CMPUT 379 - Assignment #1 (10%)

Process Management Programs

Due: Monday, February 6, 2023, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience with Unix process sates, process system time values, and process management functions (e.g., fork(), waitpid(), and execl()). The assignment refers to some of the following topics:

function	Reference in Advanced Programming in the Unix Environment [APUE 3/E]
Time values	Section 1.10
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

Part 1

The following questions provide warm up exercises on the use of the C language and some Unix system calls for developing the required system programs. Typeset and Hand in the answers in your submitted report. **Note:** some questions ask for writing C-code fragments, marks will be deducted for unnecessarily long or complex codes.

1. Assume we have a variable declaration `'PCB_t pcb'` where `PCB_t` defines a C-struct with various members (e.g., integers, character arrays, etc.). Write a C-code fragment that uses the standard C library function `memset` to initialize variable `pcb` to all zeros.

2. Often, a program needs to read a sequence of text lines (say, from the `stdin`), and process each line in a certain way. A simple C-fragment to read the lines is:

```
#define MAXLINE  128
#define MAXWORD  20
char buf[MAXLINE];
while(fgets(buf,MAXLINE,stdin) != NULL) { ... }
```

   However, `fgets` does not remove a terminating newline character `'\n'` from `buf` (which may be undesirable). Assuming that each line has at most `MAXLINE` $-$ 2 characters, and each line terminates with a newline, write a C-fragment to delete the terminating newline character from `buf`.

3. In the above question scenarios, after reading a text line in `buf`, one may need to split each input line into tokens, each delimited by a character stored in a string of *field separators*. For example, the string `char WSPACE[]= "\n \t"` may be used to define 3 field separators (the newline, space, and tab characters). So, for example, one may want to split an input string like

`timeout 10 do_work bash.man 7` into 5 tokens separated by one (or more) of the above white space characters.

Explain how to use the standard C library function `strtok` to tokenize an input string, called `inStr`, around separators in `WSPACE`, while keeping track of the number of obtained tokens in variable `count`. Assuming that each token has at most `MAXWORD` $-$ `1` characters, specify a simple data structure for storing the obtained tokens.

4. When encountering an error, many Unix system calls return -1 and set the integer `errno` to a value indicating the reason of the error (cf. Section 1.7 of [APUE 3/E]). Give a C-code fragment to print on the `stdout` (rather than the `stderr`) an error message string corresponding to the `errno` value.

Part 2

■ **Background.** Operating systems maintain key information about each admitted process to the system, including its PID, PPID, and process state information. On lab machines, such information can be obtained by using various tools and command-line arguments, e.g., one may issue the command

```
% ps -u $USER -o user, pid, ppid, state, start, cmd --sort start
```

This part of the assignment aims at exploring some possible updates done on such information during the lifetime of a process.

The description below uses a Bash script (posted online), called `myclock`. You need to download the script into your work directory and make it executable using `"chmod +x myclock"`. When invoked with no argument (e.g., using `"./myclock"`), the script loops indefinitely printing the date and time and then delaying for sometime before starting the next iteration. When invoked with one argument, e.g., using `"./myclock out1"`, the script directs the output to disk file `out1`. To terminate the resulting process, send a suitable signal to it.

■ **Program Specifications.** You are asked to write a C/C++ program, called `a1p2`, that can be invoked using

```
% a1p2 (w|s)
```

where we specify either the argument `'w'` (for wait), or `'s'` (for sleep). The steps done by the program are as follows:

1. After reading the command-line argument, the program forks a child process to execute the command-line `"./myclock out1"`. To do this, one may use the `execlp` function as follows:

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

   Notes:

   - The dot `'.'` character may be omitted if the current directory `'.'` is part of the `PATH` environment variable.
   - As explained in Section 8.10 of the [APUE 3/E], one can also use `execv` instead of `execlp`. The `execv` requires an array of pointers to arguments.

2. If argument `'w'` is specified, the program calls `waitpid` to wait for the termination of the child process.

3. Else, if `'s'` is specified, the program sleeps for 2 minutes (an interval that is long enough to obtain and record the needed data described below). The program then exits normally.

■ **More Details.** Here, we perform the following experiments, fill the entries of the table below (some values may not exist), and answer the posed questions.

> **Important:** After each experiment, terminate any leftover process created by the steps.

Experiment 1: To obtain the required values, start two terminal windows logged onto the same lab workstation. In the first window, run the program using `"a1p2 w"`. In the second window

- issue a `ps` command to obtain the information needed to fill rows 1 to 3 below.
- Subsequently, issue a command to terminate the `'a1p2'` process and then issue a `ps` command to fill rows 4 to 6.

Command-line: `"a1p2 w"`				
	PID	**PPID**	**State**	**CMD**
1. (a1p2 process)				
2. (child process)				
3. (grand child (if any))				
4. (a1p2 process)				
5. (child process)				
6. (grand child (if any))				

Experiment 2: Similar to the above experiment, in the first window, run the program using `"a1p2 s"`. In the second window issue a `ps` command to obtain the information needed to fill rows 1 to 3 of a copy of the above table (labelled with **command-line: "a1p2 s"**). Subsequently, issue a command to terminate the child (`myclock`) process, and within the 2-minute sleep interval issue a `ps` command to fill rows 4 to 6.

