Programming Assignment 3

Concurrency Puzzles

Due: August 2, 2023

1 Introduction

This assignment is meant to get you in the concurrent programming mindset with some fun puzzles.

2 Given Materials

On Brightspace under **Content/Assignments/Assignment 3** you'll see:

- family.c
- diner.c
- monks.c
- conc_makefile

Each `.c` file corresponds to the concurrency puzzle I would like you to solve. I include some `C` libraries you might find either necessary or useful to solve the problems.

The makefile has three different compilation dependencies: *monks*, *diner*, and *family*. To compile the related `.c` files type "make <dependency>". (If I wanted to compile monks.c I would type `make monks` into the command line.) You'll have to rename it to simply "makefile" to run it properly.

3 Requirements

You will write a few programs using `POSIX` threads in `C`. These programs are defined below.

3.1 Part A: Happy Family

A family decides to eat out at a Chinese restaurant. Upon arriving they discover that the owners have placed only a singular chopstick between each plate. This unorthodox utensil arrangement makes it so the family members must share their chopsticks. A family member can only use the chopsticks on either side of them, and they must use both to eat their food!

The basic loop is as follows:

```
while True :
    talk()
    get_chopsticks()
    eat()
    put_chopsticks()
```

Construct a multi-threaded program that simulates this situation, making sure to avoid deadlocks. Print out the thread id when a family member eats so I know the program isn't deadlocked. You do not need to make the program self-terminate (it may run infinitely).

Hint: Treat the chopsticks as an array of semaphores.

3.2 Part B: Diner Counter

Imagine a diner counter with 8 stools. If you arrive when there is an empty stool, you can take a seat immediately. But if you arrive when all 8 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down. Create a multi-threaded program that fulfils these requirements.

Your program will take a single command line argument: the number of customers. Print out the thread id of a patron when they begin eating, as well as the average time it takes a thread to wait.

Hint: This can be done with a single mutex and a single semaphore.

3.3 Part C: Drunken Monks

Suppose we have a collection of 10 jolly monks who all wish to have a nice evening on the monastery. They decide the best way to accomplish that task is to make use of the copious amounts of beer that they've stored up over the past year. Before the monks can begin they have to ask the brew master to fetch one of their mighty kegs. When a keg is available, the monks can only fill their mugs **one at a time**, and each keg can **only store N mugs worth of beer** at a time. Obviously, a monk can **only draft beer if the keg isn't empty**—and the brew master **refuses to fetch a new keg until the greedy monks have drank all they have** in front of them.

You are tasked with creating a thread-safe, non-deadlocking program capable of satisfying the monks' thirst.

Defining the problem clearly:

- A keg can store N mugs-worth of beer. (This will be input through the command line.)
- The Brew Master has K kegs in his storage place. (This will be input through the command line.)
- 10 monks will race to drink as much as they can. **You will track how much each monk drinks.** (The total can never be greater than $N \times K$)
- The Brew Master can only add a new keg if the old one is empty.
- A monk cannot drink from an empty keg.

Non-thread-safe psuedocode looks as follows

```
Monk thread:
    while true:
        get_mug_of_beer();
        drink();

Brew Master thread:
    while true:
        fetch_new_keg(N); //N is the number of servings
```

Hint: A clever mix of semaphores and mutexes will get you where you need to go. It might also be a good idea to track the number of servings in the ever-dwindling keg.

4 Evaluation

Each concurrent program above should be capable of avoiding deadlocks. I will make sure to run your code a dozen or so times to check for deadlocks, so be sure that they can't crop up! In the event of a deadlock, or other obvious thread-unsafe behaviour, I will look through your source code to see how much effort was put into solving the problem—partial credit will be given at my discretion. If no concurrency issues are found, you'll receive 100 points for the assignment.

I also expect to see comments throughout your source code. Your comments do not need to be overly detailed, but they should provide a high level explanation as to why certain design choices were made.

If I can't compile your code, you will receive a 0 for that given part. Compilation will give you a baseline of 5 points for a given part. Part A is worth 30 points, Part B is worth 30 points, and Part C is worth 40 points.

5 Submission

As stated previously, make sure I can run your makefile and that the binaries are named `family`, `diner`, and `monks` (referring to parts A, B, and C, respectively). Submit your code and makefile in a `tar.gz` zipped file with the following naming format <email_username>_threads.tar.gz. If I were submitting the assignment, my file would be named as follows: jraskin3_threads.tar.gz

(You can compress a file by running `tar -czvf name-of-archive.tar.gz shell.c makefile`)