CSc 360: Operating Systems (Fall 2020)

Assignment 2: Airline Check-in System

Code Due: 11:55 pm, Nov. 6, 2020

1 Introduction

In this assignment, you need to face the second programming challenge: implementing a task scheduler. You will learn how to use the three programming constructs provided by the POSIX `pthread` library:

1. thread
2. mutex
3. condition variable (convar)

to do so. Your goal is to simulate an airline check-in system, called ACS.

The check-in system includes 2 queues and 4 clerks. One queue (Queue 0) for economy class and the other (Queue 1) for business class. We assume that the customers in each queue are served FIFO, and the customers in the business class have a higher priority than the customers in the economy class. In other words, when a clerk is available, the clerk picks a customer in the business class, if any, to serve, and picks a customer in the economy class to serve only if the business class queue is empty. When a clerk is available and there is no customer in any queue, the clerk remains idle. We assume that the service time for a customer is known when the customer enters the system.

You will use threads to simulate the customers arriving and waiting for service, and your program will schedule these customers following the above rules.

We assume that initially all queues are empty, and all clerks are idle. We assume customers are given before hand and their description is stored in a file (refer to Section 2.2).

2 Customers

2.1 Properties of Customers

Each customer, which will be simulated by a thread, has the following attributes:

1. **Class Type:**: It indicates whether the customer belongs to business class or economy class.
2. **Arrival Time**: It indicates when the customer will arrive.
3. **Service Time**: It indicates the time required to serve the customer (i.e., from the time when the customer is picked up by a clerk to the time when the clerk finishes serving the customer).

All times are measured in 10ths of a second. The times will be simulated by having your threads, which represent customers, to call `usleep()` for the required amount of time.

2.2 The Format of Input File

Your program (ACS) will accept *one* parameter on the command line:

```
ACS customers.txt
```

where `customers.txt` is the name of the input file.

2.2.1 File Format

The input file is a text file and has a simple format. The first line contains the total number of customers that will be simulated. After that, each line contains the information about a single customer, such that:

1. The first character specifies the unique ID of customers.
2. A colon(`:`) immediately follows the unique number of the customer.
3. Immediately following is an integer equal to either 1 (indicating the customer belongs to business class) or 0 (indicating the customer belongs to economy class).
4. A comma(`,`) immediately follows the previous number.
5. Immediately following is an integer that indicates the **arrival time** of the customer.
6. A comma(`,`) immediately follows the previous number.
7. Immediately following is an integer that indicates the **service time** of the customer.
8. A newline (`\n`) ends a line.

To not kill yourself by checking the false input file, you *should* build a correct input file for your own test. We will not test your code with an input file that does not comply with the above format.

2.2.2 An Example

The following file specifies 7 customers

```
7
1:0,2,60
2:0,4,70
3:0,5,50
4:1,7,30
5:1,7,40
6:1,8,50
7:0,10,30
```

Note that all times are measured in 10ths of a second, and in the above example, the first customer arrives at 0.2s and her service time is 6s.

2.3 Output

Your simulation is required to output all events and state changes showing the internal behavior of ACS. The messages must include, but are not limited to:

1. A customer arrives: customer ID
2. A customer enters a queue: the queue ID, and length of the queue.
3. A clerk starts serving a customer: start time, the customer ID, the clerk ID.
4. A clerk finishes serving a customer: end time, the customer ID, the clerk ID.

Sample format of the output could be:

```
"A customer arrives: customer ID %2d. \n"
"A customer enters a queue: the queue ID %1d, and length of the queue %2d. \n"
"A clerk starts serving a customer: start time %.2f, the customer ID %2d, the clerk ID %1d. \n"
"A clerk finishes serving a customer: end time %.2f, the customer ID %2d, the clerk ID %1d. \n"
```

Note that the output of times (including arrival time, the time when a clerk starts serving a customer, the time when a clerk finishes a service) is **relative machine time**, calculated by the machine time when the output event occurs minus the machine time when the simulation starts. Therefore, the output of the times may not exactly matches (but should be close to) the results with manual calculation.

At the end of the your code, you need to output (1) the average waiting time of all customers in the system, (2) the average waiting time of all business-class customers, and (3) the average waiting time of all economy-class customers. The waiting time of a customer is defined as from the time when the customer enters a queue to the time when a clerk starts servicing the customer.

Sample format of the output could be:

```
"The average waiting time for all customers in the system is: %.2f seconds. \n"
"The average waiting time for all business-class customers is: %.2f seconds. \n"
"The average waiting time for all economy-class customers is: %.2f seconds. \n"
```

3 Design Document

You will write a design document which answers the following questions. It is recommended that you think through the following questions *very carefully* before answering them.

Unlike Assignment 1, debugging will be harder after the basic design has been coded. Therefore, it is very important to have a clear design before you start coding. So think about the following carefully and then write down the answers.

1. How many threads are you going to use? Specify the task that you intend each thread to perform.
2. Do the threads work independently? Or, is there an overall "controller" thread?
3. How many mutexes are you going to use? Specify the operation that each mutex will guard.
4. Will the main thread be idle? If not, what will it be doing?
5. How are you going to represent customers? what type of data structure will you use?
6. How are you going to ensure that data structures in your program will not be modified concurrently?
7. How many convars are you going to use? For each convar:
   (a) Describe the condition that the convar will represent.
   (b) Which mutex is associated with the convar? Why?
   (c) What operation should be performed once `pthread_cond_wait()` has been unblocked *and* re-acquired the mutex?
8. Briefly sketch the overall algorithm you will use. You may use sentences such as:

   If clerk i finishes service, release clerkImutex.

