**CS 5523 Operating Systems: Project 2**

In this assignment, we will practice the threads and synchronizations. This assignment has several parts. This assignment has borrowed some descriptions from [this link](https://www.cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html). You may refer to this link. But **You can't copy the implementation from this link. You have to implement the project by yourself.**   
  
This project will implement the Dinging Philosophers problem using 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.   
  
Each philosopher thinks at first. When he gets hungry, he sits down and picks up the two chopsticks that are closest to him, on the left side and the right side of this philosopher. If a philosopher can pick up both chopsticks, he eats for a while. After a philosopher finishes eating, he puts down the chopsticks and leave.   
  
Now we will write a multithreaded program to simulate the behavior of phylosophers. The whole project will be divided to 3 parts. In total, this project will be 60 points in total. We will provide an additional project, part4, which counts for additional 20 points.   
  
**Part 1 (10 points)**  
Make a new directory (assignment4) and a **source** directory under it. Create a main program that takes a single command line parameter, the number of threads that is going to be created. You will finish this part in several steps:   
(1) Intepret the parameter in order to get "nthreads".   
(2) Print your name and the "nthreads" in the same statement.   
(3) The main function will invoke a function   
    void creatPhilosophers(int nthreads).   
The "nthreads" is actually the number of threads that you are going to created. For each thread, creatPhilosophers() will pass the index of threads to the thread function "PhilosopherThread".   
(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) The creatPhilosophers() 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".   
For the submission, you should output all of these results into **part1-ouput.txt**; Please test on two cases, nthreads is 5 and 50.   
Also, explain how you utilize the pthread\_join in "status.txt". The corresponding code will be in "part1.c" under the source directory. **Please don't use C++ for this project since the given link is using the C++.**   
  
**Part2: using multiple mutexes. (20 points)**   
Now you are going to simulate the whole philosopher problem.   
Each philosopher is in a "thinking"-"picking up chopsticks"-"eating"-"putting down chopsticks" cycle as shown below.   
Thus, you can creat 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" part is the key point of this project. How does a philosopher pick up chopsticks? Each chopstick is shared by two philosophers and hence a shared resource. We certainly do not want a philosopher to pick up a chopstick that has already been picked up by his neighbor, which will be a [race condition](http://stackoverflow.com/questions/34510/what-is-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! This execution flow is shown below.   
Both "eating" and "thinking" functions can be easily simulated by invoking a usleep() API inside. Please use "man usleep" to find out the signature of usleep() API. However, we can not utilize a determined number in the invocation of usleep().   
One method is to utilize a random number between 1 to 500. You could utilize random() API to get the number of values, which could be initilized use srandom() to seed this random generator. The definition of these APIs can be checked using "man random".   
However, this simple solution may have two problems. One is starvation, another is the deadlock problem. See the [link](https://www.cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html) for more descriptions of these two problems. Since each philosopher only eats once before exit, then starvation is not an issue here. But deadlocks will be. Deadlocks occurs 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.   
**Note1:** The corresponding code for this part will be in "part2.c" under the source directory. You can copy "part1.c" to create the initial version of "part2.c". You will test your program again using 5 threads and 50 threads. Also, trying to output the sequence of eating to the status.txt, such as "philosopher ith is eating" in your eating() funtion.   
**Note2:**You will run your program 100 times. Then report how many times you meet the deadlock problem in the **status.txt**   
  
**Part3: solve the deadlock problem using only one mutex object and one conditional variable (30 points)**   
In this part, you will solve the problem using only one mutex object. If there is only one mutex in the system, then there is no deadlock problem any more.   
Also, this time 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 finishs 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:** The implementation will be in "part3.c" under the source directory. Similarly, let's use 50 threads to test your results. Run this program for 20 times and confirms that there is no deadlock anymore. Also, print out the eating order and include the printing into your "status.txt" file again. **Please explain how you design your conditional variables to enforce the order of eating. You can copy the key part of your code to help your explanation.**   
  
  
**Part4: solve the deadlock problem of part2 used the pthread\_mutex\_trylock (20 points)**   
We will still create multiple mutex objects, which is the same number as that of threads (similar to part2). You can utilize any number of conditional variables if necessary.   
The idea of using pthread\_mutex\_trylock is that you may not acquire the lock successfully, which can be told by checking the value of pthread\_mutex\_trylock.   
In the end, you should confirm whether you have solved the problem or not, by running your program for 20 times. Put the idea of solving this problem into the "status.txt". Assuming you are still using 50 threads for this case.   
**Note:** You will implement the code in "part4.c" and included it in the source directory. Results of one execution will be added into "status.txt" as well.

**Submission guideline   
  
You will submit through the blackboard. Please including the "status.txt", and a "source" directory that includes all source code and Makefile.   
"source" directory: mandatory   
This directory should include all of your source code (\*.c, \*.h) and Makefile. Without the soure code, you can only get 50% points of your earned at most.   
  
  
status.txt and part1-ouput.txt: manadatory   
You should include a status report in the file status.txt. Here, the status report in status.txt should include whether you have complete different parts successfully. If successful, include the outputs of different parts, named as "output-partx". Please also give brief description of what is working and what progress was made on the part that is not working. There will be a severe penalty (50% off) if the "status.txt" is not aligned with your source code. If there is no "status.txt", you won't get any points.  
  
Please compress all of these files into a zipped file and name it as "abc123.zip". abc123 should be replaced with your abcid.**

**Based on the materials from** [**this link**](https://www.cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html)**.**