University of Central Punjab

(Incorporated by Ordinance No. XXIV of 2002 promulgated by Government of the Punjab)

FACULTY OF INFORMATION TECHNOLOGY

Operating System

Thread Synchronization

Classical Problems- Dining Philosophers Problem

Lab No 9

Faculty of Information Technology

UCP Lahore Pakistan

Designed By: Zaid Munir

Classical Synchronization Problem In the previous four labs we have learnt in detail about pthread and synchronization constructs that enable synchronization amongst threads. The objective of today's lab is to apply all that knowledge and code out a classical problem of synchronization, "The dining philosopher problem".

Problem

The dining philosophers problem was invented by E. W. Dijkstra. Imagine that five philosophers who spend their lives just thinking and eating. In the middle of the dining room is 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 following figure. Each philosopher thinks. When he gets hungry, he sits down and picks up the two chopsticks that are closest to him. If a philosopher can pick up both chopsticks, he eats for a while. After a philosopher finishes eating, he puts down the chopsticks and starts to think.

Analysis

How do we write a threaded program to simulate philosophers? First, we notice that these philosophers are in a thinking-picking up chopsticks-eating-putting down chopsticks cycle as shown below.

The "pick up chopsticks" part is the key point. How does a philosopher pick up chopsticks? Well, in a program, we simply print out messages such as ``Have left chopsticks'', which is very easy to do. The problem is 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. This is a race condition. To address this problem, we may consider each chopstick as a shared item protected by a mutex lock. Each philosopher, before he can eat, locks his left chopstick and locks his right chopstick. If the acquisitions of both locks are successful, this philosopher now owns two locks (hence two chopsticks), and can eat. After finished eating, this philosopher releases both chopsticks, and thinks! This execution flow is shown below.

Because we need to lock and unlock a chopstick, each chopstick is associated with a mutex lock. Since we have five philosophers who think and eat simultaneously, we need to create five threads, one for each philosopher. Since each philosopher must have access to the two mutex locks that are associated with its left and right chopsticks, these mutex locks are global variables.

Tech specs:

You are to write a program that simulates the above problem using pthreads. In order to tackle the problem, the following things should be kept in mind:

- Initially all philosophers are hungry
- He picks up his left chopstick first and then the right one.

• Once he has picked up both chopsticks he eats for a random amount of time less than 30 seconds.

• Once done he drops both chopsticks and starts thinking again for a random amount of time less than 30 seconds.

• All philosophers repeat this task for Max_meals number of times after which they exit. Max_meals should be provided via command line to your program. This variable puts a bound on the total number of meals that can be had in the program. See sample output for clarification.

• Assign each philosopher with a number

o Philosopher I can use chopstick[i] and chopstick[i+1]

o Special case: Philosopher 4 can use chopstick [4] and chopstick [0] (Circular array)

Technical discussion

• Read the man pages for sleep () and usleep() and decide which of these you will use for your solution and why?

• When you use rand () don't forget to use srand()

• Figure out where and why you may use sched_yield()

• You should compile your code using Makefile . (This is a compulsion)

Here are some very important facts about this program:

1. The above program forces each philosopher to pick up and put down his left chopstick, followed by his right one. This is also for the purpose of simplicity. In fact, it is easy to see that the order of putting down the chopsticks is irrelevant. Try to reasoning about this yourself.

2. The most serious problem of this program is that deadlock would occur!

What if every philosopher sits down about the same time and picks up his left chopstick as shown in the following figure? 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. Modify your solution such that deadlock does not occur. Explain how you can achieve that?

Starvation is also a problem! Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they

have been sitting for a long time, they have no chance to eat. This is a starvation. Note that it is not a deadlock because there is no circular waiting, and everyone has a chance to eat!

The above shows a simple example of starvation. You can find more complicated thinking-eating sequence that also generates starvation. Please try.

