CMPUT 379 - Assignment #1 (10%)

Process Management Programs

Due: Friday, February 4, 2022, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience in using Unix system calls for accessing and utilizing system time values, process environment variables, process resource limits, and process management functions (e.g., fork(), waitpid(), and execl()).

Overview

Shell programs like Bourne shell (sh), C shell (csh), Bash, and Korn shell (ksh) provide powerful interactive programming environments that allow users to utilize many of the services provided by complex multiprocessing operating systems. In this assignment, you are asked to write a simple shell program, called `msh379`, in C/C++ that runs the commands described in the specifications section. To implement your shell program, 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 [APUE 3/E]
Time values	Section 1.10
chdir() and getcwd()	Section 4.23
Time and date routines	Section 6.10
getenv()	Section 7.9
getrlimit() and setrlimit()	Section 7.11
fork(), waitpid(), execl()	Chapter 8: Process Control
times()	Section 8.17
kill()	Section 10.9

Program Specifications

The `msh379` 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 implementation 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 `"msh379 [pid]:   "`, where `pid` is the process number of the running instance of the program. Subsequently, the program tries to execute 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 [APUE 3/E], `msh379` 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 `msh379` in executing step (3), and

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

Command Lines

The program should maintain information of at most `NTASK` (= 32) *accepted* tasks. A task is accepted if the `run` command described below can successfully run an associated user specified program. Accepted tasks are assigned indices $0, 1, \cdots, \mathtt{NTASK} - 1$, in this order. A task terminated by the `terminate` command, or finished execution on its own, continues to keep its index until the `msh379` program exits.

The program stores (at least) the following information for each accepted task: its index, pid of the task's head process, and the command line used to start the task. The program should handle the following user issued commands.

1. **cdir pathname:** Change the current working directory to the directory specified by the absolute or relative `pathname`. The specified `pathname` may begin with a shell environment variable that needs expansion. For example, the `pathname "$HOME/c379"` requires the expansion of environment variable `HOME` (the `'$'` is not part of the variable name). Implement this command using function `chdir()` (see the above table).

2. **pdir:** Print the current working directory. Implement this command using function `getcwd()` (see the above table).

3. **lstasks:** List all accepted tasks that have not been explicitly terminated by the user. Each entry of the listing contains (at least) the stored index, pid, and the command line associated with the task. Note that some of the listed tasks may have already run to completion, and thus, the corresponding process (or processes) has disappeared from the system.

4. **run pgm arg1 $\cdots$ arg4:** Fork a process to run the program specified by `pgm`, using the given arguments (at most 4 arguments may be specified). A task is considered accepted if `msh379` has successfully launched the process. The process running `pgm` is the head process of the task (depending on `pgm`, executing the task may result in creating more than one process). As mentioned above, `msh379` accepts at most `NTASK` tasks (the count includes tasks that the user has explicitly terminated).

5. **stop taskNo:** Stop (temporarily) the task whose index is `taskNo` by sending signal $SIGSTOP$ to its head process.

6. **continue taskNo:** Continue the execution of the task whose index is `taskNo` by sending signal $SIGCONT$ to its head process.

7. **terminate taskNo:** Terminate the execution of the task whose index is `taskNo` by sending signal $SIGKILL$ to its head process.

8. **check target_pid:** Here, $target\_pid$ is the pid of some process running on the same workstation; this can be a `msh379` process, or any other process that belongs to the user. If the checked process is not running (e.g., defunct), the command reports this fact and does nothing further. Else, if the process is running, the command displays all descendant processes in the tree rooted at the specified target process. In more detail, the check command performs the following steps.

   (a) Use `popen` to execute the `ps` program in the background:

   ```
   ... = popen("ps -u $USER -o user,pid,ppid,state,start,cmd --sort start", "r");
   ```

   (b) Read, display, and process each line produced by `popen`. After reading all lines, the program uses function `pclose()` to close the pipe.

   (c) Based on the data obtained from the above steps, the command decides whether the target process is still running, or it has terminated. In the earlier case, the command outputs information about each process in the process tree rooted at the target process.

9. **exit:** Terminate the head process of each accepted task that has not been explicitly terminated by the `terminate` command. Then exit the main loop.

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

Examples. Example output will be posted on eClass.

Implementation Remarks

- To run a program, `msh379` forks a child process and then uses one of the `exec` function calls. It is recommended to use `execlp`. The following examples illustrate its use.

  – To run a program called `myclock` with one argument using, e.g., the command line: `'run myclock out1'` use

  ```
  execlp("./myclock", "myclock", "out1", (char *) NULL);
  ```

  – To run a program called `xclock` with 4 arguments using, e.g., the command line: `'run xclock -geometry 200x200 -update 1'` use

  ```
  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);
  ```

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 `check` command. 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' or 'make `msh379`' should produce the `msh379` executable.
   (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 (to what degree the program works as specified) 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

70% : correctness and completeness of implementing the specified commands and features

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