CMPUT 379 - Assignment #1 (10%)

Process Management Programs
(first draft)

Due: Thursday, October 4, 2018, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience in using Unix function calls that manage processes (e.g., fork(), waitpid(), and execl()).

Overview

In this assignment, you are asked to write the following process management programs in C/C++:

1. A program called "`a1jobs`" that provides the user with a tool to run, suspend, resume, and terminate execution of programs. Here a program is viewed as a *job* that may cause one, or more, process to run.

2. A program called "`a1mon`" that monitors and keeps track of the children processes spawned by a user specified process. If the specified process terminates, the monitor program terminates any spawned child process that is still running.

The two programs are independent of each other. However, for simplicity of implementation, the `a1jobs` program described below is not required to be accurate in terminating (or suspending/resuming) all processes associated with a job, and better cleanup results can be achieved by using `a1mon` to monitor an `a1job` process (and its spawned processes).

To implement the required programs, you need to familiarize yourself with some Unix functions including the functions mentioned in the following table.

function	Reference in Advanced Programming in the Unix Environment [SR 3/E]
Time values	Section 1.10
Time and date routines	Section 6.10
getrlimit() and setrlimit()	Section 7.11
fork(), waitpid(), execl()	Chapter 8: Process Control
popen, pclose	Section 15.3
times()	Section 8.17
kill()	Section 10.9

The `a1jobs` Program

The `a1jobs` program is invoked by the command line without any argument. After invocation, the program performs the following steps in order.

1. Use `setrlimit()` to set a limit on its CPU time (e.g., 10 minutes). The goal is to provide some safeguard against a buggy process that may run forever.

2. Call function `times()` (see the table above) to record the user and CPU times for the current process (and its terminated children).

3. Run the main loop of the program. In each iteration, the program prompts the user to enter a command line (using the prompt `"a1jobs[pid]:    "`, where `pid` is the process number of the running `a1jobs` process), and executes the command. Some commands cause the loop to terminate. The set of required commands are described below.

4. Upon exiting the main loop, the program calls function `times()` again to obtain the user and system CPU times for itself and its terminated children.

5. Using a setup and output format similar to the program in Figure 8.31 of [SR 3/E], `a1jobs` should use the timing information recorded in steps (2) and (4) to compute and print the following times in **seconds**:

   (a) the total time elapsed between steps (2) and (4),

   (b) the **user** and **system** CPU times spent by `a1jobs` in executing step (3), and

   (c) the **user** and **system** CPU times spent by the children processes started in step (3).

The program should maintain information of at most `MAXJOBS` (= 32) *admitted* jobs. A job is admitted if the `run` command described below can successfully run an associated user specified program. Admitted jobs are assigned indices $0, 1, \cdots, \mathtt{MAXJOBS} - 1$, in this order. A job terminated by the `terminate` command continues to keep its index until the `a1jobs` program exits. The program stores (at least) the following information for each admitted job: its index, pid of the job's head process, and the command line used to start the job. The program should handle the following user issued commands.

1. **list:** List all admitted jobs that have not been explicitly terminated by the user. Each entry of the listing contains (at least) the stored index, pid, and command line associated with the process.

2. **run pgm arg1 ... arg4:** Fork a process to run the program specified by `pgm`, using the given arguments (at most 4 arguments may be specified). A job is considered admitted if the program is successfully executed. The process running `pgm` is the head process of the job. As mentioned above, program `a1jobs` admits at most `MAXJOBS` jobs (the count includes jobs that the user has explicitly terminated).

3. **suspend** `jobNo`**:** Suspend the job whose index is `jobNo` by sending signal $SIGSTOP$ to its head process.

4. **resume** `jobNo`**:** Resume the execution of the job whose index is `jobNo` by sending signal $SIGCONT$ to its head process.

5. **terminate** `jobNo`**:** Terminate the execution of the job whose index `jobNo` by sending signal $SIGKILL$ to its head process.

6. **exit:** Terminate the head process of each admitted job that has not been explicitly terminated by the `terminate` command. Then exit the main loop.

7. **quit:** Exit the main loop without terminating head processes.

Example. In this example, we open two terminal windows on the same lab workstation. The first window is used to run `a1jobs`. The second window is used to run `ps` to monitor the effect of `a1jobs` on the running processes. Jobs that use the following programs are started: `xclock`, `xeyes` (two standard X Windows programs), and a shell script called `myclock`. Script `myclock` periodically appends the output of the `date` program to an output file and then calls the `sleep` program to delay the start of the next iteration. These programs are chosen because their output does not clutter the terminal window running `a1jobs`.