Sample output with Max_meals = 20

philosopher 1: I am going to eat!
philosopher 3: I am going to eat!
philosopher 2: I am going to eat!
philosopher 0: I am going to eat!
philosopher 0: left=1
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 2: right=1
Philosopher 0: I got the left one!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 4: I am going to eat!
philosopher 4: left=1
Philosopher 4: I got the left one!
philosopher 3: left=0
philosopher 1: right=0
Philosopher 1: I cannot get the right one!
philosopher 4: right=0
Philosopher 4: I cannot get the right one!
philosopher 0: right=1
Philosopher 0: I got two chopsticks!
philosopher 0: I am eating!
Philosopher 2: I got two chopsticks!
philosopher 2: I am eating!
Philosopher 3: I cannot even get the left chopstick!
philosopher 3: I am going to eat!
philosopher 3: left=0

Philosopher 3: I cannot even get the left chopstick!
philosopher 1: I am going to eat!
philosopher 1: left=0
Philosopher 1: I cannot even get the left chopstick!
philosopher 2: I am going to eat!
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 2: right=1
Philosopher 2: I got two chopsticks!
philosopher 2: I am eating!
philosopher 3: I am going to eat!
philosopher 3: left=0
Philosopher 3: I cannot even get the left chopstick!
philosopher 4: I am going to eat!
philosopher 4: left=1
Philosopher 4: I got the left one!
philosopher 4: right=1
Philosopher 4: I got two chopsticks!
philosopher 4: I am eating!
philosopher 0: I am going to eat!
philosopher 0: left=1
Philosopher 0: I got the left one!
philosopher 0: right=1
Philosopher 0: I got two chopsticks!
philosopher 0: I am eating!
philosopher 0: I am going to eat!
philosopher 0: left=1
Philosopher 0: I got the left one!
philosopher 0: right=1
Philosopher 0: I got two chopsticks!
philosopher 0: I am eating!
philosopher 3: I am going to eat!

philosopher 3: left=1
Philosopher 3: I got the left one!
philosopher 3: right=1
Philosopher 3: I got two chopsticks!
philosopher 3: I am eating!
philosopher 1: I am going to eat!
philosopher 1: left=0
Philosopher 1: I cannot even get the left chopstick!
philosopher 4: I am going to eat!
philosopher 4: left=0
Philosopher 4: I cannot even get the left chopstick!
philosopher 1: I am going to eat!
philosopher 1: left=0
Philosopher 1: I cannot even get the left chopstick!
philosopher 2: I am going to eat!
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 2: right=0
Philosopher 2: I cannot get the right one!
philosopher 4: I am going to eat!
philosopher 4: left=1
Philosopher 4: I got the left one!
philosopher 4: right=1
Philosopher 4: I got two chopsticks!
philosopher 4: I am eating!
philosopher 2: I am going to eat!
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 2: right=1
Philosopher 2: I got two chopsticks!
philosopher 2: I am eating!
philosopher 1: I am going to eat!

philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 1: right=0
Philosopher 1: I cannot get the right one!
philosopher 0: I am going to eat!
philosopher 0: left=0
Philosopher 0: I cannot even get the left chopstick!
philosopher 1: I am going to eat!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 1: right=0
Philosopher 1: I cannot get the right one!
philosopher 0: I am going to eat!
philosopher 0: left=0
Philosopher 0: I cannot even get the left chopstick!
philosopher 2: I am going to eat!
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 2: right=1
Philosopher 2: I got two chopsticks!
philosopher 2: I am eating!
philosopher 3: I am going to eat!
philosopher 3: left=0
Philosopher 3: I cannot even get the left chopstick!
philosopher 0: I am going to eat!
philosopher 0: left=1
Philosopher 0: I got the left one!
philosopher 0: right=1
Philosopher 0: I got two chopsticks!
philosopher 0: I am eating!
philosopher 1: I am going to eat!
philosopher 1: left=0

Philosopher 1: I cannot even get the left chopstick!
philosopher 4: I am going to eat!
philosopher 4: left=1
Philosopher 4: I got the left one!
philosopher 4: right=1
Philosopher 4: I got two chopsticks!
philosopher 4: I am eating!
philosopher 2: I am going to eat!
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 2: right=1
Philosopher 2: I got two chopsticks!
philosopher 2: I am eating!
philosopher 3: I am going to eat!
philosopher 3: left=0
Philosopher 3: I cannot even get the left chopstick!
philosopher 0: I am going to eat!
philosopher 0: left=0
Philosopher 0: I cannot even get the left chopstick!
philosopher 1: I am going to eat!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 1: right=0
Philosopher 1: I cannot get the right one!
philosopher 1: I am going to eat!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 1: right=0
Philosopher 1: I cannot get the right one!
philosopher 4: I am going to eat!
philosopher 4: left=1
Philosopher 4: I got the left one!

