University of Calgary
Faculty of Science
Department of Computer Science

CPSC 457 - Principle Of Operating Systems
Assignment 2
Interprocess Communication using Shared Memory
Winter 2022

1 Objectives

A process is meant to achieve a specific purpose. Different processes achieve different purposes. However, it is oftentimes the case that two or more processes need to share, or exchange data, in order to perform their required duties; this is what is called *Interprocess communication*. Your assignment is to make use of the interprocess communication techniques discussed in class in order to simulate a busy day at the restaurant.

2 Background

There are two ways processes can communicate with each other: either by *sharing data* or by *sending messages*. This assignment will require you to share data between multiple processes.

One of the most famous and essential problems in interprocess communication via shared memory is the *producer-consumer* problem, also called the *bounded-buffer* problem. In such a problem, we have two processes: one called the *producer* and the other the *consumer*, and additionally, there is a *bounded buffer* in shared memory that both the producer and consumer write to and read from. The producer is in charge of adding items to the bounded buffer, and the consumer is in charge of removing them. If the buffer is full, the producer waits until a consumer takes something. If the bounded buffer is empty, the consumer waits until the producer adds something.

In this assignment, we will implement an extension to the producer-consumer problem: the *multiple-producer multiple-consumer* problem. It is identical to the *consumer-producer* problem, except as the name implies, multiple processes can produce, and multiple problems can

consume. Also, there may be more than one bounded buffer.

You will solve this problem in the context of a busy day at the restaurant, where you have two producer processes (chefs), three consumer processes (customers), and two bounded buffers (food trays).

3 Part One

Scenario

You are the owner of a locally beloved fine-cuisine restaurant, and your establishment has been reserved for an upcoming high-profile event. However, the organizers are adamant that large amounts of food not be prepared in advance, but rather, your chefs should cook fresh entrées and add them to their respective food trays all throughout the duration of the event.

You will thus be in charge of organizing your two specialized chefs (represented by two producer processes) to constantly resupply two trays of entrées (represented by two bounded buffers), for three queues of hungry customers to take from (represented by three consumer processes). Each chef specializes in 2 entrées, and they strictly add items only to their respective tray (buffer):

1. Donatello specializes in non-vegan dishes:
   (a) Fettuccine Chicken Alfredo
   (b) Garlic Sirloin Steak
2. Portecelli specializes in vegan dishes:
   (a) Pistachio Pesto Pasta
   (b) Avocado Fruit Salad

A chef should randomly pick between either of their specialized entrées, and add it to the entrée tray at a random rate between 1 and 5 seconds (inclusive).

The three consumer processes are each programmed as an infinite loop() of customers:

1. The first process will represent customers that want any non-vegan dish.
2. The second process will represent customers that want any vegan dish.
3. The third process will represent customers that want one of each.

A customer needs to wait if there's not enough of any of their desired dishes. The chefs need to wait if any of their respective trays are full. After a customer takes an item, the respective customer process should wait 10-15 seconds (inclusive) before iterating again.

3.1 How to Do It

Before beginning part one, you need to understand how to create child processes, how to share data between them, and how to synchronize them such that there are no *race conditions* between them, which occur when two or more processes want to alter the shared data at the same time. In particular, we need to have an understanding of the following functionalities:

1. Creation of multiple child processes using **fork()**, and the ability to run specific functionality on each child process
2. Data sharing using **mmap()**, so that we can share the bounded-buffers and the *semaphores* between the processes
3. The utilization of **POSIX semaphores** as a tool for process synchronization.

Once we have an adequate understanding of all the previous functionalities, we are able to start on the assignment. The following steps should help you organize your work on the assignment.

3.1.1 Create the shared buffers

Before any child processes are created, we need to create the two shared bounded-buffers: one for the tray of non-vegan entrées, and one for the tray of vegan entrées. We are using **POSIX** for this assignment so you should be using **mmap()** for your shared data.

