Project 2: Inter-Process Communication and Synchronization

March 26, 2023

1 Objectives

- Solve inter-process communication problems during concurrent execution of processes.
- Use Posix Pthread library for concurrency.
- Use semaphores.

2 Problem Statement

2.1 Stooge Farmers Problem

2.1.1 Basic requirement

Larry, Moe, and Curly are planting seeds. Larry digs the holes. Moe then places a seed in each hole. Curly then fills the hole up. There are several synchronization constraints:

- Moe cannot plant a seed unless at least one empty hole exists, but Moe does not care how far Larry gets ahead of Moe.
- Curly cannot fill a hole unless at least one hole exists in which Moe has planted a seed, but the hole has not yet been filled. Curly does not care how far Moe gets ahead of Curly.
- Curly does care that Larry does not get more than MAX holes ahead of Curly. Thus, if there are MAX unfilled holes, Larry has to wait.
- There is only one shovel with which both Larry and Curly need to dig and fill the holes, respectively.

Design, implement and test a solution for this IPC problem, which represents Larry, Curly, and Moe. Use semaphores as the synchronization mechanism.

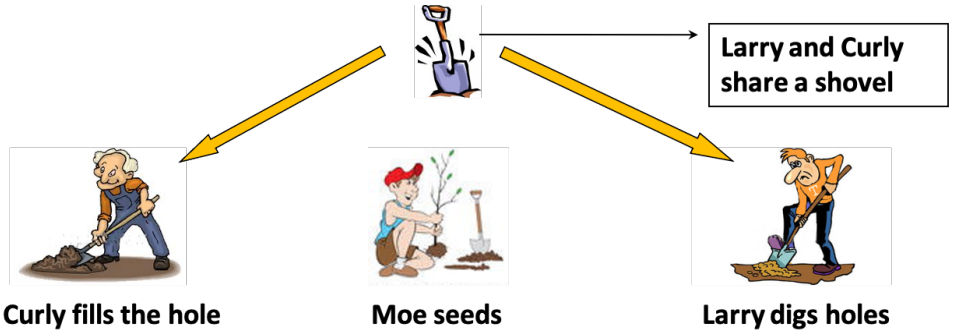

Figure 1: Stooge Farmers Problem

2.1.2 Advanced requirement

Add some additional constraint constraints so that there is no starvation in this problem.

2.1.3 More details of requirement

1. Source file name: LarryCurlyMoe.c
2. Executable file: LCM
3. LCM Command line: ./LCM Maxnum
4. Maxnum: Max number of unfilled holes.

2.2 The Faneuil Hall problem

2.2.1 Basic requirement

There are three kinds of threads: immigrants, spectators, and one judge. Immigrants must wait in line, check in, and then sit down. At some point, the judge enters the building. When the judge is in the building, no one may enter, and the immigrants may not leave. Spectators may leave. Once all immigrants check in, the judge can confirm the naturalization and immigrants who sitted down can be confirmed. After the confirmation, the immigrants who are confirmed by the judge swear and pick up their certificates of U.S. Citizenship, while the others wait for the next judge. The judge leaves at some point after the confirmation. Spectators may now enter as before. After immigrants get their certificates, they may leave.

To make these requirements more specific, let's give the threads some functions to execute and put constraints on those functions.

- Immigrants must invoke *enter*, *checkIn*, *sitDown*, *swear*, *getCertificate* and *leave*.
- The judge invokes *enter*, *confirm* and *leave*.
- Spectators invoke *enter*, *spectate* and *leave*.
- While the judge is in the building, no one may *enter* and immigrants may not *leave*.
- The judge can not *confirm* until all immigrants, who have invoked *enter*, have also invoked *checkIn*.

2.2.2 Advanced requirement

Add some additional constraint constraints so that there is no starvation in this problem.

2.2.3 More details of requirement

1. Program must work in an infinite loop.
2. Add suitable random delays between immigrants', judges', and spectators' coming to let your program be more like the real situation.
3. Also, suitable random delays between any two successive actions of immigrants, judges, and spectators are also needed.
4. We provide a sample program to let you know the output format. Please do not disassemble it.
5. Source file name: faneuil.c
6. Executable file name: faneuil
7. faneuil Command line: ./faneuil

3 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 and GNU C Compiler to compile and debug your programs. GCC-inline-assembly is allowed.

2. See the man pages for more details about specific system or library calls and commands: sem_init(3), sem_wait(3), sem_post(3), wait(2), sleep(3), gcc(1), rand(3), etc.

3. When using system or library calls you have to make sure that your pro- gram will exit gracefully when the requested call cannot be carried out.

4. One of the dangers of learning about forking processes is leafing unwanted processes active which wastes system resources. So please make sure that each process/thread is terminated cleanly when the program exits. A parent process should wait until its child processes finish, print a message, and then quit.

5. Your program should be robust. If any of the calls fails, it should print an error message and exit with an appropriate error code. So please always check for failure when invoking a system or library call.

4 Material to be submitted

1. Compress the source code of the programs into Prj2+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 Prj2README.

2. Please state clearly the purpose of each program at the start of the program. Add comments to explain your program.

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 report to analyze the algorithm you used and show the correctness of your algorithm. You can also write some other things you want in your report.

5. Send your Prj2+StudentID.tar file to Canvas. (e.g. Prj2+5108888888.tar)

6. Due date: Apr. 17, 2023, submit on-line before midnight.