philosopher 4: right=1
Philosopher 4: I got two chopsticks!
philosopher 4: I am eating!
philosopher 3: I am going to eat!
philosopher 3: left=1
Philosopher 3: I got the left one!
philosopher 3: right=0
Philosopher 3: I cannot get the right one!
philosopher 1: I am going to eat!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 1: right=1
Philosopher 1: I got two chopsticks!
philosopher 1: I am eating!
philosopher 0: I am going to eat!
philosopher 0: left=1
Philosopher 0: I got the left one!
philosopher 0: right=0
Philosopher 0: I cannot get the right one!
philosopher 4: I am going to eat!
philosopher 4: left=1
Philosopher 4: I got the left one!
philosopher 4: right=1
Philosopher 4: I got two chopsticks!
philosopher 4: I am eating!
philosopher 1: I am going to eat!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 1: right=1
Philosopher 1: I got two chopsticks!
philosopher 1: I am eating!
philosopher 2: I am going to eat!

philosopher 2: left=0
Philosopher 2: I cannot even get the left chopstick!
philosopher 2: I am going to eat!
philosopher 2: left=0
Philosopher 2: I cannot even get the left chopstick!
philosopher 0: I am going to eat!
philosopher 0: left=0
Philosopher 0: I cannot even get the left chopstick!
philosopher 0: I am going to eat!
philosopher 0: left=0
Philosopher 0: I cannot even get the left chopstick!
philosopher 3: I am going to eat!
philosopher 3: left=1
Philosopher 3: I got the left one!
philosopher 3: right=1
Philosopher 3: I got two chopsticks!
philosopher 3: I am eating!
philosopher 2: I am going to eat!
philosopher 2: left=1
Philosopher 2: I got the left one!
philosopher 2: right=0
Philosopher 2: I cannot get the right one!
philosopher 1: I am going to eat!
philosopher 1: left=1
Philosopher 1: I got the left one!
philosopher 1: right=1
Philosopher 1: I got two chopsticks!
philosopher 1: I am eating!
philosopher 0: I am going to eat!
philosopher 0: left=1
Philosopher 0: I got the left one!
philosopher 0: right=1

Philosopher 0: I got two chopsticks!
philosopher 0: I am eating!
Philosopher 2 has finished the dinner and is leaving!
Philosopher 1 has finished the dinner and is leaving!
Philosopher 4 has finished the dinner and is leaving!
Philosopher 3 has finished the dinner and is leaving!
Philosopher 0 has finished the dinner and is leaving!
Philosopher 0 ate 5 meals.
Philosopher 1 ate 3 meals.
Philosopher 2 ate 5 meals.
Philosopher 3 ate 2 meals.
Philosopher 4 ate 5 meals.
main(): The philosophers have left. I am going to exit!

For The implementation of this lab you also need a help of semaphore POSIX semaphores having following operations use in this lab.

sem_init ()
sem_wait ()
sem_post ()
sem_getvalue ()
sem_destroy ()

1.1. How to create a semaphore

All POSIX semaphore functions and types are prototyped or defined in "semaphore.h". To define a semaphore object, use

sem_t sem_name; ⟵ **Semaphore variable name**

Data type of semaphore

1.2. How to initialize semaphore

To initialize a semaphore, use sem_init ();

int sem_init (sem_t *sem, int pshared, unsigned int value);

- sem points to a semaphore object to initialize
- pshared is a flag indicating whether or not the semaphore should be shared with forked () processes.
- Value is an initial value to set the semaphore to

Example of use:

sem_init (&sem_name, 0, 10);

1.3. How to use wait and signal command

To wait on a semaphore, use sem_wait();

int sem_wait(sem_t *sem);

Example of use:

sem_wait(&sem_name);

If the value of the semaphore is negative, the calling process blocks; one of the blocked processes wakes up when another process calls sem_post().

To increment the value of a semaphore, use sem_post:

int sem_post(sem_t *sem);

Example of use:

sem_post(&sem_name);

It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

1.4. How to get semaphore current value

To find out the value of a semaphore, use

int sem_getvalue(sem_t *sem, int *value);

gets the current value of sem and places it in the location pointed to by value

Example of use:

```
int value;

sem_getvalue(&sem_name, &value);

printf("The value of the semaphors is %d\n", value);
```

1.5. How to destroy semaphore

To destroy a semaphore, use

int sem_destroy(sem_t *sem);

destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

```
sem_destroy(&sem_name);
```