1. In the first window, we start four jobs using the above three programs.

```
a1jobs[1315]: run xclock -geometry 200x200 -update 2
a1jobs[1315]: run xeyes
a1jobs[1315]: run myclock out1
a1jobs[1315]: run myclock out2
```

2. In the second window, we use `ps` to see the related running processes. Note that the head process of each `myclock` job is a shell `/bin/sh`, and the job also runs a `sleep` process.

```
USER       PID  PPID  PGID S  STARTED CMD
...     .. ..    ..    .  ...      ...
ehab      1315  1087  1315 S 20:38:35 a1jobs
ehab      1318  1315  1315 S 20:39:04 xclock -geometry 200x200 -update 2
ehab      1319  1315  1315 S 20:39:15 xeyes
ehab      1322  1315  1315 S 20:39:32 /bin/sh ./myclock out1
ehab      1350  1315  1315 S 20:39:48 /bin/sh ./myclock out2
ehab      1497  1322  1315 S 20:40:12 sleep 2
ehab      1499  1350  1315 S 20:40:12 sleep 2
...     .. ..    ..    .  ...      ...
```

3. In the first window, we suspend the `xclock` job (the clock hands stop moving), and one of the two `myclock` jobs (the output file stops growing).

```
a1jobs[1315]: suspend 0
a1jobs[1315]: suspend 2
```

4. In the second window, we use `ps` to see the changes (look at the process state column `S`).

```
USER       PID  PPID  PGID S  STARTED CMD
...    ...  ...     ..    .  ...      ...
ehab      1315  1087  1315 S 20:38:35 a1jobs
ehab      1318  1315  1315 T 20:39:04 xclock -geometry 200x200 -update 2
ehab      1319  1315  1315 S 20:39:15 xeyes
ehab      1322  1315  1315 T 20:39:32 /bin/sh ./myclock out1
ehab      1350  1315  1315 S 20:39:48 /bin/sh ./myclock out2
ehab      1880  1322  1315 Z 20:42:50 [sleep] <defunct>
ehab      1932  1350  1315 S 20:43:32 sleep 2
...    ...  ...     ..    .  ...      ...
```

5. In the first window, we list all jobs then resume the suspended jobs.

```
a1jobs[1315]: list
0: (pid=    1318, cmd= xclock -geometry 200x200 -update 2)
1: (pid=    1319, cmd= xeyes)
2: (pid=    1322, cmd= myclock out1)
3: (pid=    1350, cmd= myclock out2)

a1jobs[1315]: resume 0
a1jobs[1315]: resume 2
```

6. In the second window, we use `ps` to see the changes.

```
USER       PID  PPID  PGID S  STARTED CMD
...   ...  ...    ..    .  ...      ...
ehab      1315  1087  1315 S 20:38:35 a1jobs
ehab      1318  1315  1315 S 20:39:04 xclock -geometry 200x200 -update 2
ehab      1319  1315  1315 S 20:39:15 xeyes
ehab      1322  1315  1315 S 20:39:32 /bin/sh ./myclock out1
ehab      1350  1315  1315 S 20:39:48 /bin/sh ./myclock out2
ehab      2139  1322  1315 S 20:45:51 sleep 2
ehab      2146  1350  1315 S 20:45:52 sleep 2
...   ...  ...    ..    .  ...      ...
```

7. In the first window, we exit `a1jobs`.

```
a1jobs[1315]: exit
    job 1318 terminated
    job 1319 terminated
    job 1322 terminated
    job 1350 terminated

real:  455.04 sec.
user:    0.00 sec.
sys:    0.00 sec.
child user:    0.00 sec.
child sys:    0.00 sec.
```

8. Finally, in the second window, we use `ps` to see the changes. In this example, all processes that belong to all launched jobs are properly terminated (this may not always be the case!).

Implementation Remarks

- To run a program, `a1jobs` forks a child process and then uses one of the `exec` function calls. It is recommended to use `execlp`. The following are examples of using the function:

```
execlp("./myclock", "myclock", "out1", (char *) NULL);
execlp("xclock", "xclock", "-geometry", "200x200" ,"-update", "1", (char *) NULL\
);
```

The following call is a wrong way to pass just two arguments to the `xclock` program.

```
execlp("xclock", "xclock", "-update", "2", "", "", (char *) NULL);
```

The `a1mon` Program

The `a1mon` program is invoked by the command line: "`a1mon` *target_pid* [*interval*]", where

- *target_pid* is the *pid* of some process running on the same workstation (e.g., an `a1jobs` process) to be monitored
- *interval* is an integer that specifies a time interval in seconds. This argument is optional. If omitted from the command line it assumes a default value of 3 seconds

The program monitors all descendant processes included in the tree rooted at the specified target process. If the monitor detects that the target process has terminated, it performs a cleanup by terminating all processes in the tree. In more detail, the program performs the following steps.

1. Use `setrlimit()` to set a limit on the CPU time (e.g., 10 minutes).
2. Run the main loop of the program. In each iteration, the program
   (a) Increments an iteration counter, and prints a header like the following one:
   ```
   a1mon [counter= 9, pid= 3228, target_pid= 3114, interval= 3 sec]:
   ```
   (b) Use `popen` to execute the `ps` program in the background:
   ```
   ... = popen("ps -u $USER -o user,pid,ppid,state,start,cmd --sort start", "r");
   ```
   (c) Read, display, and process each line produced by `popen`. After reading all lines, the program uses function `pclose()` to close the pipe.

