A User Space Process Scheduler

Due at 11:59pm on Sunday, 30 April 2023

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 a file specified as an argument to the scheduler (uses standard input if unspecified), 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.

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.

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)/realloc(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 Elegant C Programming draft [*SYSC*] before tackling this project. It is available at Canvas/Files/Miscellaneous/JustCh8.pdf.

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

The usage string for `uspsv?` is:

```
usage: ./uspsv? [-q <quantum in msec>] [workload_file]
```

where `?` is replaced by `1`, `2`, `3`, or `4`. The quantum (in milliseconds) is optional, since `uspsv?` must look in the environment for an environment variable named `USPS_QUANTUM_MSEC`, before processing the command line arguments. Thus, if the environment variable is defined, then `uspsv?` takes the value from the environment variable; if the corresponding argument is also specified, the argument overrides the environment variable; if neither is specified, the program should exit with an error message. If the `workload_file` argument is not specified, the program is to read the commands and arguments from its standard input. See section 2.6.1 of Canvas/Files/Readings/Svenek-CaDS21F.pdf for a discussion of environment variables.

Accessing environment variables from your programs

The routine `getenv("var")`, defined in `<stdlib.h>`, searches the environment for the shell variable `var` and returns its value as a string of characters, or `NULL` if the variable is not defined.

To use environment variables in your program, you will use something like the following code:

```
#include <stdlib.h>

* * *
char *p;
int val = -1;

if ((p = getenv("VARIABLE_NAME")) != NULL)
    val = atoi(p);
```

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 specified file/standard input. Each line in the file contains the name of the program and its arguments (just as you would present them to bash).
- 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 and understanding the man pages for these three system calls.

NB – if your submission "fork bombs" and crashes the virtual machine in which we are attempting to assess your code, you will receive a 0 for the assignment.

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 wait for a `SIGUSR1` signal before calling `execvp()`. The `sigwait()` system call may be useful here; *SYSC* shows you a ***better*** mechanism by which this can be achieved. The USPS parent process sends the `SIGUSR1` signal to the corresponding forked (USPS) child process. Note that until a forked child process successfully 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 child process will then 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 and understanding the 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., 250 ms). 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.

You may use an appropriate ADT from the Oregon ADT library for your ready queue if you wish.

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; unfortunately, the finest time granularity 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 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, and continue with whatever else it is doing.

Your new and improved USPS v3 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 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? You will likely need to explore implementing a SIGCHLD handler – this is discussed in section 1.4.6 of *SYSC*.

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 a 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.

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

In Part 4, you will add functionality to the USPS to gather relevant data from `/proc` that conveys some information about what system resources each workload process is consuming and output this information to standard output. This should include something about the command being executed, 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 Other Considerations

5.1 System Calls

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

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

5.2 p1fxns.h

```
/*
 * a potentially useful set of subroutines for use with CIS 415
 * project 1
 */

#ifndef _P1FXNS_H_
#define _P1FXNS_H_

#include <stdbool.h>

/*
 * p1getline - return EOS-terminated character array from fd
 *
 * returns number of characters in buf as result, 0 if end of file
 */
int p1getline(int fd, char buf[], int size);

/*
 * p1strchr - return the array index of leftmost occurrence of 'c' in 'buf'
 *
 * return -1 if not found
 */
int p1strchr(char buf[], char c);

/*
 * p1getword - fetch next blank-separated word from buffer into word
 *
```

```
 * return value is index into buffer for next search or -1 if at end
 *
 * N.B. assumes that word[] is large enough to hold the next word
 */
int p1getword(char buf[], int i, char word[]);

/*
 * p1strlen - return length of string
 */
int p1strlen(char *s);

/*
 *
 * p1strdup - duplicate string on heap
 *
 */
char *p1strdup(char *s);

/*
 * p1putint - display integer in decimal on file descriptor
 */
void p1putint(int fd, int number);

/*
 * p1putstr - display string on file descriptor
 */
void p1putstr(int fd, char *s);

/*
 * p1perror(int fd, char *str) - writes 'str' and string describing
 *                                    the last error on 'fd'
 */
void p1perror(int fd, char *str);

/*
 * p1atoi - convert string to integer
 */
int p1atoi(char *s);

/*
 * p1itoa - format integer as decimal string
 */
void p1itoa(int number, char *buf);

/*
 * p1strcpy - copy str2 into str1
 */
void p1strcpy(char *str1, char *str2);

/*
 * p1strcat - concatenate str2 onto str1
 */
void p1strcat(char *str1, char *str2);

/*
 * p1strneq - determine if the first n characters of two strings are equal
 *
 * returns true if the first n characters are equal, false if not
 */
bool p1strneq(const char *s1, const char *s2, int n);

/*
 * p1strpack - pack justified strings into buffer
```

```
 *
 * packs st into buf, with a field width fw, appending fill character fc
 *      to st to make up fw;
 *      if fw < 0, packs fc before st, up to a field width of |fw|;
 * appends '\0' to buf, returns a pointer to the '\0' as the function value
 *
 * examples:
 *   p1strpack("1", 5, '0', buf) packs "10000" into buf, returns buf+5
 *   p1strpack("3", -5, '0', buf) packs "00003" into buf, returns buf+5
 *   p1strpack("3", 0, ' ', buf) packs "3" into buf, returns buf+1
 */
char *p1strpack(char *st, int fw, char fc, char *buf);

#endif /* _P1FXNS_H_ */
```

