CSc 360: Operating Systems (Fall 2017)

Programming Assignment 1
P1: A Process Manager (PMan)

Spec Out: Sept. 7, 2017
Code Due: Oct. 2, 2017 (23:55 pm)

1 Goals

This assignment is designed to help you:

1. get familiar with C programming,
2. get familiar with system calls related to process management,
3. get familiar with the process control block (PCB).

You are required to implement your solution in C (other languages are not allowed). Your work will be tested on `linux.csc.uvic.ca`.

Note: `linux.csc.uvic.ca` is a *particular* machine at the UVic Department of Computer Science. It does *not* mean "any Linux machine" at UVic. Even more importantly, it does *not* mean any "Unix-like" machine, such as a Mac OS X machine—many students have developed their programs for their Mac OS X laptops only to find that their code works differently on `linux.csc.uvic.ca` resulting in a *substantial* loss of marks.

You can remote access `linux.csc.uvic.ca` by `ssh username@linux.csc.uvic.ca`. SSH clients are available for a wide variety of operating systems including Linux, Mac OS and Windows.

Be sure to study the `man` pages for the various systems calls and functions suggested in this assignment. The system calls are in Section 2 of the `man` pages, so you should type (for example):

```
$ man 2 waitpid
```

2 Schedule

In order to help you finish this programming assignment on time, the schedule of this assignment has been synchronized with both the lectures and the tutorials. There are three tutorials arranged during the course of this assignment.

Tutorial No.	Tutorial	Milestones
First	system programming in C, P1 spec go-through, design hints	design and code skeleton
Second	more on system programming and testing	alpha code done
Third	final testing and last-minute help	final deliverable

3 Requirements

3.1 Prompt for user input

Your PMan needs to show the prompt for user input. For example, when you run PMan by type in

```
linux.csc.uvic.ca:/home/user$ ./PMan
```

It prompts

```
PMan:  >
```

for user input.

3.2 Background Execution of Programs

PMan allows a program to be started in the background—that is, the program is running, but PMan continues to accept input from the user. You will implement a simplified version of background execution that supports executing processes in the background.

If the user types: `bg foo`, your PMan will start the program `foo` in the background. That is, the program `foo` will execute and PMan will also continue to execute and give the prompt to accept more commands.

The command `bglist` will have PMan display a list of all the programs currently executing in the background, e.g.,:

```
123:  /home/user/a1/foo
456:  /home/user/a1/foo
Total background jobs:  2
```

In this case, there are 2 background jobs, both running the program `foo`, the first one with process ID `123` and the second one with `456`.

Your PMan needs to support the following commands:

1. The command `bgkill` *pid* will send the `TERM` signal to the job with process ID *pid* to terminate that job.

2. The command `bgstop` *pid* will send the `STOP` signal to the job *pid* to stop (temporarily) that job.

3. The command `bgstart` *pid* will send the `CONT` signal to the job *pid* to **re**-start that job (which has been previously stopped).

See the `man` page for the `kill()` system call for details.

Your PMan must indicate to the user when background jobs have terminated. Read the man page for the `waitpid()` system call. You are suggested to use the `WNOHANG` option.

3.3 Status of Process

Your PMan needs to support a command `pstat` *pid* to list the following information related to process *pid*, where *pid* is the Process ID.

1. comm : The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.

2. state : One of the following characters, indicating process state: R (Running), S (Sleeping in an interruptible wait), D (Waiting in uninterruptible disk sleep), Z (Zombie), T (Stopped (on a signal) or (before Linux 2.6.33) trace stopped), t (Tracing stop (Linux 2.6.33 onward)), W (Paging (only before Linux 2.6.0)), X (Dead (from Linux 2.6.0 onward)), x (Dead (Linux 2.6.33 to 3.13 only)), K (Wakekill (Linux 2.6.33 to 3.13 only)), W (Waking (Linux 2.6.33 to 3.13 only)), P (Parked (Linux 3.9 to 3.13 only)).

3. utime: Amount of time that this process has been scheduled in user mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`). This includes guest time, guest_time (time spent running a virtual CPU, see below), so that applications that are not aware of the guest time field do not lose that time from their calculations.

4. stime: Amount of time that this process has been scheduled in kernel mode, measured in clock ticks (divide by `sysconf(_SC_CLK_TCK)`).

5. rss: Resident Set Size: number of pages the process has in real memory. This is just the pages which count toward text, data, or stack space. This does not include pages which have not been demand-loaded in, or which are swapped out.

6. voluntary_ctxt_switches: Number of voluntary context switches (since Linux 2.6.23).

