SJSU CS 149 HW4 FALL 2021

REMINDER: Each homework 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 - never post or share them to any site in any way.

[Type your answer. Hand-written answer is not acceptable.]
[Replace YourName and L3SID with your name and last three digit of your student ID, respectively.]

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.
(8 pts) a. Use any 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). Hand drawing is **not** accepted.
(4 pts) b. For each algorithm in a, calculate the average waiting time.
(4 pts) c. For each algorithm in a, calculate the average turnaround time.
(4 pts) d. Specify these four "Gantt charts in text" (for online exam), corresponding to the Gantt charts in (a).

2. (80 pts) [programming question] (80 pts) Welcome to SJSU Skydiving School. Any SJSU employees or students can take one-to-one skydiving lessons taught by one of instructors.

Use C and POSIX threads, mutex locks, and semaphores to implement a solution that coordinates the activities of instructors and the students.

The Students and the instructors
There are `NUM_OF_INSTRUCTORS` skydiving instructors, and `NUM_OF_STUDENTS` students. Each instructor and each student must rest before entering the lounge room. When an instructor is ready to teach a session, the instructor enters the lounge room and checks to see if there is a student waiting. If so, they can both proceed. Otherwise the instructor waits. Similarly, when a student is ready to take a session, the student enters the lounge room and checks for an instructor and either proceeds or waits, accordingly.

Given two semaphores (to guarantee execution order/sequence) students_q and instructors_q, a simple solution is

instructor	**student**
students_q.signal(); /* if any student in q then notify */ instructors_q.wait(); /* else wait in q if no student */ teach_session();	instructors_q.signal(); /* if any instructor in q then notify*/ students_q.wait(); /* else wait in q if no instructor */ take_session();

Each instructor signals exactly one student, and each student signals one instructor, so it is guaranteed that instructors and students are allowed to proceed in pairs. But whether they actually proceed in pairs is not clear in the above solution; It is possible for any number of instructors to accumulate before executing teach_session(), and therefore it is possible for any number of instructors to execute teach_session() before any students to execute take_session().

To make things more interesting, let's add the additional constraint that an instructor can invoke teach_session() concurrently with only one student (who invokes the corresponding take_session()), and vice versa. In other words, no more than one pair of instructor and student can skydive concurrently, and while the pair (instructor I, student S) are skydiving, no other pair of instructor and student can skydive until the current skydiving pair (I, S) is done.

Use Pthreads to create one thread per student and one thread per instructor. Each instructor rests from 1 up to `MAX_REST_TIME` seconds. So does each student. Each skydive session lasts from 1 to `MAX_SESSION_TIME` seconds. To simulate students resting in student threads, instructors resting in instructor threads, and instructor teaching session in instructor thread, the appropriate threads should invoke `sleep()` for a random period of time. Instead of invoking the random number generator `rand()` which is not reentrant, each thread must invoke the reentrant version `rand_r()` before each invocation to `sleep()`.

- `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, specify `(rand_r(&seed) % n) + 1`.
- Each thread should get a unique seed value only once and keeps using the same seed value for `rand_r()` so that each thread can get a different sequence of pseudo-random numbers.
- Each time before a thread invokes `sleep()`, the thread must invoke `rand_r()` first.

For simplicity, each student thread repeats the cycle of resting, entering the lounge room, and taking a skydive session, and terminates after taking sessions `NUM_OF_SESSIONS` times from instructors (regardless from which instructor). Any instructor repeats the cycle of resting, entering the lounge room, and teaching a skydive session; an instructor is not aware of `NUM_OF_STUDENTS`, nor does an instructor know `NUM_OF_SESSIONS`. The main program invokes `pthread_join()` to wait for the termination of all student threads and then invokes `pthread_cancel()` to cancel all instructor threads. The entire program then terminates.

POSIX Synchronization

Cut&paste the followings to your source code:

```
/* the maximum time (in seconds) to rest */
#define MAX_REST_TIME      3
/* the maximum time (in seconds) of a session */
#define MAX_SESSION_TIME   4
/* number of students */
#define NUM_OF_STUDENTS    4
/* number of instructors */
#define NUM_OF_INSTRUCTORS 2
/* # of sessions each student must take before exit */
#define NUM_OF_SESSIONS    2

/* binary semaphores */
sem_t      mutex;
/* # of waiting instructors, students in the queue */
int        waiting_instructors, waiting_students;
/* guarantee execution sequence - queue for instructors, students */
sem_t      instructors_q, students_q;
/* guarantee execution sequence - session over? */
sem_t      session_over;
```

Other than the above global shared variables, you are **not** allowed to have any additional global variables.

