CSE 5031 Operating Systems
2020/21 Fall Term

Project:	5 – Part 2
Topic:	Multithreaded Race Condition
Date:	20 - 28.12.2020

Objectives:

- to develop a multi-thread application with POSIX **Pthreads** and **Semaphores**
- to synchronize threads using **state variables** and **counting semaphores**

References:

- **Lawrence Livermore National Laboratory Computing Center Pthreads tutorial portal,** https://computing.llnl.gov/tutorials/pthreads/#Pthreads
- **Linux System Programming 2d ed., Robert Love, O'Reilly 2013 (**course web site, or http://pdf-ebooks-for-free.blogspot.com.tr/2015/01/oreilly-linux-system-programming.html
- **The GNU C Library Reference Manual** (course web site, or http://www.gnu.org/software/libc/manual/pdf/libc.pdf)

Section A. Project Definition

A.1 Problem Statement

The project aims at implementing a solution to **The Sleeping Teaching Assistant** problem, a problem similar to **The Sleeping Barber** introduced in the lectures. Although solutions to both problems have lots in common, the project requires better control of the threads, you are invited to consider carefully detailed scenarios provided herein.

Project will implement the **synchronization** between two types of threads:

- ✓ **Student threads** – modelling a student who transits between **programming**, getting **TA`s advice** or **having coffee** if TA`s office is full.
- ✓ **TA thread** – modelling a teaching assistant advising students as the arrive, and taking a nap in between.

You are asked to develop a **multi-thread C** program that implements synchronization between **multiple producers** (students) and a **single consumer** (teaching assistant) that resolves race conditions using:

- → **state variables, counters** (number of active students, number of free chairs in TA`s office etc.)**,** and
- → **mutual exclusion** primitives provided by **POSIX semaphores** to preserve the integrity of shared variables.

A.2 Cooperation Scenario

A university computer engineering department has a teaching assistant (**TA**) –*single server***-** who advises students in preparing their assignments at his office if students find an available chair.

- ➢ Students go to the laboratory as they arrive (as threads are created) and start programming their project (sleep(*workperiod*)).
- ➢ After programming for a while, students go to TA`s office to get advice. But, the office is rather small and has room for only **2 chairs** for the students (**bounded buffer**), students have look for a free chair.
- ➢ If a chair is available the student takes it, notifies TA of his arrival; and waits for **TA** to start a session.
- ➢ Student whose session is started asks all the questions (sleep(*adviceperiod*)), then notifies TA that he/she has no more questions; and returns to the laboratory to resume programming.
- ➢ A student who finds that both chairs in TA`s office are occupied goes the cafeteria for a coffee break (sleep(*coffeeperiod*)), then returns to the lab to continue with programming.
- ➢ TA sits in his office as he arrives (as thread is created) and waits to be notified by a student. Once notified TA confirms student that the session has started, and waits until the student notifies him that he/she has finished. TA takes a short nap (sleep(*napperiod*)) then waits to be notified again.

A.3 Termination Scenario

You may implement several termination scenarios that ends **Student and TA threads**, and let the **main thread** to wait for their exit. The **termination strategy** you are asked to program is the decentralized approach that lets **each thread** to **exit** when it detects the end-of-session condition before starting a new cycle.

Note that a new cycle consists of:

- ✓ programming -> taking advice or going to coffee break steps for students; and
- ✓ waiting for a student -> advising -> taking a short nap steps for the TA:

In the **termination** scenario **main thread** will set the end-of-session condition for **Student threads** who will exit as they end a cycle; but the **TA thread** will continue provide its service until the last student exits.

Session termination starts when the **main thread** reads a character from keyboard and sets the **end-of-processing** condition for **Student threads**. The **main thread** waits then for the termination of all the threads, including the TA.

A **Student thread** that detects the end-of-session condition, **decrements** the number of **active students**, displays a termination message and **exits**.

TA thread has to test the **number of active** students at each cycle and **exit** if all students have already left.

A.4 Displaying the Evolution of Threads Coordination

To debug your application and prove that the coordination scenario you have implemented is working as specified, you need to **display** the **action** a thread performs at each **step**.

You need especially print out the identifier of the thread and define:

- ✓ the start and exit of a thread;
- ✓ the action simulated **before going to sleep**,
- ✓ the event **before posting it** (notifying TA for example)
- ✓ the event **after waking up**.

The example shown here after provides the trace of a test in which one instance of **Student and TA threads** run for only one cycle. You are strongly invited to **use your own words** to describe the event/action, and display the status of **other state variables** you deem necessary.

```
                         <-- TA opened his office
                         <-- TA waiting for a student
std <2145236736> entered lab
std <2145236736> programming for 10 units
std <2145236736> checking TA office
std <2145236736> entered TA office; 1 seats occupied
                         <-- TA starts advising session
std <2145236736> asking questions
std <2145236736> leaving TA office
std <2145236736> signing off; active students= 0
                         <-- TA finished advising, taking a short nap
                         <-- TA signing off
```

Section B. Implementing “student” and “TA” Transactions

B.1 Testbed Design

You will first develop the “**Student**” and “**TA**” transactions and run only **one instance** of each to debug your thread coordination process and correct it as necessary. Once the testbed operates as specified you will add more student transactions to check the correctness of your solution to race conditions.

Although not necessary, you may choose to open a second **pseudo terminal** and display the outputs of “**Student**” and “**TA**” threads on that window, as you did in the last project.

B.2 Implementation Guidelines

For test purposes you are advised to set the values of the **sleeping periods** as defined here after. Once operational you may start to change them and test your design and implementation under different conditions.

- workperiod: 10 seconds
- adviceperiod: 5 seconds
- coffeeperiod: 3 seconds
- napperiod: 1 seconds

Organize your program in **three threads** of execution, performing the actions defined here after.

a) **main thread** - *the default thread running main( ) function-* should:
   - ✓ initialize the **mutex semaphore(s),** and state variables;
   - ✓ (optional) open the **stream** to access the second pseudo terminal file in write access mode;
   - ✓ create the **Student** and **TA** threads, and display their **TID**;
   - ✓ read from **stdin** any termination character;
   - ✓ set the end-of-session condition for the **Student** threads;
   - ✓ wait for the termination of all the **threads**;
   - ✓ terminate.

b) **Student thread** should implement coordination and termination scenarios defined in sections A2 and A3.

c) **TA thread** should implement coordination and termination scenarios defined in sections A2 and A3.

B.3 Test Guidelines

Test your application creating **one** instance of “**Student**” and “**TA**” transactions. Once it works as specified capture its output as shown in section A.4 and store it in a text file named “**student1.txt**”.

Test your application with **four** instances the “**Student**” and **one** instance of “**TA**” transactions. If the application works as specified capture its output and store it in a text file named “**student4.txt**”.

Section c. Project V Part 2 Report Submission

Do not submit a result if your program does not work as specified. Name the source code as “**Prj 5 – Part 2.c**” and add a comment line in stating your name and student-id.

Store “**Prj 5 – Part 2.c**”, “**student1.txt**”, “**student4.txt**” files in the “**Prj5-Part2**” folder, located at the course web site under the tab **CSE5031 - OS Section -X/Assignment**; where “**X**” stands for (1,2,3,4) your laboratory session group.

Warning

You are encouraged to discuss the implementation procedures and general concepts behind the projects with your fellow students. However, **plagiarism is strictly forbidden**! Submitted report should be the result of **your personal work**!

Be advised that you are **accountable** of your submission not only for this project, but also for the mid-term, and final examinations. Your project grade may be reevaluated retrospectively, had you fail to answer correctly the same or a similar examination questions that you have solved with success in your submissions.