

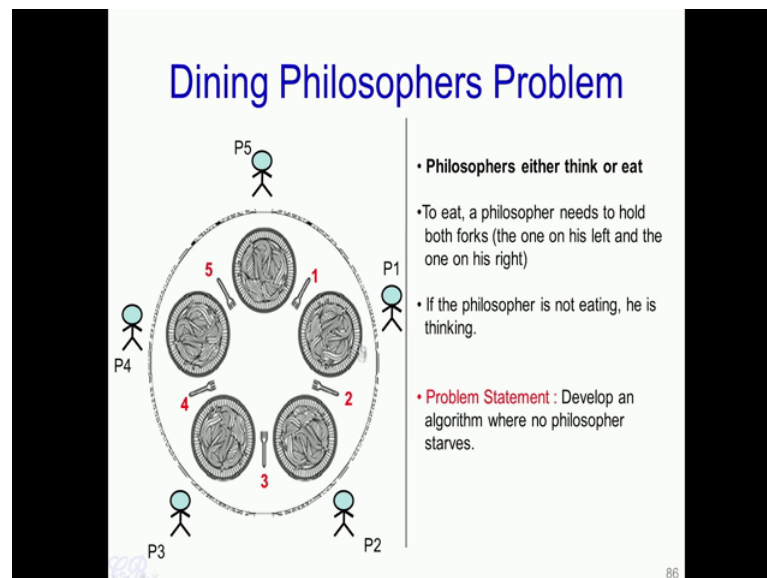
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 07
Lecture – 30
Dining Philosophers Problem

Hello. In the previous video, we had talked about semaphores. We had seen how semaphores could be used to solve synchronization problems in the producer consumer example that we had taken.

In this video, we will look at other problem, where semaphores are useful. So, this is the dining philosophers' problem and it is a classic example of the use of semaphores.

(Refer Slide Time: 00:54)



Let us start with the problem. Let us say we have five philosophers P 1, P 2, P 3, P 4, P 5 who are sitting around a table. Now in front of them, there are five plates; one for each of the philosophers; and five forks 1, 2, 3, 4, 5. Now each of these philosophers could do just one of two things. Each philosopher could either think or eat.

In other words, if the philosopher is not thinking then he is eating and vice versa. Now, in order to eat, a philosopher needs to hold both forks that is the fork on his left and the one on the right that is for instance if p 1 wants to eat then he needs to have the fork 1

and fork 2. These are the two forks, which are close to him. Similarly, if P 3 wants to eat then forks 4 and forks 3 are required. Now the problem is or the problem what we are trying to solve is to develop an algorithm, where no philosopher starves that is every philosopher should eventually get a chance to eat.

(Refer Slide Time: 02:30)

First Try

```
#define N 5

void philosopher(int i){
  while(TRUE){
    think(); // for some_time
    take_fork(R);
    take_fork(L);
    eat();
    put_fork(L);
    put_fork(R);
  }
}
```

What happens if only philosophers P1 and P3 are always given the priority?
P4, P5, and P2 starves... so scheme needs to be fair

88

Let us start with the very naive solution to this particular problem. Let us say we have a solution over here, where we define N as 5 corresponding to each philosopher. And we have a function for philosopher. So, this function takes integer value i and this i could be values of 1 to 5 corresponding to each philosopher that is P 1, P 2, P 3, P 4 or P 5.

Now in the function, we have an infinite loop, where the ith philosopher will think for some time and then after some time he begins to feel hungry. So, he will take the fork on his right then take the fork on his left then he is going to eat for some time. And after that he is going to put down the left fork, and then put down the right fork, and this continues in a loop infinitely.

So, for instance philosophy for P 1 will think for some time then feel hungry then he would pick up the right fork that is the fork number 1 then pick up the left fork that is fork 2 then eat for some time and put down the forks first 2 and then 1. So, this seems like a very simple and easy solution to the problem. But as we will see that there are certain issues that could crop up; the issues come because each of these philosophers,

which essentially execute this function independently, or thinking, and feeling hungry, and eating all independently.

