CS342301 2023 MP4 – Dining Philosophers

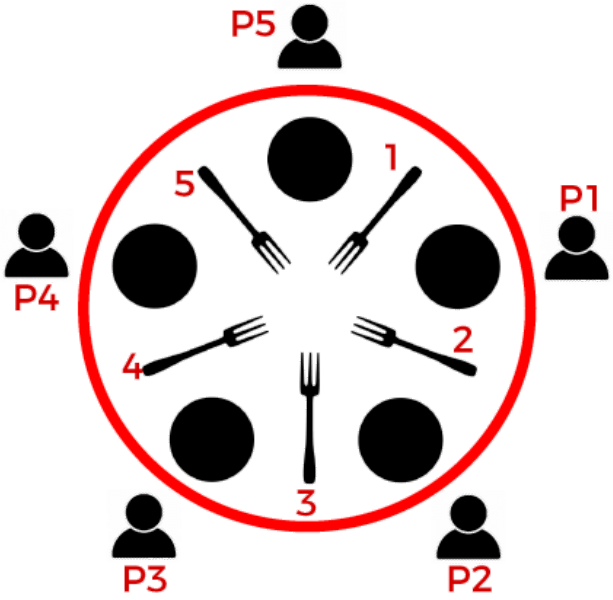
Deadline: 2023/06/01 23:59

I. Goal

1. Understand how to work in the Linux environment.
2. Understand how multi-thread programming can be done with the pthread library.
3. Understand the concept of deadlock and starvation.

II. Description

The dining-philosophers problem is a classic synchronization problem in computer science. In this project, there are five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, and the table is laid with five single forks. There's a need to allocate the forks among the philosophers in a deadlock-free and starvation-free manner.



Dining Philosophers Problem

The program starts with 5 Philosopher threads, 5 Fork threads and 1 Table thread.

The philosopher thread runs in a loop that he alternates between thinking and eating. When the philosopher tries to eat, he will pick up the two forks that are closest to him (i.e. the forks that are between him and his left and right neighbors). If the fork is already taken, the philosopher waits until the fork becomes available.

The fork thread simulates the fork on the table. If the fork is free, the philosopher can just pick up the fork. Otherwise (the fork is taken), it will send a wait to the other philosopher who wants to pick up, and signal him when the fork is released.

The table thread controls the number of philosophers that are able to pick up the forks and eat.

III. Assignment

a. Code Structure

1. Philosopher (philosopher.hpp): Philosopher runs in a thread that repeatedly thinks and picks up the forks to eat.
   - think: sleep for a random period between one and four seconds to simulate thinking.
   - eat: sleep two seconds to simulate eating.
2. Fork (fork.hpp): Fork runs in a thread that ensures the fork is only taken by one philosopher.
3. Table (table.hpp): Table runs in a thread that controls the number of philosophers that try to pick up the forks and eat.
4. Config (config.hpp): Define the parameters. (The number of philosophers and the time they spend eating and thinking)

b. Implementation

1. Philosopher
   - Complete the run routine.
   - Implement the interface to enter / leave the table.
   - Implement the interface to pick up / put down the forks.
   - Note: The left fork id is the same as his id, and the right fork id = left fork id + 1, except the last philosopher is 0. (E.g. Philosopher 0: 0, 1. Philosopher 1: 1, 2. Philosopher 4: 4, 0)
2. Fork: Implement a mutex to avoid two philosophers using the fork simultaneously.
   - Note: You should use the pthread mutex lock and condition variable to implement the mutex.
3. Table: Implement a semaphore to ensure there are at most 4 philosophers trying to pick up the forks and eat.
   - Note: You should use the pthread mutex lock and condition variable to implement the semaphore.
   - The philosopher who is not entering the table cannot pick up any forks.
4. main.cpp: Implement your main function. The main function will create and start the philosophers at the beginning, and send termination signals to all of them after running a period of time, and it ends after all the philosophers terminate.
5. Bonus (20%)
   - Although finishing the above implementation, it is possible that a philosopher never has a chance to pick up the forks and eat, which leads to starvation.
   - Try to solve the starvation problem. You need to explain your

implementation in detail and prove its correctness.

c. Notes & Hints

1. Search for all **"TODO"**, that's all you need to do.
2. It is allowed to implement code in .hpp files or in .cpp files, but you should make sure that the Makefile is working properly. Also, we will replace all the .hpp files when verifying your code, so make sure your implementation is in the .cpp files, and they also work fine with the original .hpp files.
3. You need to implement your own test files for unit testing or verifying the complete implementation.
4. You can only use mutex lock and cv to implement the Fork class and the Table class. Accessing other philosophers' state or adding a global object to control all the philosophers' state are forbidden (E.g. monitor). You will get 0 points for violating this rule.

d. Compile, Run and Test

1. Copy the code for MP4 to a new folder
   ```
   $ cp -r /home/os2023/share/Dining-Philosopher .
   ```
2. Compile and run your implementation.
   ```
   $ make
   $ ./main
   ```
3. There's no sample test cases.

IV. Grading

a. Implementation Correctness - 25%

1. Must implement this homework through the class we supplied.
2. Must use Pthread library to implement the mutex lock and semaphore.
3. Remember to adjust all defined parameters back to the original value
4. Remember to remove your additional debug message, or you will get some points deduction.

b. Report - 50%

1. Cover page, including student ID.
2. Explain your implementation of the class Fork, Table and Philosopher. (Please tell us as much detail as possible.)
3. Explain the following questions below:
   1. Why does the function pthread_cond_wait() need a mutex variable as second parameter, while function pthread_cond_signal() does not?
   2. Which part of the implementation ensures the fork is only used by one philosopher at a time ? How ?
   3. Which part of the implementation avoids the deadlock (i.e. philosophers are all waiting for the forks to eat) happen? How does it avoid this?

4. After finishing the implementation, does the program is starvation-free? Why or why not?
5. What is the purpose of using pthread_setcancelstate() ?

4. Explain the pros and cons of using monitor to solve a dining-philosopher problem compared to this homework ? (You should at least analyze the performance and scalability.)
5. Any feedback you would like to let us know.
6. Upload to eeclass with the filename MP4_report_[StudentID].pdf (E.g. MP4_report_111062584.pdf)

c. Demo - 25%

1. We will ask several questions about your codes.

2. Demos will take place on our server, so you are responsible to make sure your code works on our server.

d. Notes

1. TAs won't help you recover from file loss or misedited. Please make sure you backup your code. You can use git or simply make a copy in your local.
2. **Late submissions will not be accepted.** Refer to the course syllabus for detailed homework rules and policies.
3. **Plagiarism**
   a. **NEVER SHOW YOUR CODE** to others.
   b. If the codes are similar to other people (including your upperclassman) and you can't answer questions properly during the demo, you will be identified as plagiarism.We will ask several questions about your codes.