The bounded-buffers themselves should be of type **int**; a value of 0 means this slot in the entrée tray is unoccupied, 1 means it has food type one (Fettuccine Chicken Alfredo for the non-vegan buffer, Pistachio Pesto Pasta for the vegan buffer), and 2 means it has food type 2 (Garlic Sirloin Steak for the non-vegan tray, Avocado Fruit Salad for the vegan tray). Give these buffers a size of MAX_BUFFER_SIZE, which is a constant equal to 10.

At this point, you should be able to **fork()**, and if your shared buffers were created with the appropriate flags, then both child and parent should be able to read and write from this buffer. Test it before continuing!

3.1.2 Create the shared semaphores

Next up, for each bounded-buffer, we need a shared **binary semaphore** that makes sure only one process can write to the buffer at any one time. Call this semaphore *mutex* as it is acting as a lock for the buffer. For each buffer, we will also need two shared **counting semaphores** - called *full* and *empty* - that ensure a producer can't go into its critical section if the buffer is full, and likewise, a consumer can't do so if their respective buffer is empty.

For each respective buffer, the binary semaphore is initialized to 1, the empty semaphore is initialized to MAX_BUFFER_SIZE, and the full semaphore is initialized to 0.

Play with these semaphores with children processes to make sure mutual exclusion is being met.

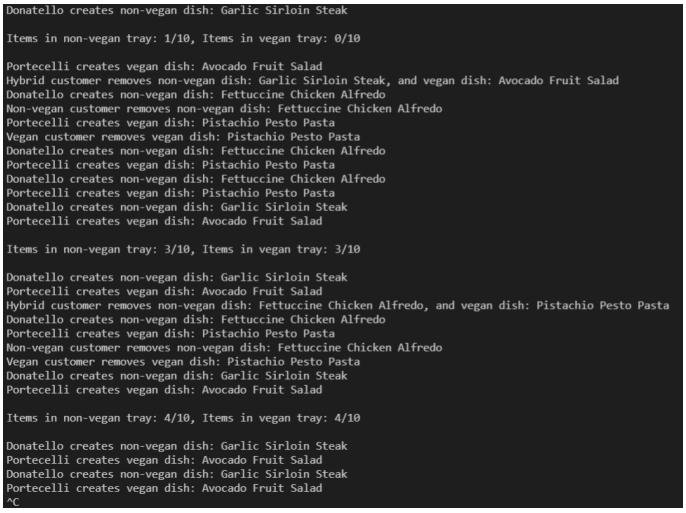
3.1.3 Create the producers and consumers

You should now have the necessary setup to start working on your producer and consumer processes to simulate the busy restaurant. Each time a producer or consumer executes its critical section (i.e. adds/takes from the buffer), they should print the following:

1. Whether it is a producer or consumer doing the action
2. Which producer/consumer it is:
   (a) chef Donatello vs chef Portecelli for the producers
   (b) non-vegan vs vegan vs hybrid for the consumers
3. Which variation of the dish the producer/consumer has added/taken from the buffer

Keep in mind that you should be implementing bounded-buffers! This means utilizing and keeping track of the **in** and **out** positions of each buffer. Simply adding/taking from any empty/full slot will not receive credit!

Additionally, since your producers and consumers are presumably children processes, your main process should be printing how full each tray is every 10 seconds. The following screenshot shows an example print for part one of your program:



Note: The output will be manually parsed, so don't stress the formatting and do it however you see visually pleasing.

4 Part Two

In this part you need to re-implement the above problem using threads. From an educational point of view, this should reflect your understanding of the use of threads vs processes (fork), and how to handle their side effects, if any. In your implementation, you need to consider the following:

1. Use thread instead of fork wherever possible.
2. Avoid zombie processes and threads (**pthread_exit** and **pthread_join**, etc.)
3. Avoid race conditions
4. Use as minimum shared memory as possible.

5 Grading

Part 1 will carry 17 points out of the 25 and part 2 will carry 8 points out of the 25 (the full assignment carries 25 points).

6 Deadline

The assignment is due by March 9 at 11:55 PM, after that the deduction policy of 20% daily (part of the day is considered as a full day) will be applied.

7 Submission

To submit use "tar -czvf file.tar.gz directory" and upload the file.tar.gz. file should be the student name+ID.