CS4440 Introduction to Operating Systems
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 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 compressed copy of an existing file of bits (0's and 1's) using system calls for file manipulation. For compression use the following rules:
    a. Compress only the sequence of length >= 16 1's or 0's; for example a sequence of 26 0's is replaced by **-26-** and a sequence of 78 1's is replaced by **+78+**
    b. Sequence of 1's and 0's of length < 16 are copied as such
    c. The input file will have multiple segments of 1's and 0's separated by a space or <return>. Example input file:
    1111111111100001010101010101010101010101010101010101010101010101010101010101 1010101010000000000000000000000000000000001 01111111111111111111111111
    This has 3 segments as shown by the different colors.
    The compressed output file for this will be:
    1111111111100001010101010101010101010101010101010101010101010101010101010101 101010101-36-1 0+24+

    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 compress function needs to be written to do the actual compressing and copying. {MyCompress.c → MyCompress} Your program should work with the large test file you can come up with. For simplicity in IO processing assume the input file of 1's and 0's is a file of characters. The output file will have characters of 1's and 0's as well as the compressed information as numerical value.
    While you may use the data given above for prototyping, you will need fairly large data files to get some realistic time measurements.

2. *(decompress)* Write a C program that inputs a compressed file as created above and decompresses it into an output file. (MyDecompress.c → MyDecompress)

3. (fork) Write a C program that creates a new process to compress a file using the MyCompress. This program should spawn a new process using ***fork*** system call. Then use ***execl*** to execute MyCompress 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 compress operation using *wait* system call. (ForkCompress.c → ForkCompress)

4. (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 compress files, the names of which are specified as command-line arguments. (PipeCompress.c → PipeCompress)

5. (concurrency) Write a concurrent version of the compression described above in problem #1. Create very large file, divide the file into equal chunks (say n) and allocate the work to n processes forked. Then assemble the result of each process and output it. You will have an input data file and output compressed data file. (ParFork.c → ParFork)

6. (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)

7. (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)

8. (redirecting output using dup or dup2) Add to the MoreShell ability to execute command lines with commands connected by pipes. Use ***dup2*** or ***dup*** system call to redirect IO. Example: ls -l | wc . (DupShell.c DupShell).

9. (Multi-threading using Posix threads) Use Pthreads library to write a solution for question 5 above using the thread model for concurrency. (ParThread.c → ParThread)

10. Compare the times for (i) sequential version (ii) forked (process) version and (iii) pthread version. Use a large data file so that times are significant. Record this information (inference) in your README file.

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.

2. 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 (or Prj1.zip) 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 **MUST** 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. This file will also have time information inferred from problem #10.

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 on CSNS.**