CSE4/521 Introduction to Operating System Fall 2009
Project 1: Realizing Concurrency using Unix processes and threads

Objectives:

Learn to:

- Design and develop systems programs using C/C++
- Effectively use Unix system calls for process control and management, especially, *fork, exec, wait, pipe and kill system call API.*
- Concurrent execution of processes.
- Use Posix Pthread and OpenMp library for concurrency.

Problem Statement:

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

1. (System calls) Write a C program that makes a new *encrypted* copy of an existing file using ***system calls*** for file manipulation. For encryption use secret key encryption. The Key will specified as a command line parameter. Use a suitable encryption function to encrypt the source file content using the key. The names of the two files, source and the destination are to be specified as command line arguments. Open the source file in read only mode and destination file in read/write mode. While the main function will carry out file opening and closing, a separate encrypt function needs to be written to do the actual encrypting and copying. {MyEncryt.c → MyEncrypt)

2. (fork) Write a C program that creates a new process to encrypt a file using the MyEncrypt. This program should spawn a new process using ***fork*** system call. Then use ***execl*** to execute MyEncrypt program. The source and destination file names presented as command-line arguments should be passed to *execl* as system call arguments. The main process waits for completion of encrypt operation using *wait* system call. (ForkEncrypt.c → ForkEncrypt)

3. (pipe) Write a C program that forks two processes one for reading from a file (source file) and the other for writing (destination file) into. These two programs communicate using ***pipe*** system call. Once again the program accomplishes encrypt files, the names of which are specified as command-line arguments. (PipeEncrypt.c → PipeEncrypt)

4. (timing) Use various system calls for ***time*** to compare the three versions of the file encrypt programs as specified above. Observe that you may not see any significant difference since these are tiny programs.

5. (shell) 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***. (MinShell.c→ MiniShell)

6. (passing and parsing arguments) 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. (MoreShell.c → MoreShell)

7. (redirecting output using dup) Add to the MoreShell ability to execute command lines with commands connected by pipes. Use ***dup*** system call to redirect IO. Example: ls -l | wc . (DupShell.c DupShell).

8. (Multi-threading using Posix threads) Use Pthreads library to write a solutions for (i) computation of PI using Manto Carlo numerical method, (ii) Computation and display of a Mandelbrot set.

9. (OpenMp) Repeat step 8 but this time using OpenMp, an API for parallel programming. You can use the OpenMp available on Microsoft Visual Studio.

Implementation Details:

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.

1. 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. 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.
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.
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.

Material to be submitted:

1. Compress the source code of the programs into Prj1.tar file. Use meaningful names for the file so that the contents of the file are obvious. A **single makefile** that makes the executables out of any of the source code should be provided in the compressed file.
2. Submit a README file that lists the files you have submitted along with a one sentence explanation. Call it Prj1README
3. 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.
4. 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.Submit for undergraduates. **submit_cse421 Prj1.tar Prj1README Prj1Script**
6.Submit for undergraduates. **submit_cse521 Prj1.tar Prj1README Prj1Script**
7.Due date: 10/10/2009, submit on-line before midnight.