CS421 Introduction to Operating System
Fall 2006
Project #1
Implementing Concurrency using processes and threads

Bina Ramamurthy

September 7, 2006

1 Objective

To familiarize the students with:

- Programming in C/C++.
- Unix system/library calls, especially *fork, pipe, exec, wait, kill* and *exit.*
- Concurrent execution of user and system processes.
- Using POSIX *threads* for concurrency.

2 Problem Statement

This project is to be developed in several small steps to help you understand the concepts better.

1. (10 points) Implement a C or C++ program that takes three command line arguments, A, B and C, computes and outputs the sum of A and B, and the difference of (A+B) and C.

   Write two functions one for *addition* and another for *subtraction.* You are required to invoke two child processes to solve this problem. However, since the problem is such that calculating $(A + B) - C$ is dependent on the result of (A+B), the two processes will need to communicate with each other. Use *fork* to spawn the child processes. The first child should calculate and print (A+B), and then use a *pipe* to pass that value to the second child, which will subtract C from that value, and print the result.

2. (5 + 5 points) Write a program that uses UNIX fork system call to create two processes. The first process will be an *exec*ed instantiation of the "cat" command and it should pass its output to the standard output. The second process will be an *exec*ed instantiation of the "wc" command, and it should pass its output to the standard output too. You may use any other suitable commands instead of *cat* and *wc*, if you prefer. Write your program in two versions:

   (a) *Concurrent Mode.* The parent process will fork two child process almost concurrently (one immediately after another). On large input sets, you should notice that the output of the two child processes is interleaved. To make this more observable choose the data file that contains no numbers or digits. Then the output from "cat" will be alphabetical and the output from "wc" will be numerical. Highlight the output from "wc".

   (b) *Sequential Mode.* The parent process will force a sequential execution of the two child processes, to prevent interleaving of outputs. In other words the output should be the concatenation from the two child processes.

3. (10 points) Write a shell-like program that illustrates how UNIX spawns processes. This simple program will provide its own prompt to the user, read the command from the input and execute the command. It is sufficient to handle just "argument-less" commands, such as `ls` and `date`.

4. (5 points) Make the mini-shell (from the previous part) a little more powerful by allowing arguments to the commands. For example, it should be able to execute commands such as `more filename` and `ls -l /tmp` etc.

5. (10 points) Implement the simple addition and subtraction problem (problem 1 above) using POSIX *threads*. Use shared variables as the means of communication between the threads.

6. (5 points) When using system or library calls you have to make sure that your program will exit gracefully when the requested call cannot be carried out. For every system call make sure you provide this facility. Show that this feature works in each of the first 5 programs above by inputting illegal input.

```
if (do sys_call) fails
 print an error message and exit with error code.
endif
```

3 Implementation Details

1. In general, the execution of any of the programs above will is carried out by specifying the executable program name followed by the command line arguments. The command line arguments could be A, B, and C or could be file names for *cat* or *wc*.

2. See the man pages for more details about specific system or library calls and commands: UNIX fork(2), pipe(2), execve(2), execl(3), execlp(3), cat(1), wait(2) etc.

3. One of the dangers of learning about forking processes is leaving unwanted processes active and wasting system time. Make sure each process terminates cleanly when processing is completed. Parent process should wait until the child processes complete, print a message and then quit. Do not leave behind any processes. See kill(2), ps(1), ps(1b), sps(1).

4. Your program should be robust. If any of the calls fail, it should print error message and exit with appropriate error code. Always check for failure when invoking a system or library call. By convention, most UNIX calls return a value of negative one (-1) in case of an error (but check the `RETURN VALUES` section of each man page for details), and you can use this information for a graceful exit. Use cerr, perror(3), or strerror(3) library routines to print error message when appropriate.

4 Material to be Submitted

1. Submit online the **source code each of the programs**. Use meaningful names for the file so that the contents of the file is obvious.

2. Only internal documentation is needed. Please state clearly the purpose of each program at the start of the program. Add comments to explain your program. (-5 points, if insufficient.)

3. Test runs: It is very important that you show that your program works for all possible inputs. Submit online **a single typescript** file clearly showing the working of all the programs for correct input as well as graceful exit on error input.

4. Submit a README file in case there are any major deviation in the way we prepare the executables.

5 Due Dates

9/23/06 (Saturday), submit on-line before midnight.