CSCI 340 - Operating Systems

Assignment 4 Total Points 100

Objectives

In this assignment you will develop a C program which uses binary semaphores to solve a synchronization problem. This assignment will allow you to gain experience in the following areas:

- **Threads**: This includes *creating* threads, *joining* threads, and *cancelling* threads. You will be using the *pthreads* library.
- **Semaphores**: You will use binary semaphores in your solution using the provided *binary_semaphore.c* implementation along with the *binary_semaphore.h* include file. For your solution, you will have to make decisions regarding:
  - how many semaphores to use
  - what initial values to give the semaphores
  - where to call *semWaitB()* and *semSignalB()* on each semaphore
- **Deadlock** and **Starvation**: As you develop, you will likely encounter deadlock. You will need to determine why you got deadlock, and determine a solution. Though starvation is possible, there should be many example command-line inputs to your program that don't exhibit starvation. In short, if you are ever getting deadlock, or often/always getting starvation, then your solution is wrong.
- **Parallel Debugging**: You will have to debug your code, which is parallel. This is challenging, and using a debugger is often not helpful. Using well-placed *printf()* statements in your code is good way to debug.
- **Non-determinism**: It is normal for your program to exhibit *non-determinism* (ie. The output will be different each time you run it, even with the same command-line input parameters). The reason for this is *timing* (of the scheduler, of the sleep function, of system calls, etc). This is normal. **However**, the output from your code solution should always satisfy the synchronization constraints (listed in the next section).
- **Simulation Principles**: This assignment is really a simulation. There are are several useful simulation routines provided that you will be using including:
  - sleeping (delaying) for a fraction of a second
  - seeding and generating random numbers *within a multithreaded* program (ie. re-entrant random numbers)
  - generating a *random integer within a range*
- **Command-line Parsing**: You will need to obtain command-line strings and convert them to integers. You should do some error checking for valid input.
- **Dynamic Memory**: You will need to dynamically allocate an *array of threads* and *array of seeds* (for re-entrant random number generation).

Description

A security guard walks around the CS department checking study rooms. CS majors use study rooms, obviously to study. The fire safety code specifies a maximum capacity on the number of students in a room. The guard checks for this. In addition, the guard will check an empty study room to make sure there are no other safety issues. **However**, when the guard goes to check a study room, if there are students in the room, the guard will not interrupt the studying, but rather wait outside the room until either no students are in the room or the number of students in the room equals or exceeds the fire safety capacity.
More specifically, here are the rules (synchronization constraints):

1. Any number of students can be in the study room at the same time.
2. The guard can enter the room **only if** there are no students in the room (to conduct a security assessment) **or if** there are equal or more than *capacity* students in the room (to clear out the room for fire safety). The variable, *capacity*, is a command-line input.
3. When the guard is in the room, no additional students may enter, but students may leave. Thus, students may leave the room at any time.
4. The guard may not leave the room until all students have left (fire safety restored).
5. There is only one guard!

Provided Files

The three files listed below are provided to you.

- **binary_semaphore.h**: Header file that defines the binary semaphore *struct* used in this assignment, and the three binary semaphore operations:
  1. void semInitB(binary_semaphore* s, int state);
  2. void semWaitB(binary_semaphore* s);
  3. void semSignalB(binary_semaphore* s);

  No change should be made to this file.
- **binary_semaphore.c**: The file containing the implementation of the functions listed in *binary_semaphore.h*. Having a different file for the implementation separates interface (the include file) from the implementation (the .c file). No change should be made to this file.
- **security_guard.c**: Source code file that includes a "skeleton" version of a solution along with necessary includes, constants, and useful functions. **Note**: You may not include (i.e. #include) additional library headers. The ones that are provided, are the only libraries needed/allowed for this assignment.

The *security_guard.c* file contains many comments that provide discussion, hints, and guidance. Read the comments carefully!
Sample output is provided in the file, *sample_output.txt*. **Your program should generate output similar to the sample output**. In addition, I've included a Linux executable, *security_guard_solution*, that should run on most 64-bit Linux systems. The sample output was generated by entering:

```
./security_guard_solution 8 4 10
```

Compare and test your code against my solution with different input parameters.

Todo

In the *security_guard.c* file, you must complete the following function implementations with appropriate synchronization calls (ie. *semWaitB() and semSignalB()*), proper conditionals and appropriate maintenance of global variables (ie. *guard_state* and *num_students*). You will also need to declare your global binary semaphores.

```
void guard_check_room()             // main synchronization logic for the guard
void student_study_in_room(long id) // main synchronization logic for a student
int main(int argc, char** argv)     // get input, init, allocate, manage threads
```

For each function listed above, numerous **TODO** comments are provided in the file, *security_guard.c*, to guide you in this assignment. Read them carefully!

Collaboration and Plagiarism

This is an **individual assignment**, i.e. **no collaboration is permitted**. Plagiarism will not be tolerated. Submitted solutions that are very similar (determined by the instructor) will be given a grade of zero. Do your own work, and everything will be OK.

Submission

Create a compressed tarball, i.e. *tar.gz*, that contains only the completed *security_guard.c* file. The name of the compressed tarball must be your last name in lower case. For example, *ritchie.tar.gz* would be correct if the original co-developer of UNIX (Dennis Ritchie) submitted the assignment. Only assignments submitted in the correct format will be accepted (no exceptions). Submit the compressed tarball to the appropriate Dropbox on OAKS by the due date. You may resubmit the compressed tarball as many times as you like, only the latest submission will be graded.
To be fair to everyone, late assignments will not be accepted. Exceptions will only be made for extenuating circumstances, i.e. death in the family, health related problems, etc. You will be given **9 days** to complete this assignment. Poor time management is not an excuse.
Do not email your assignment after the due date. It will not be accepted. If you need help, stop by during office hours to discuss the assignment. I am always happy to listen to your approach and make suggestions. However, I cannot tell you how to code the solution. In addition, code debugging is your job. You should use print statements to help understand why your solution is not working correctly.

Grading Rubric

For this assignment the grading rubric is provided in the table shown below.

reasonable number and names of semaphores	5 points
guard_check_room() function implementation	30 points
student_study_in_room() implementation	30 points
main() command-line input and convert	5 points
main() memory management	5 points
main() semaphore initializations	5 points
main() student thread creation and seed init	5 points
main() thread cancelling	5 points
similar output format to sample	10 points

NOTE: Few points, if any, will be given to a program which either does not compile or generates a runtime error. Some examples of a runtime error are *segmentation fault*, *divide by zero*, *memory fault.*