Semaphores `instructors_q` and `students_q` are the queues where instructors and students wait. The global variables `waiting_instructors` and `waiting_students` are counters that keep track of the number of instructors and students that are waiting in the queues, respectively. The binary semaphore `mutex` guarantees exclusive access to these two counters, and is also used to guarantee only one pair of teacher and student involved in a session at a time (i.e., no concurrent sessions). `session_over` is used to ensure that a pair of instructor and student are done with a session.

The high-level logic of students and instructors are as follows

- Each student must rest first before entering the lounge room. So does each instructor.

- When a student enters the lounge room, if there is an instructor waiting, the student notifies the instructor about his/her arrival and then both proceed to a session. If there is no waiting instructor when a student arrives, the student waits in the student queue until the student is notified by an arriving instructor, and then both proceed to a session.
- When an instructor enters the lounge room, if there is a student waiting, the instructor notifies the student about his/her arrival and then both proceed to a session. If there is no waiting student when an instructor arrives, the instructor waits in the instructor queue until the instructor is notified by an arriving student, and then both proceed to a session.
- When a session is on, the student waits for the end of the session. The instructor thread calls `sleep()` to simulate teach_session() and then notifies the student after the session is over. Any additional pair of instructor and student can then proceed to another session. Both instructor and student must take a rest before re-entering the lounge room.
- How to guarantee no concurrent teach_session()/take_session()? Keep holding the binary semaphore (mutex) until a session is over.
- There is only one airplane in the skydiving school. All sessions are conducted from this airplane which, in addition to the pilot, can fit only a pair of instructor and student. While any session is on, neither additional instructor nor additional student can enter the lounge room. After a session is over, additional instructor (if any) or additional student (if any) can then enter the lounge room.

Your program must follow the flows and steps as defined above, and your program must use the semaphores and mutex variables based on the stated purpose as defined. Any deviation, such as (but not limited to) using the semaphores as counting semaphores, is subject to deduction.

Your program output format must be similar to the followings (the exact sequence is different, of course):

```
kong@ubuntu2004:~/tmp$ ./skydivingschool
CS149 Fall 2021 Skydiving School from FirstName LastName
instructor[0, 0]: rest for 2 seconds
student[0, 0]: rest for 1 seconds
student[1, 0]: rest for 2 seconds
student[2, 0]: rest for 2 seconds
student[3, 0]: rest for 1 seconds
instructor[1, 0]: rest for 3 seconds
student[0, 0]: waiting_instructors=0, wait_students (excluding me) = 0
student[3, 0]: waiting_instructors=0, wait_students (excluding me) = 1
instructor[0, 0]: waiting_instructors (excluding me) = 0, wait_students=2
instructor[0, 1]: teach a session for 4 seconds
student[0, 1]: learn to skydive
instructor[0, 1]: rest for 3 seconds
student[2, 0]: waiting_instructors=0, wait_students (excluding me) = 1
student[1, 0]: waiting_instructors=0, wait_students (excluding me) = 2
student[0, 1]: rest for 3 seconds
instructor[1, 0]: waiting_instructors (excluding me) = 0, wait_students=3
instructor[1, 1]: teach a session for 4 seconds
student[3, 1]: learn to skydive
instructor[1, 1]: rest for 2 seconds
student[3, 1]: rest for 1 seconds
instructor[0, 1]: waiting_instructors (excluding me) = 0, wait_students=2
instructor[0, 2]: teach a session for 1 seconds
student[2, 1]: learn to skydive
instructor[0, 2]: rest for 1 seconds
student[2, 1]: rest for 1 seconds
student[0, 1]: waiting_instructors=0, wait_students (excluding me) = 1
student[3, 1]: waiting_instructors=0, wait_students (excluding me) = 2
instructor[1, 1]: waiting_instructors (excluding me) = 0, wait_students=3
instructor[1, 2]: teach a session for 2 seconds
student[1, 1]: learn to skydive
instructor[1, 2]: rest for 2 seconds
instructor[0, 2]: waiting_instructors (excluding me) = 0, wait_students=2
instructor[0, 3]: teach a session for 2 seconds
student[1, 1]: rest for 2 seconds
student[0, 2]: learn to skydive
instructor[0, 3]: rest for 3 seconds
instructor[1, 2]: waiting_instructors (excluding me) = 0, wait_students=1
student[0, 2]: done
instructor[1, 3]: teach a session for 2 seconds
student[3, 2]: learn to skydive
instructor[1, 3]: rest for 2 seconds
student[3, 2]: done
student[2, 1]: waiting_instructors=0, wait_students (excluding me) = 0
student[1, 1]: waiting_instructors=0, wait_students (excluding me) = 1
instructor[0, 3]: waiting_instructors (excluding me) = 0, wait_students=2
instructor[0, 4]: teach a session for 1 seconds
student[2, 2]: learn to skydive
instructor[0, 4]: rest for 3 seconds
instructor[1, 3]: waiting_instructors (excluding me) = 0, wait_students=1
instructor[1, 4]: teach a session for 2 seconds
student[1, 2]: learn to skydive
student[2, 2]: done
student[1, 2]: done
main: done
kong@ubuntu2004:~/tmp$
```


