Operating System
CS 370
Fall 2016

Project: 5
Dining Philosophers Using Threads

Due Date: Thursday, 17th November, 2016
Must be submitted before midnight

Total Points: 100

Teaching Assistant: Pradip Singh Maharjan
Office Hours:
MW: 2:00 pm to 4:00 pm
TuTh:10:00 am to 12:00 pm

Course Web Page : `http://osserver.cs.unlv.edu/moodle/`

Requirements:

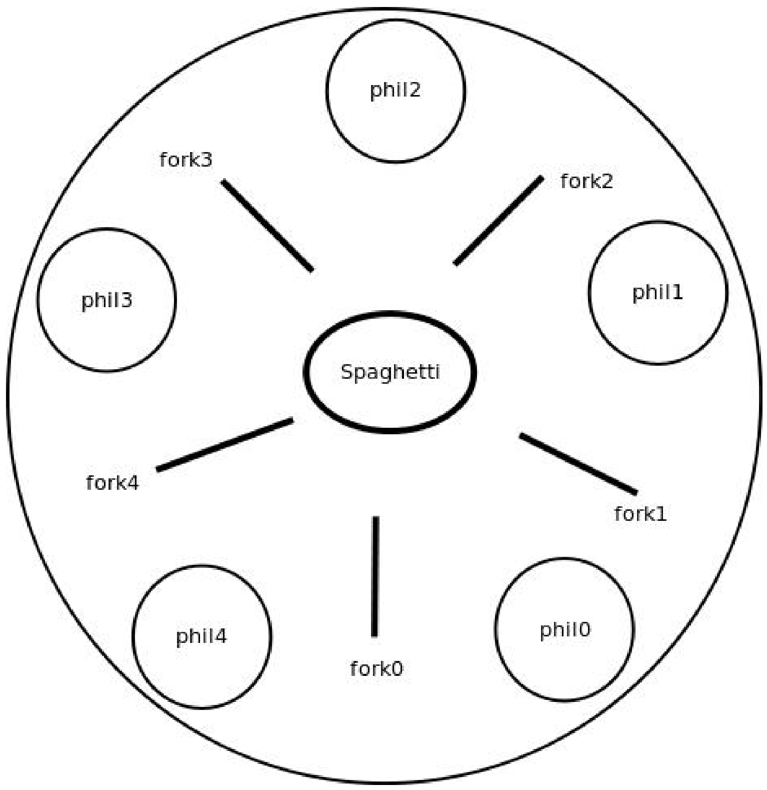
- Must be implemented in C (not C++)
- Must compile on cardiac.cs.unlv.edu
- First submittal of this project must compile as “gcc project5a.c -pthread”. Similarly, second or third submittals must compile as “gcc project5b.c -pthread” or “gcc project5c.c -pthread” respectively.
- Project must be submitted through website ONLY.
- Argument to the executable should control the program

Purpose:

The purpose of this project is to gain more experience with multi threaded programming. You will use semaphore or mutex for critical sections (to access resources forks and food). You will get to experience a simulation of the algorithm that is covered in your course.
This project is more or less similar to project-4 in terms of implementation. You will use similar technique for creating threads and using semaphores.

The Dining Philosophers Scenario :

The dining philosophers scenario is a classic which is structured as follows. Five philosophers, numbered zero to four, are sitting at a round table, thinking. As time passes, different individuals become hungry and decide to eat. There is a platter of spaghetti on the table but each philosopher only has one fork to use. In order to eat, they must share forks. The fork to the left of each philosopher (as they sit facing the table) has the same number as that philosopher.



Deadlock :

An actual deadlock occurs when every philosopher is holding his own fork and waiting for the one from his neighbor to become available:

- Philosopher zero is holding fork zero, but is waiting for fork one
- Philosopher one is holding fork one, but is waiting for fork two
- Philosopher two is holding fork two, but is waiting for fork three
- Philosopher three is holding fork three, but is waiting for fork four
- Philosopher four is holding fork four, but is waiting for fork zero
- In this situation, nobody can eat and the philosophers are in a deadlock.

Description:

For this project you will implement Dining Philosophers problem , observe the Deadlock situation and provide solution (one of the solution) for the same as discussed in the class. You are to use a thread for each philosopher. This will be a shared memory system and you will need to simulate this. You have to use the flag in your program to demonstrate two situation in your program. First, Deadlock observation (nobody can eat and the philosophers are in a deadlock) ; Second, Limiting the number of philosophers entering the dining room to four (i.e one less value than the total philosophers).

Potential or Actual Deadlock Situation:

In this situation, nobody can eat and the philosophers are in a deadlock. Rerun the program a number of times and you will see that the program may sometimes hang, or run to completion at other times. Now, put some delay (sleep()) after taking left fork, so that it will make higher chances of having deadlock situation. There will be same number of forks as philosophers. Also, assume there are enough food for all the philosophers. You need to use semaphore for each fork to access it.
Here is the pseudocode for each Philosoher thread method (for no deadlock free strategy):

```
semaphore array[0..4] fork <- [1,1,1,1,1]

loopforever
p1: think
p2: wait(fork[i])
p3: wait(fork[i+1])
p4: eat
p5: signal(fork[i])
p6: signal(fork[i+1])
```

Limiting the number of Philosophers entering the room to four:

In this strategy, you will limit the number of philosophers entering the dining room to be less than the number of philosophers(i.e four). The following pseudo-code shows the logic for each philosopher thread method (for deadlock free strategy):

```
semaphore array[0..4] fork <- [1,1,1,1,1]
semaphore room <- 4

loop forever
p1: think
p2: wait(room)
p3: wait(fork[i])
p4: wait(fork[i+1])
p5: eat
p6: signal(fork[i])
p7: signal(fork[i+1])
p8: signal (room)
```

Ideally, the food never ends and philosopher's can eat at anytime and as much as they want. But, for simulation purpose we will allow each philosopher to eat only 5 times. This allows us to terminate the program in finite time and see the output events for all philosophers.

