

## **Bonus Assignment**

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

**Course: Operating Systems**

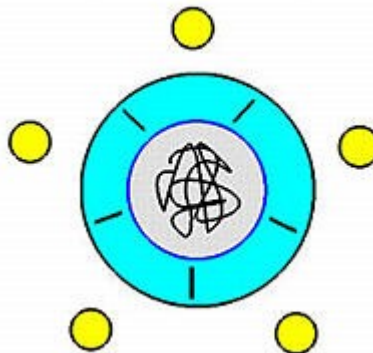
**Instructor: Dr. Ayaz ul Hassan Khan**

### **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 "Dining 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 eating, 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 eating, 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.