   (d) Based on the data obtained during the iteration, the program decides if the target process is still running. If the target process has terminated, the program terminates each process in the process tree rooted at the target process. The program then terminates.

   (e) The next iteration, if any, is delayed by the number of seconds specified by the `interval` variable.

Example. In this example, we open three terminal windows on the same lab workstation.

1. In the first window, we run `a1jobs` and start some jobs, as in the previous example.
2. In the second window, we run `a1mon` to monitor `a1jobs`:

```
a1mon [counter= 9, pid= 3228, target_pid= 3114, interval= 3 sec]:
USER       PID  PPID S  STARTED CMD
...     ... ...  . ... ...
ehab      3114  2506 S 23:26:25 a1jobs
ehab      3115  3114 S 23:26:37 xclock -geometry 200x200 -update 2
```

```
ehab      3116  3114 S 23:26:43 xeyes
ehab      3117  3114 S 23:26:51 /bin/sh ./myclock out1
ehab      3132  3114 S 23:27:00 /bin/sh ./myclock out2
ehab      3228  2517 S 23:27:38 a1mon 3114
ehab      3290  3117 S 23:28:01 sleep 2
ehab      3292  3132 S 23:28:02 sleep 2
ehab      3293  3228 S 23:28:03 sh -c ps -u $USER -o user,pid,ppid,state,start,cmd --s
ehab      3294  3293 R 23:28:03 ps -u ehab -o user,pid,ppid,state,start,cmd --sort sta
...    ... ...  . ... ...
--------------------
List of monitored processes:
[0:[3115,xclock], 1:[3116,xeyes], 2:[3117,/bin/sh], 3:[3132,/bin/sh], ...
```

3. In the third window, we issue `kill -SIGKILL 3114` to terminate the `a1jobs` program.
4. In the second window, we record the actions taken by the `a1mon` program:

```
a1mon [counter= 9, pid= 3228, target_pid= 3114, interval= 3 sec]:
USER        PID  PPID S  STARTED CMD
...    ... ...  . ... ...
...    ... ...  . ... ...
--------------------
List of monitored processes:
[0:[3115,xclock], 1:[3116,xeyes], 2:[3117,/bin/sh], 3:[3132,/bin/sh], ...

a1mon: target appears to have terminated; cleaning up
terminating [3115, xclock]
terminating [3116, xeyes]
terminating [3117, /bin/sh]
terminating [3132, /bin/sh]
terminating [3896, sleep]
terminating [3898, sleep]
exiting a1mon
```

More Details

1. This is an individual assignment. Do not work in groups.
2. Only standard include files and libraries provided when you compile the program using `gcc` or `g++` should be used.
3. **Important:** you **cannot** use `system()` to implement any of the above functionalities. You can use `popen()` to run the `ps` program, as described in the section on `a1mon`. No other use of `popen` is permitted.
4. Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In such cases, you should make reasonable design decisions that do not contradict what we have said and do not significantly change the purpose of the assignment. Document such design decisions in your source code,

and discuss them in your report. Of course, you may ask questions about this assignment (e.g., in the Discussion Forum) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.

5. When developing and testing your program, **make sure you clean up all processes before you logout of a workstation.** Marks will be deducted for processes left on workstations.

Deliverables

1. All programs should compile and run on Linux lab machines (e.g., ug[00 to 34].cs.ualberta.ca)
2. Make sure your programs compile and run in a fresh directory.
3. Your work (including a Makefile) should be combined into a single tar archive 'submit.tar' or 'submit.tar.gz'.
   (a) Executing 'make' should produce the `a1jobs` and `a1mon` executables
   (b) Executing 'make clean' should remove unneeded files produced in compilation.
   (c) Executing 'make tar' should produce the 'submit.tar' archive.
   (d) Your code should include suitable internal documentation of the key functions.
   (e) Typeset a project report (e.g., one to three pages either in HTML or PDF) with the following (minimal set of) sections:
   - **Objectives:** state the project objectives and value from your point of view (which may be different from the one mentioned above)
   - **Design Overview:** highlight in point-form the important features of your design
   - **Project Status:** describe the status of your project; mention difficulties encountered in the implementation
   - **Testing and Results:** comment on how you tested your implementation, and discuss the obtained results
   - **Acknowledgments:** acknowledge sources of assistance
4. Upload your tar archive using the **Assignment #1 submission/feedback** link on the course's web page. Late submission (through the above link) is available for 24 hours for a penalty of 10%.
5. It is strongly suggested that you **submit early and submit often**. Only your **last successful submission** will be used for grading.

Marking

Roughly speaking, the breakdown of marks is as follows:

15% : successful compilation of reasonably complete programs that are: modular, logically organized, easy to read and understand, and includes error checking after important function calls

05% : ease of managing the project using the makefile

40% : `a1jobs`: correctness of implementing the specified commands and features (e.g., `setrlimit`), and printing the required timing information.

30% : `a1mon`: correctness of obtaining, displaying, and processing the processes status lines, keeping track of the processes in the target process tree, detecting that the target process has terminated, and terminating its descendant processes.

10% : quality of the information provided in the project report