A Multiple Occurrence Test Harness

Due at 11:59pm on Monday, 31 October 2016

In this project, you will implement a program that initiates and schedules multiple instances of an application program. This Test Harness (TH) determines the command to execute, the number of processes executing that command, and the number of processor cores upon which to schedule those processes; these are obtained from environment variables or command arguments to TH. It creates the processes, and then schedules the processes to run concurrently in a time-sliced manner. It will also monitor the processes, keeping track of how the processes are using system resources.

There are 4 parts to the project, each building on the previous one. The objectives of the project are to give you a good introduction to environment variables, processes, signals and signal handling, and round-robin scheduling.

All coding must be done in the C programming language; it must be gcc-compilable and runnable in the Linux virtual machine environment. You must restrict yourself to Linux system calls (those routines described in section 2 of the manual; this means, for example, you may ***not*** use printf(3), fgets(3), system(3), strlen(3), etc. I have provided a set of helper functions defined in p1fxns.h that you may find useful. The only exception to this edict is that you may use malloc(3)/free(3)/calloc(3) if you need to; note that other Chapter 3 calls that you may use are enumerated in the table in section 5.1 below.

You should thoroughly read and understand Chapter 8 of the Linux Programming Environment before tackling this project. It is available on Canvas.

You should tackle this problem in four steps, outlined below. Each step should be in a separate source file, thv?.c, where ? is replaced by 1, 2, 3, or 4. The solution at each step will be graded separately.

The usage string for thv? is:

```
usage: ./thv? [--number=<nprocesses>] [--processors=<nprocessors>] --command='command'
```

where ? is replaced by 1, 2, 3, or 4. The number of processes and processors is optional, since thv? will look in the environment for environment variables named TH_NPROCESSES and TH_NPROCESSORS, respectively, before processing the command line arguments. Thus, if either or both of these environment variables are defined, then thv? takes the values from the environment variable. If the corresponding argument is also specified, the argument overrides the environment variable. See chapter 6.9 of the *Linux Programming Environment* for a discussion of environment variables and accessing them from within your program.

1 TH launches all the processes at once, ignoring the number of processors

The goal of Part 1 is to develop the first version of the TH such that it can launch all of the processes to execute at one time. thv1 will perform the following steps:

- Obtain the number of processes to create (nprocesses), the number of processors upon which to run those programs (nprocessors), and the command to execute in each of the processes (command) from the environment variables and command arguments to TH.

- Note the current time (start)
- Launch 'nprocesses' processes executing 'command' using fork(), execvp(), and any other required system calls. To make things simpler, assume that the programs will run in the same environment as used by TH.
- Once all of the processes are running, wait for each process to terminate.
- Note the current time (stop)
- Compute the elapsed time (stop - start) that it took for all of the processes to complete their processing and display on standard output[1]
- Exit

The launching of each processes executing 'command' will look something like this in pseudocode:

```
prepare argument structure
for i in 0 .. nprocesses-1
    pid[i] = fork();
    if (pid[i] == 0)
        execvp(args[0], args)
for i in 0 .. nprocesses-1
    wait(pid[i])
```

While this may appear to be simple, there are many things that can go wrong. You should spend some time reading the entire man page for the fork(), execvp(), and wait() system calls.

2 TH Takes Control

Successful completion of Part 1 gives you a basic working TH. Our ultimate goal is to schedule the processes to execute on the number of processors indicated in a time-shared manner. Part 2 takes the first steps to enable TH to gain control for this purpose.

Firstly, we need to implement a way for thv2 to stop all processes just before they call execvp() so the TH can decide which processes to run first. The idea is to have each forked child process wait for a SIGUSR1 signal before calling execvp(). The sigwait() system call may be useful here; chapter 8 in the *Linux Programming Environment* shows you a better mechanism by which this can be achieved. The TH parent process sends the SIGUSR1 signal to the corresponding forked (TH) child process. Note that until a forked child process performs the execvp() system call, it is running the TH program code.

Once this is working, the TH is in a state (after launching all of the processes) where each process is waiting on a SIGUSR1 signal. The first time that a forked process is selected to run by the TH scheduler, it is started by the TH sending the SIGUSR1 signal to it.

Secondly, we need to implement a mechanism for the TH to signal a running process to stop (using the SIGSTOP signal) and to continue it again (using the SIGCONT signal). This is the mechanism that the TH will use on a process after it has been started the first time. Sending a SIGSTOP signal to a running process is like running a program in the shell and typing Ctrl-Z to suspend (stop) it. Sending a suspended process a SIGCONT signal is like bringing a suspended job into the foreground in the shell.

[1] The format string to use for displaying this on standard output is
"The elapsed time to execute %d copies of \"%s\" on %d processors is %7.3fsec\n".