- student[a, b]: a is student ID (0, 1, etc.), b is # of sessions already taken by student a.
- instructor[c, d]: c is instructor ID (0, 1, etc.), d is total # of sessions already taught by instructor c.

- For a given a, the value of b in student[a, b] keeps increasing until reaching `NUM_OF_SESSIONS`.
- For a given c, the value of d in instructor [c, d] keeps increasing.
- The sum of the final d values from all instructors = (`NUM_OF_SESSIONS * NUM_OF_STUDENTS`).
- The # of waiting_instructors (or waiting_students) in the sample output is the exact value of the global variable `waiting_instructors` (or `waiting_students`) at that moment. "excluding me" implies that we print out its value before modifying its value.

Compile your program with "`gcc -o skydivingschool skydivingschool.c -pthread`". You can execute the program in a Linux terminal with "`./skydivingschool`".

Reminders

1. Each invocation the program always prints out "CS149 Fall 2021 Skydiving School from FirstName LastName" only once. Take screenshots of the entire program execution (including "CS149 Fall 2021 …"). It is OK to have several screenshots. The last screenshot must capture the end of program execution.

2. Any API in a multi-threaded application must be thread-safe and reentrant (e.g., call `rand_r()` instead of `rand()`). Invoking any non thread-safe or non reentrant API is subject to deduction.

3. You are not allowed to have any additional global variables other than those aforementioned global variables.

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

5. You are not allowed to call `sem_getvalue()` for any semaphore.

6. Any instructor is not aware of the number of students (`NUM_OF_STUDENTS`), nor does any instructor know how many sessions a student can request for (`NUM_OF_SESSIONS`). To avoid I/O buffer flushing bug, add "`fflush(NULL);`" after each `printf()` in the program.

7. One must invoke the random number generator each time to get any resting time, and skydiving session time. Other than skydiving a session (simulated by calling `sleep()`), do not call `sleep()` within any critical section.

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

9. No recursion is allowed in any function.

10. Best practice:

a. include necessary header files only.

b. remove any warning messages in compilation.

c. error handling: always checks the return code from any API.

What to do

a. (15 pts) screenshots of stdout output from the program execution. Screenshots that are not readable, or screenshot without "CS149 Fall 2021 …" from the program, will receive 0 point for the entire homework.

b. (5 pts) Include code snippet (not paragraphs) and describe the flow of steps performed by both instructor and student – including the exact calling sequence of synchronization constructs.

c. (60 pts) source code. Submit separate .c file.

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: a and b (note: c is in separate file)
- `skydivingschool_YourName_L3SID.c` Your source code without binaries/executables. 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:
https://guides.instructure.com/m/4212/l/352349-how-do-i-view-annotation-feedback-comments-from-my-instructor-directly-in-my-assignment-submission

NOTE: the course requires you to use Linux VM even on Mac. There are known Pthread related issues on Mac. In particular, POSIX unnamed semaphore is **not** supported on Mac. If you still use Mac for HW4, you could use *named semaphore* or *Grand Central Dispatch* instead. For named semaphore, change the data type of each semaphore from "`sem_t`" to "`sem_t *`". For Grant Central Dispatch, change the data type of each semaphore from "`sem_t`" to "`dispatch_semaphore_t`". For details, see the following links:

http://man7.org/linux/man-pages/man7/sem_overview.7.html

https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man2/sem_open.2.html

http://stackoverflow.com/questions/1413785/sem-init-on-os-x

https://developer.apple.com/documentation/dispatch/dispatchsemaphore