CS421 Introduction to Operating System
Spring 2007
Project #1
Concurrency using processes and threads

Bina Ramamurthy

January 24, 2007

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) The goal of this part is to be able to spawn processes to realize basic concurrent processing and to be able to pass data between the processes spawned. For the computation you will use a simple addition of number 0 to 30000. You will accomplish this task using two alternative strategies:
   1. Spawn a single child process that will perform the addition and return the result to the child using pipe. Parent prints the result. 2. Spawn three child processes from the parent and let each perform one third of the addition (1 to 10000, 10001 to 20000, 20001 to 30000 respectively). Child1 computes its workload and passes the result top the child2, which then accumulates the result and passes it to the child3 that sends the final accumulated sum to the parent. Parent process prints the result.
   Use *fork* to spawn the child processes. Use a *pipe* to pass that value to the between children and to parent.

   Execution time monitoring: For each startegy above, compute the time for performing the calculation. To make it reasonable to analyze the time interval, we can assume that each computation takes some extra time to finish. Use sleep, nanosleep or other sleep related function to simulate extar time. Compare the times for the two strategies above.

2. (5 + 5 points) The goal of this part to illustrate *exec* command and to study sequential and concurrent processes. Write a program that uses UNIX fork system call to create two processes. Exec from the children appropriate unix commands that will clearly provide observable differences in concurrent and sequential mode.

   Write your program in two versions:

   (a) *Concurrent Mode.* The parent process will fork two child process almost concurrently (one immediately after another).

(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. (5 points) Add to the mini-shell ability to execute command lines with commands connected by pipes. Example: `ls -l | wc`

6. (10 points) Implement the simple addition problem (problem 1 above) using POSIX *threads*. Use shared variables as the means of communication between the threads.

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.

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. 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.

4. 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.

5. 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. Submit a README file that lists the files you have submitted along with a one phrase explanation.

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.

5 Due Dates

2/17/2007 (Saturday), submit on-line before midnight.