7. nonvoluntary_ctxt_switches: Number of involuntary context switches (since Linux 2.6.23).

If process *pid* does not exist, PMan returns an error like:

```
Error:  Process 1245 does not exist.
```

In the above example, 1245 is the *pid* value.

To summarize, your PMan should support the following commands: `bg`, `bglist`, `bgkill`, `bgstop`, `bgstart`, and `pstat`. If the user types an unrecognized command, an error message is given by PMan, e.g.,

```
PMan:> ttest
PMan:> ttest:  command not found
```

4 Deliveries and Marking Scheme

For your final submission of each assignment you are required to submit your source code to connex dropbox. You should include a readme file to tell TA how to compile and run your code. TAs are not supposed to fix the bugs, either in your source code or in your make file. If following your readme file, the TAs cannot compile and run your code, you may get a zero mark for the assignment.

The marking scheme is as follows:

Components	Weight
Make file	5
Error handling	10
`bg`	10
`bglist`	10
`bgkill`	10
`bgstop`	10
`bgstart`	15
`pstat`	20
Code style	5
Readme.txt	5
Total Weight	100

5 Odds and Ends

5.1 Implementation Hints

1. Use `fork()` and `execvp()` so that the parent process accepts user input and the child process executes arbitrary commands.

2. Use a data structure (e.g., a linked list) to record the background processes.

3. Use the `\proc` pseudo-file system to find out the information required by `pstat`. Note that `\proc` is not a real file system, because all files have a size zero. The files just include a pointer to process control block (PCB) in the OS kernel. Due to this reason, **never try to write anything into `\proc`!**

5.2 Warning

Since you will use `fork()` in your assignment, it is important that you do not create a `fork()` bomb, which easily eats up all the pid resources allocated to you. If this happens, you cannot log into your account any more, even from a different machine. Both the instructor and the TAs cannot help you out. You have to go to IT support asking them to kill your buggy process. Clearly, IT support will not be happy if many students ask for such a help.

To avoid the mishap, you **MUST** use `ulimit -u` in bash to find out the default max number of user processes allocated to you (it is normally 50), and reduce this number to a safe value. For instance, if the default max number of user processes is 50, you can use

```
% ulimit -u 35
```

to reduce this number to 35. Therefore, even if your code has a bug and has created a `fork()` bomb, you still have unused 15 processes left and you can log in from a different machine to kill the buggy process.

Please take this warning seriously!

5.3 Compilation

You've been provided with a `Makefile` that builds the sample code (in `p1s.tar.gz`). It takes care of linking-in the GNU `readline` library for you. The sample code shows you how to use `readline()` to get input from the user, only if you choose to use the `readline` library.

5.4 Helper Programs

5.4.1 `inf.c`

This program takes two parameters:

tag: a single word which is printed repeatedly

interval: the interval, in seconds, between two printings of the tag

The purpose of this program is to help you with debugging background processes. It acts a trivial background process, whose presence can be "felt" since it prints a tag (specified by you) every few seconds (as specified by you). This program takes a tag so that even when multiple instances of it are executing, you can tell the difference between each instance.

This program considerably simplifies the programming of PMan which deals with re-starting, stopping, and killing programs.

5.4.2 `args.c`

This is a very trivial program which prints out a list of all arguments passed to it.

This program is provided so that you can verify that your PMan passes *all* arguments supplied on the command line—often, people have off-by-1 errors in their code and pass one argument less.

5.5 Code Quality

We cannot specify completely the coding style that we would like to see but it includes the following:

1. Proper decomposition of a program into subroutines (and multiple source code files when necessary)—A 500 line program as a single routine won't suffice.
2. Comment—judiciously, but not profusely. Comments also serve to help a marker, in addition to yourself. To further elaborate:
   (a) Your favorite quote from Star Wars or Douglas Adams' Hitch-hiker's Guide to the Galaxy does not count as comments. In fact, they simply count as anti-comments, and will result in a loss of marks.
   (b) Comment your code in English. It is the official language of this university.
3. Proper variable names—`leia` is not a good variable name, it never was and never will be.
4. Small number of global variables, if any. Most programs need a very small number of global variables, if any. (If you have a global variable named `temp`, think again.)
5. **The return values from all system calls and function calls listed in the assignment specification should be checked and all values should be dealt with appropriately.**

5.6 Plagiarism

This assignment is to be done individually. You are encouraged to discuss the design of your solution with your classmates, but each person must implement their own assignment.

Your markers will submit the code to an automated plagiarism detection program. We add archived solutions from previous semesters (a few years worth) to the plagiarism detector, in order to catch "recycled" solutions.

The End
