SJSU CS/SE 149 HW4 Spring 2018

REMINDER: Each homework (including programming question) is individual. "Every single byte must come from you." Cut&paste from others is not allowed. Keep your answer and source code to yourself only.

1. (20 pts) Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
*P*1	2	3
*P*2	1	1
*P*3	8	5
*P*4	4	2
*P*5	5	4

The processes are assumed to have arrived in the order *P*1, *P*2, *P*3, *P*4, *P*5, all at time 0. Use a software to draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, nonpreemptive SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2), and calculate the average waiting time for each algorithm. Hand drawing is not accepted.

2. [programming question] (80 pts) A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and the TA can help only one student at a time in the office. There are two chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time. For simplicity we assume there are total four students, and each student receives TA's helps at most two times.

Using C and POSIX threads, mutex locks, and unnamed semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

The Students and the TA

Use Pthreads to create student threads. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in one of chairs in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students sitting on chairs in the hallway waiting for help. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.

To simulate students programming in student threads, and the TA providing help to a student in the TA thread, the appropriate threads should sleep (by invoking `sleep()`) for a random period of time (up to three seconds). Instead of invoking the random number generator `rand()` which is not thread-safe, each thread should invoke the thread-safe version `rand_r()`.

- `rand_r()` computes a sequence of pseudo-random integers in the range [0, RAND_MAX]. If you want a value between 1 and n *inclusive*, use `(rand_r(&seed) % n) + 1`.
- It is recommended to use a different seed value for `rand_r()` in each thread so that each thread can get a different sequence of pseudo-random numbers.

To simplify the situation, when TA helps a student, only the TA thread invokes `sleep()` while the student thread does not invoke sleep().

For simplicity, each student thread terminates after getting help *twice* from the TA. The TA is not aware of the number of students, nor does the TA keep track of how many helps are yet to be offered to students. After all student threads

terminate, the program cancels the TA thread by calling `pthread_cancel()` and then the entire program terminates.

Based on aforementioned requirements, your C program should have the following `#defines`:

```
/* the maximum time (in seconds) to sleep */
#define MAX_SLEEP_TIME     3

/* number of potential students */
#define NUM_OF_STUDENTS          4

#define NUM_OF_HELPS       2

/* number of available seats */
#define NUM_OF_SEATS  2
```

POSIX Synchronization

You need the following POSIX mutex locks and (unnamed) semaphores:

```
#include <pthread.h>
#include <semaphore.h>

/* mutex declarations, a global variable */
pthread_mutex_t  mutex_lock;      /* protect the global variable waiting_student */

/* semaphore declarations, global variables */
sem_t students_sem;     /* ta waits for a student to show up, student notifies ta his/her arrival*/
sem_t ta_sem;           /* student waits for ta to help, ta notifies student he/she is ready to help*/

/* the number of waiting students, a global variable */
int waiting_students;
```

Other than the above global variables, you are **not** allowed to have any additional global variables for synchronization purpose, such as `students_finished`, `is_ta_sleeping`, `open_seats`, or variables to that effect. It is OK, but not recommended, to have additional global variables to record information (e.g., thread ids), as long as they are not for synchronization purpose.

Coverage of POSIX mutex locks and unnamed semaphores (sem_init, sem_wait, sem_post) are provided in Section 5.9.4. Consult that section for details.

Before using any mutex and semaphore, one must initialize it. One must destroy any mutex and semaphore before the process terminates.

Note one does not need to have a waiting student queue. As long as there is available seat, the student can simply invoke wait on the proper semaphore. When the TA offers the help by invoking signal on the proper semaphore, the default Pthread implementation on Linux removes a waiting student thread in FIFO order.

Please note

1. Each invocation the program always prints out "CS149 SleepingTA from FirstName LastName" only once. Take screenshots of the program execution (including "CS149 SleepingTA from …").

2. Any API in a multi-threaded application must be thread-safe (e.g., on Linux `rand()` vs `rand_r()`). Invoking any non thread-safe API is subject to deduction.

3. No additional global variables for synchronization purpose is allowed.

4. The TA is not aware of the number of students, nor does the TA keep track of how many helps are yet to be offered to students. TA thread will be terminated by `pthread_cancel()`.

5. One must invoke the random number generator each time to get the student programming time (student thread calls `sleep()`) and the TA helping time (TA thread calls `sleep()`).

6. You must utilize array for various data structures, and utilize loop to remove repeating code. Any repetitive variable declaration and/or code are subject to deduction.

7. No recursion is allowed in any function.

The following is a sample output:

```
CS149 SleepingTA from FirstName LastName
        Student 2 programming for 2 seconds
        Student 1 programming for 3 seconds
        Student 0 programming for 3 seconds
        Student 3 programming for 3 seconds
                Student 2 takes a seat, # of waiting students = 1
Helping a student for 3 seconds, # of waiting students = 0
Student 2 receiving help
        Student 2 programming for 3 seconds
                Student 1 takes a seat, # of waiting students = 1
                Student 0 takes a seat, # of waiting students = 2
                        Student 3 will try later
        Student 3 programming for 3 seconds
                        Student 2 will try later
        Student 2 programming for 1 seconds
Helping a student for 3 seconds, # of waiting students = 1
Student 1 receiving help
        Student 1 programming for 3 seconds
                Student 3 takes a seat, # of waiting students = 2
                        Student 2 will try later
        Student 2 programming for 3 seconds
                        Student 1 will try later
        Student 1 programming for 3 seconds
Helping a student for 3 seconds, # of waiting students = 1
Student 0 receiving help
        Student 0 programming for 3 seconds
                Student 2 takes a seat, # of waiting students = 2
                        Student 0 will try later
        Student 0 programming for 3 seconds
                        Student 1 will try later
        Student 1 programming for 3 seconds
Helping a student for 3 seconds, # of waiting students = 1
Student 3 receiving help
```

Compile your program with "`gcc –o sleepingTA sleepingTA.c -pthread`". You can execute the program with "`./sleepingTA`".

Submit the following files as individual files (do not zip them together):

- `CS149_HW4_YourName_L3SID` (.pdf, .doc, or .docx), which includes
  - Q1: answers
  - Q2:
    - in less than one page, high level description of how synchronization is done by each thread. In particular, describe the exact calling sequence of two semaphores and mutex lock in TA thread and student thread, respectively.
    - screenshots of the program execution. Screenshots that are not readable, or screenshot without "CS149 SleepingTA from …" from the program, will receive 0 point for the entire homework.

- `sleepingTA_YourName_L3SID.c` Your source code without binaries/executables. Please ident your source code and include comments.

The ISA and/or instructor leave feedback to your homework as comments and/or annotated comment. To access annotated comment, click "view feedback" button. For details, see the following URL:

http://guides.instructure.com/m/4212/l/106690-how-do-i-use-the-submission-details-page-for-an-assignment

NOTE: the course requires you to use Linux VM even on Mac. Unnamed POSIX semaphore used in HW4 is **not** supported on Mac OS X. If you still use Mac for HW4, you could use Grand Central Dispatch instead. For details, see http://stackoverflow.com/questions/1413785/sem-init-on-os-x.