# CS 214: Systems Programming, Spring 2014
# Programming Assignment 5: Multithreaded Book Order System

## 1 Introduction

For this assignment, you will write a multithreaded book order simulation program. This will give you an opportunity to exercise mutexes and thread coordination. You will write a producer-consumer program with a single producer and multiple consumers. Your producer and your consumers will run as separate threads, so you will use mutexes to protect and manage shared data structures.

## 2 Producer: Book Order Input Thread

Your program will spawn a single order input thread (the producer). The input thread will read in a data file containing multiple book orders, one book order per line. Each book order will consist of:

- `Book Title (a string in double quotes)`
- `Customer ID number (a unique integer)`
- `Category (a unique alphanumeric string)`

Each book order will be handled by a separate book order processor thread (the consumers). You have (at least) two choices for where the producer thread puts the book orders. First, you may opt to put the book orders into separate queues, one for each category. Each consumer thread will have exclusive access to a specialized queue. Second, you may choose to put all book orders into a single shared buffer that will be used by all the consumer threads. The book order queues (or buffer) are initially empty.

Example book categories may include `Sports`, `Housing` or `Politics`.

## 3 Consumers: Book Order Processor Threads

Your program will also spawn multiple Book Order Processor threads (the consumers), one consumer thread for each book category. The Book Order Processor threads will extract all book orders for their category from the data structure shared with the producer and print the individual orders as they are processed. See below for a description of order processing. Remember that individual customers can and will make different book orders for books from different categories.

The Book Order Processor thread will also use a customer database that holds:

- `Customer name`
- `Customer ID number (should match order entered into queue)`
- `Credit Limit (a dollar amount)`

An order is processed by first finding the customer in the database, then deciding if the customer's credit limit is greater than or equal to the price of the ordered book. If the customer has sufficient funds, the credit limit is debited and the order processor thread prints an order confirmation listing the book name, price and shipping information (customer name, address, state, zip code). If the customer does not have sufficient funds, the order processor thread should print out an order rejection that identifies the customer name, the book order details and the remaining credit limit value for the customer. After all book orders have been processed, the program must print out a final report listing the following for each customer:

- `Customer name`
- `Customer ID number`
- `Remaining credit balance after all purchases (a dollar amount)`

And for each customer, make two lists. The first list is for successful book orders (the ones they can afford), and the second is for unsuccessful book orders (the ones they could not afford). Each line of the successful book orders should include:

- `Book title`
- `Book price (a dollar amount)`
- `Remaining credit balance after this purchase (a dollar amount)`

Each line of unsuccessful book orders would include:

- `Book title`
- `Book price (a dollar amount)`

The program should also print the total revenue gained from all successful book orders.

You may have to spawn other threads to make this book ordering system work.

Requirement 1: there should be a minimum of BUSYWAITING in your code.

Requirement 2: there should be NO DEADLOCKS and NO RACE CONDITIONS in your code.

## 4 Program Start-up

The command-line arguments will specify the following:

- `Arg 1:  the name of the database input file`
- `Arg 2:  the name of the book order input file`

- `Arg 3: the name of a file containing category names`

Your program starts by reading in the customer database file and setting up the customer data base. It then spawns the producer thread and the consumer threads, one consumer thread for each category.

## 5 Implementation

This program is an implementation of the Producer-Consumer problem where the producer may create large amounts of input (i.e. any number of book orders), but have only a small, fixed space to communicate book orders to the consumers. We are providing a sample input file, but this is only a sample input. We may test your program with larger inputs. Your program must be able to handle any number of book orders, not just the ones in our sample input file. Therefore, it is not acceptable for the producer to simply read in all the book orders into a large buffer and quit, and then have the consumers simply read the book orders left behind by the now terminated producer.

Your program must show the *interleaving* of book order creation by the producer and book order processing by the consumers. Minimally, you code should produce the following messages:

- Producer waits because queues or buffers are full.
- Producer resumes when queues or buffer space is available.
- Consumer waits because queues or buffers are empty.
- Consumer resumes when queues or buffers have orders ready for processing.

These messages would show the interleaving of book order creation and book order processing.

## 6 Program Termination

After all in input has been processed, the producer and all the consumer threads shut shut down and the program should print out the final report as described above.

## 7 Extra Credit

Normally, the producer and consumer threads run in the same process. For extra credit, you can make the producer and consumer threads run as separate processes that communicate through `shared memory`. Your producer program might set up the shared memory and then use `fork()` and `exec()` to spawn the consumer process(es). You could have a single consumer process that handles all book categories (maybe with separate threads for each category) or you could have separate consumer processes, each of which handles a separate book category. As a responsible parent process, the producer process would use `wait()` to clean up the child consumer process(es) and clean up the shared memory.

If you choose to use separate processes, it would be useful (and look cool) to have the producer process print out messages about the consumer process(es) is creates–something like "Created child consumer process <PID> to handle category <CATEGORY>". You can also add similar messages about how the producer parent wait()s for each of its children.

You could also add messages about set up and removal of shared memory.

## 8 What to turn in

- A writeup documenting your design **paying particular attention to the thread synchronization requirements of your application**.
- All source code including both implementation (.c) and interface(.h) files.
- A makefile for producing an executable program file.

Your grade will be based on:

- Correctness (how well your code is working, including avoidance of deadlocks.
- Efficiency (avoidance of recomputation of the same results, avoidance of busy-waits).
- Good design (how well written your design document and code are, including modularity and comments).