Input Specification:

There will be no input file for this. Simply assume there are 5 philosophers.Implemenent the two different strategies described above with correct argument/flag in your program. If you set enable_room flag it should simulate strategy2.

Sample Input/Output:

Here is sample output for potential deadlock situation: (After compilation, you will use command "./a.out 1" to observe Deadlock)

```
Philosopher 1: taking left fork 1
Philosopher 1: taking right fork 2
Philosopher 1: EATING.
Philosopher 1: putting down left fork 1
Philosopher 2 is done thinking and now ready to eat.
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
Philosopher 2: EATING.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
Philosopher 2: EATING.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
Philosopher 0 is done thinking and now ready to eat.
Philosopher 0: taking left fork 0
Philosopher 0: taking right fork 1
Philosopher 0: EATING.
Philosopher 0: putting down left fork 0
Philosopher 0: putting down right fork 1
Philosopher 0: taking left fork 0
Philosopher 0: taking right fork 1
Philosopher 2: EATING.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
Philosopher 0: EATING.
Philosopher 0: putting down left fork 0
Philosopher 0: putting down right fork 1
Philosopher 0: taking left fork 0
Philosopher 0: taking right fork 1
Philosopher 1: putting down right fork 2
Philosopher 4 is done thinking and now ready to eat.
Philosopher 4: taking left fork 4
Philosopher 2: EATING.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
Philosopher 2: EATING.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
Philosopher 2: EATING.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 3 is done thinking and now ready to eat.
Philosopher 0: EATING.
Philosopher 0: putting down left fork 0
Philosopher 0: putting down right fork 1
Philosopher 3: taking left fork 3
Philosopher 4: taking right fork 0
Philosopher 4: EATING.
Philosopher 4: putting down left fork 4
Philosopher 3: taking right fork 4
Philosopher 1: taking left fork 1
Philosopher 4: putting down right fork 0
Philosopher 3: EATING.
Philosopher 3: putting down left fork 3
Philosopher 3: putting down right fork 4
Philosopher 3: taking left fork 3
Philosopher 0: taking left fork 0
Philosopher 4: taking left fork 4
Philosopher 2: taking left fork 2
(hang)
```

Here is the sample output for second strategy (with room semaphore). This is just one instance of many outputs: (After compilation, you will use command "./a.out 0" to observe this output)

```
Philosopher 1 is done thinking and now ready to eat.
Philosopher 1: taking left fork 1
Philosopher 1: taking right fork 2
PHILOSOPHER 1: EATING.
Philosopher 1: putting down left fork 1
Philosopher 1: putting down right fork 2
Philosopher 0 is done thinking and now ready to eat.
Philosopher 0: taking left fork 0
Philosopher 0: taking right fork 1
Philosopher 4 is done thinking and now ready to eat.
Philosopher 4: taking left fork 4
Philosopher 2 is done thinking and now ready to eat.
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
PHILOSOPHER 2: EATING.
Philosopher 1 is done thinking and now ready to eat.
Philosopher 3 is done thinking and now ready to eat.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 2 is done thinking and now ready to eat.
Philosopher 3: taking left fork 3
PHILOSOPHER 0: EATING.
Philosopher 0: putting down left fork 0
Philosopher 0: putting down right fork 1
Philosopher 0 is done thinking and now ready to eat.
Philosopher 1: taking left fork 1
Philosopher 1: taking right fork 2
PHILOSOPHER 1: EATING.
Philosopher 1: putting down left fork 1
Philosopher 1: putting down right fork 2
Philosopher 1 is done eating.
Philosopher 4: taking right fork 0
PHILOSOPHER 4: EATING.
Philosopher 4: putting down left fork 4
Philosopher 4: putting down right fork 0
Philosopher 4 is done thinking and now ready to eat.
Philosopher 3: taking right fork 4
Philosopher 0: taking left fork 0
Philosopher 0: taking right fork 1
PHILOSOPHER 0: EATING.
Philosopher 0: putting down left fork 0
Philosopher 0: putting down right fork 1
Philosopher 0 is done eating.
PHILOSOPHER 3: EATING.
Philosopher 3: putting down left fork 3
Philosopher 3: putting down right fork 4
Philosopher 3 is done thinking and now ready to eat.
Philosopher 3: taking left fork 3
Philosopher 4: taking left fork 4
Philosopher 4: taking right fork 0
PHILOSOPHER 4: EATING.
Philosopher 4: putting down left fork 4
Philosopher 4: putting down right fork 0
Philosopher 4 is done eating.
Philosopher 3: taking right fork 4
PHILOSOPHER 3: EATING.
Philosopher 3: putting down left fork 3
Philosopher 3: putting down right fork 4
Philosopher 3 is done eating.
Philosopher 2: taking left fork 2
Philosopher 2: taking right fork 3
PHILOSOPHER 2: EATING.
Philosopher 2: putting down left fork 2
Philosopher 2: putting down right fork 3
Philosopher 2 is done eating.
```

Point Breakdown:

- Implementing correct argument/flag to simulate different strategy [15 pts]
- Implementing the Deadlock strategy correctly [35 pts]
- Implementing limiting philosophers with room semaphore [35 pts]
- Coding style
  - Proper and informative comments [5 pts]
  - Informative variable names [5 pts]
  - Code organization (methods are not too long) [5 pts]