (the one you just launched and which is producing this output) and a shell process (probably 'bash', but could be different, depending on your settings). Write down the PID of the shell process.

Step 3: Enter command `cat /proc/XXXX/stat`, where XXXX is the PID of your shell process from the previous step. You will see a bunch of numbers containing lots of information about this process. You can learn the meaning of the numbers by entering `man proc` command (which would show you the manual for proc) and scrolling down. You can see most of this information in human readable form in the file '/proc/XXXX/status'.

Step 4: We want to explore the state changes of processes, but there is not much interesting activity by the shell process. In order to see more interesting behavior, we will launch and observe other, specially created programs. Copy the provided file 'lab1.tar' into your working directory on the Linux system (type cd **../../vagrant** to access it; Remember that that is the directory that corresponds to OS-VM on the host machine and which you access by typing **cd OS-VM** from the PowerShell prompt) and unttar it by executing command `tar -xvf lab1.tar`. This should extract the following files:

1. *procmon* This is a program which periodically (every second) reads the file '/proc/PID/stat' and extracts and prints the process state and the number of jiffies (1/100 of a second) this process spent in kernel and user mode, respectively. This program is launched with one parameter – the PID of the process it has to

monitor. *procmon* terminates when it cannot open the /proc/PID/stat file – usually because the process PID has terminated.

2. Program files *calcloop* and *cploop*. These are two programs with quite different behaviour:
    a. The program *calcloop* does the following:
        i. Goes into a loop (10 iterations) that sleeps for 3 seconds, and then starts another loop that increments a variable 400,000,000 times. The calculation takes approximately 2 secs of real time (this may vary according to the load on the system).
    b. The program *cploop* does the following:
        i. Creates a file that is 500,000 bytes long (fromfile)
        ii. Goes into a loop (10 iterations) that sleeps for 3 seconds, and then copies the fromfile to the tofile. The copy operation takes approximately 2 secs of real time (this may vary according to load on the system).
        iii. The program uses two system calls to copy bytes one at a time.
3. Shell program files *tstcalc* and *tstcp*. *tstcalc* launches the *calcloop* program, gets its PID and then launches *procmon* with this PID to monitor the behaviour of *calcloop*. After the *calcloop* has run for 20s, *tstcalc* terminates it and if *procmon* did not terminate by itself within 1s, it is terminated as well. The output of *procmon* is redirected to file 'calc.log', where it can be later examined. *tstcp* does the same for *cploop*, saving the *procmon*'s output in file 'cp.log' To run these shell programs, type in their name preceded by "./", i.e. ./tstcalc and ./tstcp.
4. A directory that contains the C programs *procmon.c*, *calcloop.c* and *cploop.c* for your perusal.

Step 5: Launch tstcalc. Do the same with tstcp.

Step 6: Examine the contents of the files 'calc.log' and 'cp.log' and answer the following questions:

1. Explain the changes you see in the state, system time, and user time, for the *calcloop* process relating these changes to the operations completed by *calcloop.*
2. Explain the changes you see in the state, system time, and user time, for the *cploop* process relating these changes to the operations completed by *cploop.*
3. Explain the difference of time spent in system time and user time between the two programs (i.e why would one program spend more time in user mode and/or system mode than the other).

Step 7: OK, all of this was easy. Now let' s try to do some programming, playing around with the fork() and exec() calls we learned about in the lectures. Your goal: write a C program mon.c which will provide most of the functionality of tstcalc and tstcp:

1. mon.c has one command line argument: the name M of the program it has to launch
2. mon.c will launch the program M (the program M does not take any parameters) and learn its PID.

3. mon.c will launch *procmon* with the argument PID (Note: the PID first needs to be converted to an integer string using `sprintf(pidStr, "%d", prpid)` to be passed as an argument for `procmon` when you call it with `execl`)
4. mon.c will sleep for 20s, then it will terminate program M, will sleep 2 more seconds and will (try to) terminate *procmon*.
5. Do not worry about redirecting the output of *procmon* into a log file (no need to bother you with file I/O in C at this moment)

Step 8: Compile your C progam by entering `cc mon.c –o mon`. If the compilation succeeded, an executable called mon has been created.
Step 9: Enter `mon calcloop` and observe. If everything worked well, you should see the same output as you have seen in the log file 'calc.log'. You can redirect this output into a file by entering `mon calcloop > filename`.

Background you might need:

- The execution of a C program starts in procedure main() which has two arguments: integer argc containing the number of command line arguments and an argc-element array of strings argv, containing the command line arguments. By convention, argv[0] is the name of the program, argv[1] is its first argument.
- the fork() command creates a new process. Both the parent and the child continue as if they returned from a fork() call, however the parent gets as return value the PID of the child, while the child gets 0. Type `man fork` to read more about fork().
- The execl(path, arg1, …) command replaces the current process with the specified program launched with the provided arguments. Type `man execl` to learn about the exact meaning of its arguments.
  The code 'execl("calcloop", "calcloop", NULL)' will replace the current process with the *calcloop* program. The code 'execl("/bin/ls", "ls", "-l", NULL' will launch `ls –l`.
- You will need to convert the integer PID into a string to pass to procmon. In C you can do that by calling library function string *itoa(int num, string *buf, int radix). (Check whether this really works in your environment!) Alternatively, you can use: `sprintf(…, "%d", …)`
- Function sleep() will cause the program to sleep for the specified number of seconds.
- Function kill(pid, sig) will send a signal sig to process pid.
  - have a look at signal.h to see the different signals
  - google is your friend, this search result on 'signal.h' might be helful (there are many more)

  http://www.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html
- To run an executable file, type ./<filename>
- To make a file executable, type chmod 777 <filename>
- The above is dangerous: type man chmod to learn more about chmod

- To compile a C program in the environment I suggested you use, you will first need to type:
    - sudo apt-get install libc6-dev