CS356 Operating System Projects Spring 2015

Project 1: Linux Process Concurrency and Synchronization

Objectives:

- Design and develop systems programs using C/C++
- Effectively use Linux 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.
- Solve inter-process communication problems during concurrent execution of processes.
- Use semaphores.

Problem Statement:

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

1. Write a C program that makes a new copy of an existing file using ***system calls*** for file manipulation. 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 copy function needs to be written to do the actual copying. Copying can be done in blocks of suitable size. (MyCopy.c → MyCopy)

2. Write a C program that creates a new process to copy the files using the MyCopy. This program should spawn a new process using ***fork*** system call. Then use ***execl*** to execute MyCopy 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 copy operation using ***wait*** system call. (ForkCopy.c → ForkCopy)

3. 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 copying files, the names of which are specified as command-line arguments. (PipeCopy.c → PipeCopy)

4. Use various system calls for ***time*** to compare the three versions of the file copy programs as specified above.

5. Write a shell-like program that illustrates how Linux 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***. (MyShell.c → MyShell)

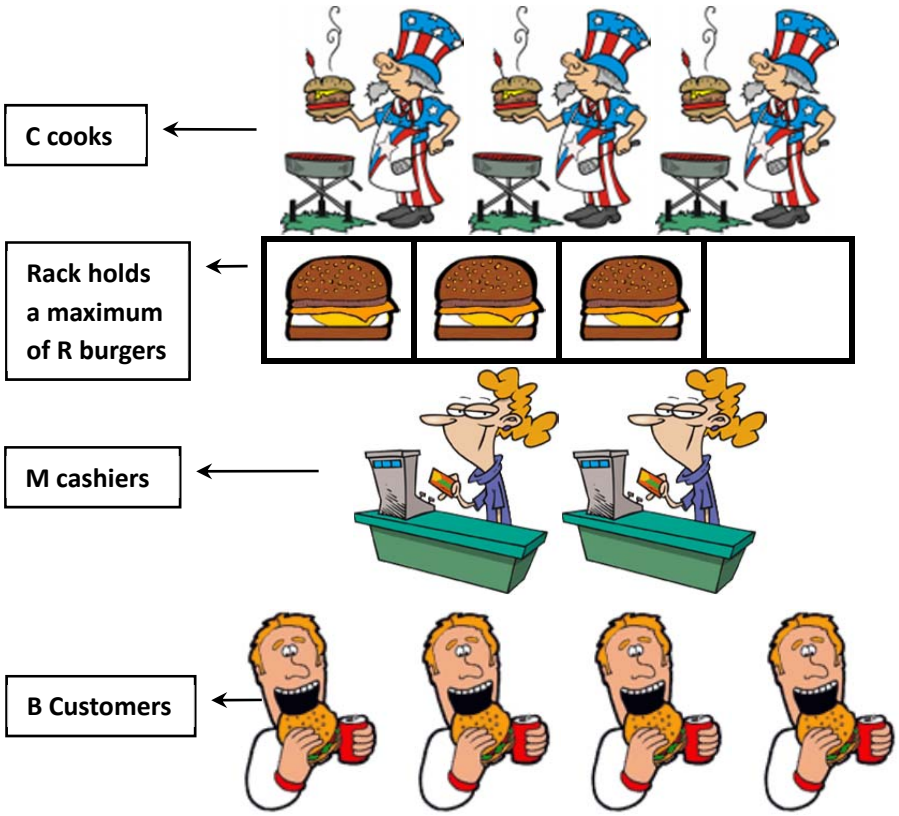
6. Make the mini-shell (from the previous part) a little more powerful by allowing arguments to the commands. For example, it should be able to execute commands such as ***more filename*** *and* ***ls –l ~/tmp*** etc. (MoreShell.c → MoreShell)

7. Add to the mini-shell 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. Mergesort is an effcient algorithm for sorting large data set. Implement mergesort (i) using a single threaded program (MergesortSingle.c → MergesortSingle) and (ii) then with multiple threads (one of each divided component) and compare the times of the two implementations. Use **Pthreads** for realizing the threads. (MergesortMulti.c → MergesortMulti)

9. **Burger Buddies Problem**: Design, implement and test a solution for the IPC problem specified below. Suppose we have the following scenario:
   - Cooks, Cashiers, and Customers are each modeled as a thread.
   - Cashiers sleep until a customer is present.
   - A Customer approaching a cashier can start the order process.
   - A Customer cannot order until the cashier is ready.
   - Once the order is placed, a cashier has to get a burger from the rack.
   - If a burger is not available, a cashier must wait until one is made.
   - The cook will always make burgers and place them on the rack.
   - The cook will wait if the rack is full.
   - There are NO synchronization constraints for a cashier presenting food to the customer.

   Implement a (concurrent multi-threaded) solution to solve the problem and test it thoroughly. Show output runs that illustrate the various possibilities of the set up. (BurgerBuddies.c → BBC)



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. Use Ubuntu Linux to test your programs.
2. See the man pages for more details about specific system or library calls and commands: 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.

Material to be submitted:

1. Compress the source code of the programs into **Prj1+StudentID.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. Enclose a README file that lists the files you have submitted along with a one sentence explanation. Call it **Prj1README**.
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. Send your **Prj1+StudentID.tar** file to cs356.sjtu@gmail.com.
5. Due date: **Apr. 24, 2015**, submit on-line **before midnight**.