Thus, in Part 2, you will implement these two steps to create a thv2 building on thv1 in the following way:

- Fork nprocesses child processes; immediately after each process is created using fork(), the child process waits on the SIGUSR1 signal before calling execvp().
- Note the current time (start)
- The TH parent process sends each program a SIGUSR1 signal to wake them up. Each process will then wake up and invoke execvp() to run the workload process.
- After all of the processes have been awakened and are executing, the TH sends each process a SIGSTOP signal to suspend it.
- After all of the processes have been suspended, the TH sends each process a SIGCONT signal to resume it.
- Once all processes are back up and running, the TH waits for each process to terminate.
- Note the current time (stop)
- Compute the elapsed time (stop - start) that it took for all of the processes to complete their processing and display on standard output
- Exit

thv2 demonstrates that we can control the suspension and resumption of processes.

Handling asynchronous signalling is far more nuanced than described here – you should spend time reading the entire man pages for these system calls and references online, the relevant sections of Chapter 8 of the *Linux Programming Environment*, and other printed resources (such as the books suggested on the course web page) to gain a better understanding of signals and signal handling.

3 TH Schedules Processes

Now that the TH can suspend and resume processes, we want to implement a scheduler that runs the processes according to some scheduling policy. The simplest policy is to equally share the processor by giving each process the same amount of time to run (e.g., 250 ms). In this case, there are 'nprocessors' processes executing at any given time. After the time slice has completed, the currently running processes need to be suspended, and another 'nprocessors' processes need to be resumed. The TH decides the next set of processes to run, starts a timer, and resumes those processes.

thv2 knows how to resume a process, but we still need a way to have each child process run for only a certain amount of time. Note, if some process is running, it is still the case that the TH is running concurrently with it. Thus, one way to approach the problem is for the TH to poll the system time to determine when the time slice has expired. This is inefficient, as it is a form of busy waiting. Another approach is to set an alarm using the alarm(2) system call. This tells the operating system to deliver a SIGALRM signal after some specified time; unfortunately, the finest granularity time that can be specified to the alarm system call is 1 second. The setitimer(2) system call enables one to establish an interval timer. Signal handling is done by registering a signal handling function with the operating system. This SIGALRM signal handler is implemented in the TH. When the signal is delivered, the TH is interrupted and the signal handling function is executed. When it does, the TH will suspend the running processes, determine the next 'nprocessors' processes to run, and send each a SIGCONT signal, and continue with whatever else it is doing.

Your new and improved thv3 is now a working process scheduler. However, you need to take care of several things. For instance, there is the question of how to determine if a process is still executing. At some point (we hope), the process is going to terminate. Remember, this process is a child process of the TH. How does the TH know that a process has terminated? In thv2, we just called wait(). Is that sufficient now? You will likely need to explore implementing a SIGCHLD handler.

4 TH as Big Brother

With thv3, the processes are able to be scheduled to run with each receiving an "equal" share of the processors. Note, thv3 should be able to work with any number of processes and any number of processors. thv4 now displays how the process execution is proceeding (by looking in the /proc directory for information on the child processes), in addition to displaying the total elapsed time after all have concluded.

In Part 4, you will add some functionality to the TH to gather relevant data from /proc that conveys some information about what system resources each workload process is consuming. This should include something about execution time, memory used, and I/O. It is up to you to decide what to look at, analyze, and present. Do not just dump out everything in /proc for each workload process. The objective is to give you some experience with reading, interpreting, and analyzing process information. Your thv4 should output the analyzed process information periodically as the processes are executing. One thought is to do something similar to what the Linux top(1) program does.

5 Other Considerations

5.1 System Calls

In this project, you will likely want to learn about these system calls:

fork(2)	execvp(3)	wait(2)
signal(2)	kill(2)	_exit(2)
setitimer(3)	errno(3)	gettimeofday(2)
open(2)	read(2)	close(2)
getenv(3)		

5.2 Error Handling

All system call functions that you use will report errors via the return value. As a general rule, if the return value is less than zero, then an error has occurred and errno is set accordingly. You must check your error conditions and report errors. To expedite the error checking process, you are allowed to use the perror(3) library function. Although you are allowed to use perror, it does not mean that you should report errors with voluminous verbosity. Report fully but concisely.

5.3 Memory Errors

You are required to check your code for memory errors. Code that contains memory leaks and memory violations will have marks deducted. Fortunately, the valgrind tool can help you detect and correct these issues.

5.4 Developing Your Code

The best way to develop your code is in Linux running inside the virtual machine image provided to you. This way, if you crash the system, it is straightforward to restart. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state.

You should use your Bitbucket GIT repositories for keeping track of your programming efforts. As a reference, you can perform the command line steps below to create a new project directory and upload it to your uoregon-cis415 repository.

```
% cd /path/to/your/uoregon-cis415
% mkdir project1
% echo "This is a test file." >project1/testFile.txt
% git add project1
% git commit –m "Initial commit of project1"
% git push –u origin master
```

Any subsequent changes or additions can be saved via the add, commit, and push commands.

5.5 Helping your Classmates

This is an individual assignment. You should be reading the manuals, reading relevant sections of the *Linux Programming Environment*, hunting for information, and learning those things that enable you to do the project. However, it is important for everyone to make progress and hopefully obtain the same level of knowledge by the project's end. If you get stuck, seek out help to get unstuck. Sometimes just having another pair of eyes looking at your code is all you need. If you cannot obtain help from the graduate TA or the lecturer, it is possible that a classmate can be of assistance.

