KARACHI INSTITUTE OF ECONOMICS & TECHNOLOGY

PAF KIET

Bonus Assignment

Type: Group Assignment (3 or 4 members per team)

Course: Operating Systems

Instructor: Dr. Ayaz ul Hassan Khan

Objective

In this assignment, we will practice threads and synchronizations. This assignment has several Tasks, with 100 points in total.

Problem Statement

This Assignment will implement the "Dinging Philosophers" problem with multithreading. The "Dining Philosophers" problem is invented by E. W. Dijkstra. Imagine that five philosophers who spend their lives just thinking and easting, without anything else. They will sit on a circular table with five chairs. The table has a big plate of spaghetti. However, there are only five chopsticks available, as shown in the figure 1.

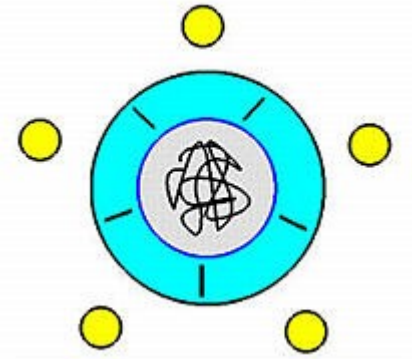

Figure 1

Task 1: (30 marks: Code Implementation = 25 marks, Report = 5 marks)

Create a main program that takes a single command line parameter, the number of threads ("nthreads") that is going to be created. You will finish this Task in several steps:

1. Interpret the parameter in order to get "nthreads". **(5 marks)**
2. Print your name and the "nthreads" in the same statement, e.g. "Ali, the input is 5 threads". **(5 marks)**
3. The main function will invoke a function void creatPhilosophers(int threadindex).
   The "nthreads" is actually the number of threads that you are going to create in your program. For each thread, creatPhilosophers() will pass the **index** of each thread to the thread function "PhilosopherThread". For instance, if you are creating 5 threads, and the thread index will be from 0 to 4. **(5 marks)**

4. After the creation, every PhilosopherThread simply prints a sentence like "This is philosopher X", where X is the actual index passed by the creatPhilosophers(). After do this, the PhilosopherThreads just return NULL. **(5 marks)**
5. After the creation, the main thread will wait for the finish of all philosopher threads using pthread_join() API. After all threads have been joined successfully, the function will print a sentence like "N threads have been joined successfully now". **(5 marks)**

Task 2: Using multiple mutexes (40 marks: Code Implementation = 30 marks, Report = 10 marks)

In this task, you are going to solve the philosopher problem with pthread's mutex APIs.
Each philosopher is in a "thinking"-"picking up chopsticks"-"eating"-"putting down chopsticks" cycle as shown below.
Thus, you can create four different functions to implement these four steps correspondingly. These functions will have the deterministic signatures as follows.

- void thinking ();
- void pickUpChopsticks (int threadIndex);
- void eating();
- void putDownChopsticks (int threadIndex);

The "pick up chopsticks" Task is the key point of this assignment. Each chopstick is shared by two philosophers and hence a shared resource. We certainly do not want a philosopher pick up a chopstick that has already been picked up by his neighbor, which will be a race condition. To address this problem, we may implement each chopstick as a mutex lock. Each philosopher, before he can eat, he should lock his left chopstick and lock his right chopstick. If the acquisitions of both locks are successful, this philosopher now owns two locks (hence two chopsticks), and can eat. After finishes easting, this philosopher invokes the function putDownChopsticks() to release both chopsticks, and exit the current thread!
Both "eating" and "thinking" functions can be easily simulated by invoking a sleep() API inside. However, we cannot utilize a determined number in the invocation of sleep(). One method is to utilize a random number Between 1 to 500. You could utilize rand () to get the number of the random value, which could be initialized use srand() to seed this random generator.
However, this simple solution may have two problems. One is starvation, another is the deadlock problem. Since each philosopher only eats once before his exit in Task2, then starvation is not an issue here. But deadlocks will be an issue. Deadlocks occur when every philosopher sits down and picks up his left chopstick at the same time? In this case, all chopsticks are locked and none of the philosophers can successfully lock his right chopstick. As a result, we have a circular waiting (i.e., every philosopher waits for his right chopstick that is currently being locked by his right neighbor), and hence a deadlock occurs.

Note: You will test your program again using different number of threads, such as 2 threads or 20 threads. You need to print out "philosopher is eating" in your eating () function, where it should be replaced with the actual index of each thread.

Task 3: Solve the deadlock problem with one lock and one conditional variable (30 marks: Order of eating & solve deadlock problem = 20 marks, Report = 10 marks)
In this task, you can solve the problem using only one mutex object. If there is only one mutex in the system, then there is no deadlock problem anymore.
For Task3, you will need to **enforce the order of eating**. The 0th philosopher should eat at first, then 1th philosopher, and so on.
The idea is to utilize a conditional variable and an index that indicates which philosopher should be the next one to eat. Initially, before creating threads, we could set the index to 0 in order for the first one to eat at first.
When every thread finishes their thinking and before they grab the chopsticks, they will check whether it is their turn to eat or not. If yes, then they will perform the eating. Otherwise, they will wait on the shared conditional variable. The thread that release the lock will wake up all threads waiting on the conditional variable. But only one of them can move forward based on the determined order, whether the index of the thread is the next one or not. If not, then those threads will have to wait again.

Note: Use 2 threads to test your results. Run this program for 1000 times again and confirms that there is no deadlock anymore. Please show the printing of the eating order as output.

Submission:

Deadline: December 14, 2018 till midnight.

Required Deliverables:

1. Task 1: source code files, a report with code explanations and screenshots of sample executions and Please test on two cases, nthreads is 5 and 20.
2. Task 2: source code, run your program 1000 times then report whether you met the deadlock problem or not. Also, please also think about when it is easier to see the deadlock, with the small or the large number of threads? Also, you need to put this reason as well in your report with proper explanation and screenshots of sample execution. If you are not sure about this, you could use your program to verify this.
3. Task 3: source code, a report with code explanation and screenshots of sample execution. **Please explain how you design your conditional variables to enforce the order of eating.**

Note: Sample code is in **Code Directory.** You should provide proper comments in source code to get full marks. Submit a zip file as bonusassignment_os_fall2018_<your student numbers>.zip in the created assignment on google classroom.

Contribution of each member should be highlighted properly in the source codes and report as tags like [By Student ID and Name].