A User Space Process Scheduler

Due at 11:59pm on Thursday, 30 April 2015

In this project, you will implement a program that initiates and schedules a workload of other processes. This User Space Process Scheduler (USPS, not to be confused with the US Postal Service ☺) reads a list of programs, with arguments, from standard input, starts up the programs in 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. For extra credit, you can try to improve the scheduling strategy to give more time to CPU-bound processes and less time to I/O-bound processes.

There are 4 parts to the project, each building on the other. The objective of the project is to give you a good introduction to processes, signals and signal handling, and scheduling. Part 5 is for extra credit.

All coding must be done in the C programming language; it must be 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, and you may use `perror(3)` to display errors associated with any of the system calls.

1 USPS Launches the Workload

The goal of Part 1 is to develop the first version of the USPS such that it can launch the workload and get all of the processes running together. USPS v1 will perform the following steps:

- Read the program workload from the standard input. Each line in the file contains the name of the program and its arguments.
- For each program, launch the program to run as a process using the `fork()`, `execvp()`, and any other required system calls – see below. To make things simpler, assume that the programs will run in the same environment as used by USPS.
- Once all of the programs are running, wait for each process to terminate.
- After all of the processes have terminated, the USPS exits.

The launching of each program in the workload will look something like this in pseudocode:

```
for i in 0 .. numprograms-1
    pid[i] = fork();
    if (pid[i] == 0)
        prepare argument structure;
        execvp(program[i], args[i])
for i in 0 .. numprograms-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 all three of these system calls.

2 USPS Takes Control

Successful completion of Part 1 gives you a basic working USPS. Our ultimate goal is to schedule the programs in the workload to execute in a time-shared manner. Part 2 takes the first steps to enable USPS to gain control for this purpose.

Firstly, we need to implement a way for the USPS to stop all processes just before they call execvp() so the USPS can decide which process to run first. The idea is to have each forked child process to wait for a SIGUSR1 signal before calling execvp(). The sigwait() system call will be useful here. The USPS parent process sends the SIGUSR1 signal to the corresponding forked (USPS) child process. Note that until a forked child process performs the execvp() system call, it is running the USPS program code.

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

Secondly, we need to implement a mechanism for the USPS 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 USPS 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.

Thus, in Part 2, you will implement these two steps to create a USPS v2 building on USPS v1 in the following way:

- Immediately after each process is created using fork(), the child process waits on the SIGUSR1 signal before calling execvp().
- After all of the processes have been created and are awaiting the SIGUSR1 signal, the USPS parent process sends each program a SIGUSR1 signal to wake them up. Each process will then return from the sigwait() call and invoke execvp() to run the workload process.
- After all of the processes have been awakened and are executing, the USPS sends each process a SIGSTOP signal to suspend it.
- After all of the workload processes have been suspended, the USPS sends each process a SIGCONT signal to resume it.
- Once all processes are back up and running, the USPS waits for each process to terminate. After all have terminated, USPS exits

USPS v2 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 reference online and printed resources (such as the books suggested on the course web page) to gain a better understanding of signals and signal handling.

3 USPS Schedules Processes

Now that the USPS can suspend and resume workload 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., 1 second). In this case, there is 1 workload process executing at any given time. After its time slice has completed, we need to suspend that process and start up another ready process. The USPS decides the next workload process to run, starts a timer, and resumes that process.

USPS v2 knows how to resume a process, but we still need a way to have it run for only a certain amount of time. Note, if some workload process is running, it is still the case that the USPS is running concurrently with it. Thus, one way to approach the problem is for the USPS to poll the system time to determine when the time slice has expired. This is inefficient, as it is a form of busy waiting. Alternatively, you can set an alarm using the `alarm(2)` system call. This tells the operating system to deliver a `SIGALRM` signal after some specified time. Signal handling is done by registering a signal handling function with the operating system. This `SIGALRM` signal handler is implemented in the USPS. When the signal is delivered, the USPS is interrupted and the signal handling function is executed. When it does, the USPS will suspend the running workload process, determine the next workload process to run, and send it a `SIGCONT` signal, reset the alarm, and continue with whatever else it is doing.

Your new and improved USPS v3 is not 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 workload process is still executing. At some point (we hope), the workload process is going to terminate. Remember, this workload process is a child process of the USPS. How does the USPS know that the workload process has terminated? In USPS v2, we just called `wait()`. Is that sufficient now? Be careful.

4 USPS as Big Brother

With USPS v3, the workload processes are able to be scheduled to run with each receiving an "equal" share of the processor. Note, USPS v3 should be able to work with any set of workload programs it reads in. In particular, we will provide you with some workload to run (to be determined) that will (ideally) give some feedback to you that your USPS v3 is working correctly.[1] It is also possible to see how the workload execution is proceeding by looking in the `/proc` directory for information on workload processes.

In Part 4, you will add some functionality to the USPS 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 USPS v4 should output the analyzed process information periodically as the workload programs are executing. One thought is to do something similar to what the Linux `top(1)` program does.

5 (Extra Credit) – Adaptive USPS

When the USPS schedules a workload process to run, it assumes that the process will actually execute the entire time of the time slice. However, suppose that the process is doing I/O, for example, waiting for the user to enter something on the keyboard. In general, workload processes could be doing different things and thus have different execution behaviors. Some processes may be CPU-bound, while others may be I/O-bound. If the USPS know something about process behavior, it is possible that the time slice could be adjusted for each type of process. For instance, I/O-bound processes might be given a little less time and CPU-bound processes a bit

[1] Note, you can also write your own simple test programs.

more. By adjusting the time slice, it is possible that the entire workload could run more efficiently.

Part 5 is to implement some form of adjustable scheduler that uses process information to set process-specific time intervals.

6 Other Considerations

6.1 System Calls

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

- fork(2)
- execvp(2)
- wait(2)
- sigwait(2)
- alarm(2)
- signal(2)
- kill(2)
- exit(2)
- read(2)
- write(2)
- open(2)
- close(2)

6.2 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 you 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.

6.3 Memory Errors

You are required to check your code for memory errors. This is non-trivial task, but a very important one. Code that contains memory leaks and memory violations will have marks deducted. Fortunately, the `valgrind` tool can help you detect and correct these issues.

6.4 Developing Your Code

The best way to develop your code is in Linux running in side 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 you programming work. You may also need to turn in your project in this way. 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.

6.5 Helping your Classmate

This is an individual assignment. You should be reading the manuals, 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 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.

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.

7 Submission

To be supplied …