In your status report on the project, you should provide the names of classmates that you have assisted, with an indication of the type of help you provided. You should also indicate the names of classmates from whom you have received help, and the nature of that assistance. You do ***not*** need to document assistance from the lecturer of the graduate TA.

Each of your source files must start with an "authorship statement", contained in C comments, as follows:

- state your name, your login, and the title of the assignment (CIS 415 Project 0)
- state either "This is my own work." or "This is my own work except that …", as appropriate.

Note that this is not a license to collude. We will be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion.

6 Submission[2]

You will submit your solutions electronically by uploading a gzipped tar archive via Canvas.

Your TGZ archive should be named <duckid>-project1.tgz, where <duckid> is your "duckid". It should contain thv1.c, thv2.c, thv3.c, thv4.c, a Makefile for creating executables named thv1, thv2, thv3, and thv4, and a document named report.pdf or report.txt, describing the state of your solution, and documenting anything of which we should be aware when marking your submission. If you have created any additional .c or .h files upon which thv[1-4] depend, be sure to include them in the archive, as well. If you have included p1fxns.h in your source files, you should include p1fxns.h and p1fxns.c in your TGZ archive.

Within the archive, these files should be contained in a folder named <duckid>. Thus, if you upload "jsventek-project1.tgz", then we should see something like the following when we execute the following command:

```
$ tar –ztvf jsventek-project1.tgz
-rw-r--r-- jsventek/group     1021 2016-10-30 16:37 jsventek/Makefile
-rw-r--r-- jsventek/group     5815 2016-10-30 16:37 jsventek/p1fxns.c
-rw-r--r-- jsventek/group     2367 2016-10-30 16:37 jsventek/p1fxns.h
-rw-r--r-- jsventek/group     3670 2016-10-30 16:30 jsventek/th1.c
-rw-r--r-- jsventek/group     5125 2016-10-30 16:37 jsventek/th2.c
-rw-r--r-- jsventek/group     6531 2016-10-30 16:37 jsventek/th3.c
-rw-r--r-- jsventek/group     8127 2016-10-30 16:37 jsventek/th4.c
-rw-r--r-- jsventek/group   629454 2016-10-30 16:30 jsventek/report.pdf
```

as well as any other files that you have included.

Each of your source files must start with an "authorship statement", contained in C comments, as follows:

- state your name, your duckid, and the title of the assignment (CIS 415 Project 1)
- state either "This is my own work." or "This is my own work except that …", as appropriate.

We will be compiling your code and testing against unseen commands. We will also be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion.

[2] ***A 20% penalty will be assessed if you do not follow these submission instructions. See the handout for Project 0 if you do not remember how to create the gzipped tar archive for submission.***

Marking Scheme for CIS 415, Project 1

Your submission will be marked on a 100 point scale. I place substantial emphasis upon **WORKING** submissions, and you will note that a large fraction of the points is reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles, links, and runs correctly. The information returned to you will indicate the number of points awarded for the submission.

You must be sure that your code works correctly on the virtual machine under VirtualBox, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the virtual machine before submission.

Version	Points	Description
(10)	10	Your report – honestly describe the state of your submission
TH v1	6	for workable solution
		process environment variables
(20)		process command line arguments
		parse command correctly
		for nprocesses
		fork a child process
		in the child process, exec the parsed command
		parent waits till all the children are terminated
		parent terminates gracefully after all the children are terminated
	1	if it successfully compiles
	1	if it compiles with no warnings
	1	if it successfully links
	1	if it links with no warnings
	6	if it works correctly
	4	if there are no memory leaks
TH v2	6	for workable solution
		each child waits for SIGUSR1 upon creation
(20)		parent sends SIGUSR1 to all the children after creating all of them
		child execs command after handling SIGUSR1
		parent sends SIGSTOP to all the children after sending SIGUSR1
		all the children respond and stop
		parent sends SIGCONT to all the children after sending SIGSTOP
		all the children respond and resume
		parent waits till all the children are terminated
		parent terminates gracefully after all the children are terminated
	1	if it successfully compiles
	1	if it compiles with no warnings
	1	if it successfully links
	1	if it links with no warnings
	6	if it works correctly
	4	if there are no memory leaks

Version	Points	Description
TH v3 (30)	10	for workable solution each child waits for SIGUSR1 upon creation parent enables interval timer for quantum properly implemented round robin scheduling stop running processes if any are running select the next processes to run send SIGUSR1 or SIGCONT appropriately to selected child parent properly reaps terminated children parent terminates gracefully after all the children are terminated
	1	if it successfully compiles
	1	if it compiles with no warnings
	1	if it successfully links
	1	if it links with no warnings
	10	if it works correctly
	6	if there are no memory leaks
TH v4 (20)	6	for workable solution periodically accesses the proc file system extracts meaningful statistics formats them and prints them
	1	if it successfully compiles
	1	if it compiles with no warnings
	1	if it successfully links
	1	if it links with no warnings
	6	if it works correctly
	4	if there are no memory leaks