5.3 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 `p1perror()` function described above. Although you are allowed to use `p1perror()`, it does not mean that you should report errors with voluminous verbosity. Report fully but concisely.

5.4 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. Note that if you are running your USPS under `valgrind`, and if the `execvp()` in a child branch after a `fork()` fails, `valgrind` continues to run in that child process; if that child process exits without returning any heap-allocated memory, `valgrind` will complain, and you will lose marks.

5.5 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 work. 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.

5.6 Testing Your Code

You should construct various workload files to test your `uspsv?` programs. Note that since your programs are not using a search path before exec'ing each command in your workload file, the first word of each of your command lines must be a full path. If you want to use `echo`, for example, you must use `which` to determine the absolute path for `echo`; on your Debian Linux system, this is `/usr/bin/echo`.

If you wish to reference programs in the current directory, then the first word in your workload file should be `./program` where `program` is obviously modified to be the name of the program in the current directory.

In `P1start.tgz`, I have provided the source for two programs that you can use in your workload files while testing your programs: `cpubound.c` and `iobound.c`. Here is a command line for building `cpubound`:

```
gcc -o cpubound -W -Wall cpubound.c
```

You will need to read the source files to know the arguments that these commands take.

5.7 Helping your Classmate

This is an individual assignment. You should be reading the manuals, reading relevant sections of *SYSC*, 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, the LAs, or the instructor, 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 1)
- 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 uspsv1.c, uspsv2.c, uspsv3.c, uspsv4.c, a Makefile for creating executables named uspsv1, uspsv2, uspsv3, and uspsv4, and a document named 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 uspsv[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 **not** be contained in a folder. Thus, if I upload "jsventek-project1.tgz", then I should see something like the following when I execute the following command:

```
$ tar -ztvf jsventek-project1.tgz
-rw-r--r-- jsventek/group     1021 2016-10-30 16:37 Makefile
-rw-r--r-- jsventek/group     5815 2016-10-30 16:37 p1fxns.c
-rw-r--r-- jsventek/group     2367 2016-10-30 16:37 p1fxns.h
-rw-r--r-- jsventek/group     3670 2016-10-30 16:30 uspsv1.c
-rw-r--r-- jsventek/group     5125 2016-10-30 16:37 uspsv2.c
-rw-r--r-- jsventek/group     6531 2016-10-30 16:37 uspsv3.c
-rw-r--r-- jsventek/group     8127 2016-10-30 16:37 uspsv4.c
-rw-r--r-- jsventek/group     1536 2016-10-30 16:30 report.txt
```

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 25% 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.***

Rubric 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 your virtual machine, 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
USPS v1 (20)	6	for workable solution process environment variable process command line arguments for each line from standard input parse command and arguments correctly 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
USPS v2 (20)	6	for workable solution each child waits for SIGUSR1 upon creation 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
USPS 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 process select the next process 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
USPS v4 (20)	6	for workable solution - working version of uspsv3 PLUS 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
	10	Extra credit if the display is particularly interesting and/or ingenious