Let us consider this particular scenario. Let us say the philosophers P 1 and P 3 have a higher priority that is whenever the request for the fork to be picked up then the system will always ensure that they are given the forks. So, what would happen in such a case so we will get a case where P 1 eats whenever he wants, and P 3 eats whenever he wants, while the other philosophers P 2, P 4 and P 5 which have the lower priority in picking the fork will not be able to eat. For instance, the philosopher P 2 neither could pick up the right fork or the left fork, and therefore, P 2 cannot eat.

In a similar manner, P 4 and P 5 have just one fork between them, which they could possibly pick up; the other fork in each case would with high probability be given to the philosophers P 1 and P 3. Thus P 2, P 4 and P 5 will starve; and this is not the ideal solution for our problem.

(Refer Slide Time: 05:56)

First Try

```
#define N 5  
void philosopher(int i){  
    while(TRUE){  
        think(); // for some_time  
        take_fork(R);  
        take_fork(L);  
        eat();  
        put_fork(L);  
        put_fork(R);  
    }  
}
```

What happens if all philosophers decide to pick up their right forks at the same time?
Possible starvation due to deadlock

89

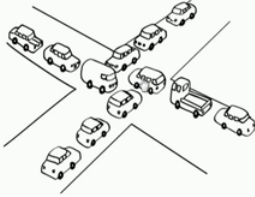
Let us see another possible issue that could take place. Let us say by some chance all the philosophers' pickup their right fork simultaneously. So, we have philosopher P 1 picking up his right fork, philosopher P 2 picking the right fork, P 3, P 4, P 5 pickup the right fork respectively.

Now, in order to eat each of the philosophers have to pick up their left fork; and this could lead to a starvation. Essentially P 1 is waiting for P 2 to put down the fork, so that he could pick it up. Then P 2 is waiting for P 3 to put down the fork; P 3 is waiting for P 4 to put down the fork; P 4 is waiting to for P 5; and P 5 is waiting for P 1. So, essentially we see that every philosopher is waiting for another philosopher that is creating a chain. And this waiting will go on infinitely leading to starvation, which we often call as a deadlock.

(Refer Slide Time: 07:19)

Deadlocks

- A situation where programs continue to run indefinitely without making any progress
- Each program is waiting for an event that another process can cause



The diagram illustrates a deadlock at a four-way intersection. Four roads meet at a central point. On each road, a line of cars is moving towards the center. Each car is positioned such that it is blocking the path of another car from the perpendicular road. For example, a car from the top road is blocking the path of a car from the right road, which is blocking the path of a car from the bottom road, which is blocking the path of a car from the left road, which is blocking the path of a car from the top road. This circular dependency prevents any car from moving forward, creating a deadlock. The number '90' is visible in the bottom right corner of the slide.

So, to define a deadlock more formally; a deadlock is a situation, where programs continue to run indefinitely without making any progress. Each program is waiting for an event that another program or process can cause. So, you see in this case each of the philosophers is waiting for a particular event that is putting down the fork, which other philosophers should do. So, there is a circular wait that is present and this leads to a deadlock that by starvation.

(Refer Slide Time: 08:07)

Second try

Imagine,
All philosophers start at the same time
Run simultaneously
And think for the same time
This could lead to philosophers taking fork and putting it
down continuously: a deadlock.

```
while(TRUE){  
    think();  
    take_fork(R);  
    if (available(L)){  
        take_fork(L);  
        eat();  
        put_fork(R);  
        put_fork(L);  
    }else{  
        put_fork(R);  
        sleep(T)  
    }  
}
```