■ **Deliverables.** In addition to submitting the program file, for each experiment, submit a completed table in your report. Explain

1. any relation between the recorded IDs

2. information obtained about the process state field.

Part 3

■ **Background.** The UNIX operating system keeps track of certain process times of each running process (and under some conditions) its terminated children processes. See Sections 1.10 and 8.17 of the [APUE 3/E] book. This part of the assignment aims at exploring the use of such facilities further.

In addition to the `myclock` script used above, we also use a program (posted online), called `do_work.c`, intended to consume both *user* and *system* CPU time when running. Thus, enabling experimenting with various system calls and UNIX tools that measure the CPU resources consumed by a process. For further information, see the program's internal documentation.

■ **Program Specifications.** You are asked to write a C/C++ program, called `a1p3`, that can be invoked using

```
% a1p3 (0|1|-1) < inputFile
```

where we specify an integer argument (either 0, 1, or -1), and use shell file redirection to pass input to the program (so, the program can read the input from the `stdin` file stream). The steps done by the program are as follows:

1. Upon start, the program calls function `times()` to record the *user* and *system* CPU times of the current process (and its terminated children).

2. After reading the command line argument, the program invokes a main loop to read (from the `stdin`) the text lines of the `inputFile`. The file's format is as follows.

   - A line starting with `'#'` is a comment line (skipped)
   - Empty lines are skipped. For simplicity, an empty line has a single `'\n'` character.
   - Else, a line specifies a command line to execute. Each such line has at most 5 strings. The first string is taken as a name of an executable file, followed by at most 4 command line arguments.

3. If the input line is a command line, the program

   (a) forks a child process that calls one of the `exec` functions to execute the specified executable file. If the `exec` system call fails, the child process prints an error to this effect and exits.

   (b) The main program stores the following information for each command line processed (whether executed successfully or failed): the PID of the forked child process, and the corresponding command line text.

   Note: At most `NPROC` $=$ `5` command lines (or forked processes) exist in the input file.

4. Upon exiting the main loop, the program prints information about the commands processed. Subsequently, the program processes the command line argument as follows:

   (a) If 0 is specified, the main program does not wait for any child and proceeds to step 5.

   (b) If 1 is specified, the program calls `waitpid` to wait for the termination of any of its children processes and then proceeds to step 5.

   (c) If -1 is specified, the program calls `waitpid` as many times as required to wait for the termination of all children processes and then proceeds to step 5.

5. Next, the program calls function `times()` again to obtain the user and system CPU times.

6. Using a setup and output format similar to the program in Figure 8.31 of [APUE 3/E], the program computes and prints the following times in **seconds**:

   (a) the total time elapsed between steps 1 and 5,

   (b) the reported **user** and **system** CPU times used by the main program, and

   (c) the reported **user** and **system** CPU times of the children processes.

Implementation Remarks

- Here is another example of using `execlp`: to run a program called `xclock` with 4 arguments `'xclock -geometry 200x200 -update 1'`, use

  ```
  execlp("xclock","xclock","-geometry","200x200","-update","1", (char *) NULL);
  ```

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

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

- For convenience, you may assume the following limits in your program:

`MAXLINE`	128	// Max # of characters in an input line
`MAX_NTOKEN`	16	// Max # of tokens in any input line
`MAXWORD`	20	// Max # of characters in any token
`NPROC`	5	// Max # of commands in a test file

Examples.

Example output will be posted on eClass.

Deliverables.

In addition to submitting the program file, create (and submit) a test file that has `NPROC` $= 5$ command lines. Each command line should run at least 5 seconds of real time. Run the program with the three possible argument values: 0, 1, and -1. Comment on the obtained CPU times in each case.

More Details (all parts)

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. 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.

4. 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 (all parts)

1. All programs should compile and run on the lab machines (e.g., ug[00 to 34].cs.ualberta.ca) using only standard libraries.

2. Make sure your programs compile and run in a fresh directory.

3. Your work (including a Makefile and test files) should be combined into a single tar archive **'your_last_name-a1.tar'** or **'your_last_name-a1.tar.gz'**.

   (a) Executing 'make a1p2' or 'make a1p3' should produce the corresponding executable.

   (b) Executing 'make clean' should remove unneeded files produced in compilation.

   (c) Executing 'make tar' should produce the above '.tar' or '.tar.gz' archive.

   (d) Your code should include suitable internal documentation of the key functions. If you use code from the textbooks, or code posted on eclass, acknowledge the use of the code in the internal documentation. Make sure to place such acknowledgments in close proximity of the code used.

   (e) Typeset a project report (e.g., one to three pages either in HTML or PDF) with the following (minimal set of) sections:

   – Answers to the questions posed in each part

   – **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:

05% : ease of managing the project programs using the submitted Makefile

15% : Part 1 answers

30% : Correctness, testing and results of Part 2

50% : Correctness, testing and results of Part 3