Note: You are required to type in your document and submit it in pdf file format together with your code. Other file format or handwriting will **not** be accepted.

4 Submission

The code and the design document are submitted through `connex-->Assignments` in one zip file. The tutorial instructor will give the detailed instruction in the tutorial.

4.1 Submission Requirements

1. The name of the submission file must be `p2.tar.gz`
2. `p2.tar.gz` must contain all your files in a directory named `p2`
3. Inside the directory `p2`, there must be a `Makefile`.
4. Invoking `make` on it must result in an executable named `ACS` being built, *without user intervention.*
5. You should *not* submit the assignment with a compiled executable and/or object (`.o`) files.
6. Inside the directory `p2`, there must be an input file following the format described in Section 2.2, although we will test you code using our own input file.
7. The design document specified in Section 3.

5 Plagiarism

This assignment is to be done individually. You are encouraged to discuss the design of the solution with your classmates, but each student must implement their own assignment. The markers may submit your code to an automated plagiarism detection service.

6 Miscellaneous

6.1 Manual Pages

Be sure to study the `man` pages for the various functions to be used in the assignment. For example, the `man` page for `pthread_create` can be found by typing the command:

```
$ man pthread_create
```

At the end of this assignment you should be at least familiar with the following functions:

1. File access functions:
   (a) `atoi`
   (b) `fopen`
   (c) `feof`
   (d) `fgetc` and `fgets`
   (e) `fclose`
2. Thread creation functions:
   (a) `pthread_create`
   (b) `pthread_exit`
   (c) `pthread_join`
3. Mutex manipulation functions:
   (a) `pthread_mutex_init`
   (b) `pthread_mutex_lock`
   (c) `pthread_mutex_unlock`
4. Condition variable manipulation functions:
   (a) `pthread_cond_init`

(b) `pthread_cond_wait`

(c) `pthread_cond_broadcast`

(d) `pthread_cond_signal`

It is absolutely critical that you read the `man` pages, and attend the tutorials.
Your best source of information, as always, is the `man` pages.
For help with POSIX threads:

http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

A good overview of `pthread` can be found at: http://www.llnl.gov/computing/tutorials/pthreads/

6.2 Important Notes

We want to (re-)emphasize the following points:

1. **You are required to type in your design document. Hand writing will not be accepted.**
2. We will give a **time quota of** 3 **minutes** for your program to run on a given input. This time quota is given so that non-terminating programs can be killed. So make sure your input file does not include too many long customers (e.g., arrive too late or service time too long). Since your program simulates in 10ths of a second, this should not be an issue, at all.
3. It is required that you use **relative machine time**. This is to avoid cheating with an implementation that does not really simulate the customers but instead performs an offline analysis to obtain results. The markers will read your C code to ensure that the `pthread` library is used as required. Offline analysis means that your program does not simulate mutual exclusion and thread synchronization but obtains the output based on algorithm analysis. **You will get** 0 **marks if you are caught using offline analysis.**
4. As you progress through your degree the projects and assignments will continue to become more complicated and difficult. It is impossible to describe in detail every possible case/feature in the assignment specification. Instead you are expected to apply the techniques you have learned so far in your degree, use common sense, and ask questions to lecture instructor or TAs when something is unclear. We will announce further clarification, if necessary, on `Connex`. Complaining the specification is unclear at the last minute is **not acceptable.**
5. You are required to use C. Any other language is **not acceptable.**
6. You are required to strictly follow the input file format. Failing to do so will result in the deduction of scores.
7. Your work will be tested on linux.csc.uvic.ca. Make sure your code can work on linux.csc.uvic.ca
8. Programming with semaphore is permitted but not recommended. You have been warned that debugging with semaphore is much harder.

7 Marking

We will mark your design document and your code submission.

7.1 Design Document (10% of this assignment)

You are required to answer all questions listed in Section 3. Your design will be marked based on the clear and correct logic and the correctness of your algorithm.

7.2 Code (90% of this assignment)

7.2.1 Functionality

1. Your `ACS` must correctly schedule the customer services, with our own test files. We will not disclose all test files before the final submission. This is very common in software engineering.

2. You are required to catch return errors of important function calls, especially when a return error may result in the logic error or malfunctioning of your program.

3. Your program must output at least the information described in Section 2.3.

7.2.2 Code Quality

We cannot specify completely the coding style that we would like to see but it includes the following:

1. Proper decomposition of a program into subroutines (and multiple source code files when necessary)—A 1000 line C program as a single routine would fail this criterion.

2. Comment—judiciously, but not profusely. Comments also serve to help a marker, in addition to yourself. To further elaborate:

   (a) Your favorite quote from Star Wars or Douglas Adams' Hitch-hiker's Guide to the Galaxy does not count as comments. In fact, they simply count as anti-comments, and will result in a loss of marks.

   (b) Comment your code in English. It is the official language of this university.

3. Proper variable names—`leia` is not a good variable name, it never was and never will be.

4. Small number of global variables, if any. Most programs need a very small number of global variables, if any. (If you have a global variable named `temp`, think again.)

5. **The return values from all system calls and function calls listed in the assignment specification should be checked and all values should be dealt with appropriately.**

7.3 Detailed Test Plan

The detailed test plan for the code submission is as follows.

Components	Weight
Make file	5
Input file	5
Normal cases	25
Special cases (illegal values)	5
Output format	5
Catch system call return values	5
Correct statistics	15
Comments (functional decomposition)	5
Code style	5
Critical sections	10
Readme	5
Total Weight	90

Last but not the least, please read this document **carefully**. TAs and the lecture instructor will not respond for questions or appeals that have been clearly explained in this document. If you are not sure, confirm with the course instructor.