91

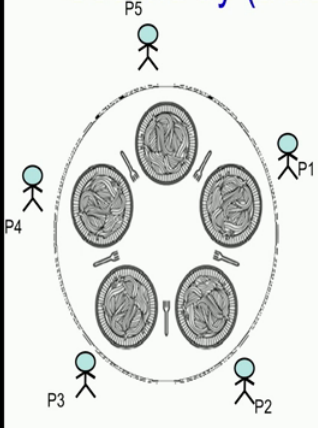
Let us look at another attempt to solve this particular problem. Let us say we have the same function over here. And this time the philosopher takes the right fork, then he would determine if the left fork is available; if the left fork is available the philosopher would take the left fork eat for sometime then put down both the forks the right as well as the left, and the loop continues as we showed.

However, if the left fork is not available, then we go to the else part and the philosopher will put back the right fork. Essentially, the fork which was picked up over here the right fork would be put back on to the table, if the philosopher finds out that the left fork is not available, so this will allow another philosopher to probably eat. And after this is done there is a sleep for some fixed interval T before the philosopher tries again.

Let us see what is the issue with this particular case. Let us consider is particular scenario where all philosophers start at exactly the same time, they run simultaneously and think for exactly the same time. So, this could lead to a situation where the philosophers pick up their fork all simultaneously then, they find out that their left forks are not available, so they put down their forks simultaneously, then they sleep for some time and then they repeat the process. So, you see that the five philosophers are again starved they will be continuously just picking up their right fork and putting it back onto the table. So, this solution is not also going to solve our purpose, since we have the philosophers starving again.

(Refer Slide Time: 10:26)

Second try (a better solution)



```
#define N 5

void philosopher(int i){
while(TRUE){
think();
take_fork(R);
if (available(L)){
take_fork(L);
eat();
put_fork(L);
put_fork(R);
}else{
put_fork(R);
sleep(random_time);
}
}
```

92

A slightly better solution to this case is where instead of sleeping for a fixed time, the philosopher would put down the right fork and sleep for some random amount of time. While this does not guarantee that starvation will not occur, it reduces the possibility of starvation. Now such a solution is implemented in a protocol like Ethernet.

(Refer Slide Time: 10:57)

Solution using Mutex

- Protect critical sections with a mutex
- Prevents deadlock
- But has performance issues
 - Only one philosopher can eat at a time

```
#define N 5

void philosopher(int i){
while(TRUE){
think(); // for some_time
lock(mutex);
take_fork(R);
take_fork(L);
eat();
put_fork(L);
put_fork(R);
unlock(mutex);
}
}
```

93

Let us look at a third attempt to solve this particular problem. So, this particular solution uses a mutex. Essentially before taking the right or the left fork, the philosopher needs to lock a mutex. And the mutex is unlocked only after eating and the forks are put back on

to the table. So, there is a lock mutex over here before picking the forks; and an unlock mutex after the forks are put down onto the table. So, this solution essentially ensures that starvation will not occur, it prevents that locks.

However, the problem here is that because we are using a mutex, so at most one philosopher can enter into this critical section. In other words, at most one philosopher could eat at any particular instance. So, while this solution works it is not the most efficient solution. So, we would want something which does much better than this.

(Refer Slide Time: 12:20)

Solution with Semaphores

Uses N semaphores ($s[1], s[2], \dots, s[N]$) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
while(TRUE){
think();
take_forks(i);
eat();
put_forks(i);
}
}

void take_forks(int i){
lock(mutex);
state[i] = HUNGRY;
test(i);
unlock(mutex);
down(s[i]);
}

void put_forks(int i){
lock(mutex);
state[i] = THINKING;
test(LEFT);
test(RIGHT);
unlock(mutex);
}

void test(int i){
if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
state[i] = EATING;
up(s[i]);
}
}
```

94

Let us look at our fourth attempt, and this one using semaphores. Let us say that we have N semaphores; so the semaphores s_1 to s_n , and we have one semaphore per philosopher. For all these semaphores are initialized to 0; in addition the philosopher can be in one of 3 states – hungry, eating or thinking. So, over a period of time, each philosopher will move to one of these states. For instance when the philosopher is thinking, the state will be thinking; then the philosopher becomes hungry, so it goes to the hungry state then eating, and then back to thinking, and this process continuous till (Refer Time: 13:12).

So, the general solution that we will be seeing here is that a philosopher can only move to the eating state, if neither neighbor is eating that is a philosopher can eat only if its left neighbor as well as its right neighbor is not eating. So, in order to implement this particular solution, we have four functions. So, first is the philosopher, which is the

infinite loop and corresponds to the philosopher i. so the philosopher will think then it will take forks then eat for some time and put down the forks and this repeats continuously.

Now, in the take forks a function, first we set that the philosopher is in a hungry state; the states of the philosopher is set to hungry then the function called test is invoked. So, what test will do is that is going to check whether the state of the philosopher is hungry and as well as the state of the philosopher to the left as well as to the right is not in eating state. If this indeed is true then the philosopher can eat. And at the end, after eating, the forks are put down and the state of the philosopher goes to thinking.

(Refer Slide Time: 15:06)

Solution to Dining Philosophers

```
void philosopher(int i){
  while(TRUE){
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}
```

```
void take_forks(int i){
  lock(mutex);
  state[i] = HUNGRY;
  test(i);
  unlock(mutex);
  down(s[i]);
}
```

```
void put_forks(int i){
  lock(mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  unlock(mutex);
}
```

```
void test(int i){
  if (state[i] = HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING){
    state[i] = EATING;
    up(s[i]);
  }
}
```

$s[i]$ is 1, so down will not block.
The value of $s[i]$ decrements by 1.

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

96

Let us look at these particular functions more in detail. Let us say that these are the five philosophers P 1, P 2, P 3, P 4, and P 5. And let us say initially that all of them are in the thinking state so this is represented by the T right here. And as we know the initial value for the semaphores are all 0. Let us say all the philosophers are in the thinking state. And then philosopher 3 goes from the thinking state and it goes to the take fork; and as a result the take fork function for philosopher 3 gets invoked, and the state of the philosopher then changes from thinking to hungry.

Then the function test is invoked, and this condition is checked; essentially the state of philosopher 3 is hungry, so this condition is true then the state of the left philosopher is not eating, because P 2 in the thinking state as well as the right philosopher is not eating,

because P 4 also is in the thinking state. So, as a result, this condition goes to true and the state for philosopher three is set to eating, then the corresponding semaphore is incremented from 0 to 1. So the test function returns and lets the mutex which gets unlocked, and then there is the down.

So, down as we know would check the value of s i, and if this value is less than or equal to 0 it is going to block or loop infinitely. And if the value is greater than 0, which is the case over here, then it just decrements the value of s i, so thus here s i was the value had the value of 1, so the down will not block, but rather it is just going to decrement the value of s i to 0. So, s i or the corresponding semaphore has a value of 0. And then the philosopher 3 can go to the eating state and consume his food.

(Refer Slide Time: 17:35)

Solution to Dining Philosophers

```
void philosopher(int i){
  while(TRUE){
    think();
    take_forks(i);
    eat();
    put_forks();
  }
}
```

```
void take_forks(int i){
  lock(mutex);
  state[i] = HUNGRY;
  test(i);
  unlock(mutex);
  down(s[i]);
}
```

```
void put_forks(int i){
  lock(mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  unlock(mutex);
}
```

```
void test(int i){
  if (state[i] = HUNGRY &&
  state[LEFT] != EATING &&
  state[RIGHT] != EATING){
    state[i] = EATING;
    up(s[i]);
  }
}
```

	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	1	0

blocked

99

Now, let us see another situation, where philosopher 4 moves from the thinking state to the take forks state. Now take forks get invoked for philosopher 4 here, and there is a lock mutex the state is set to hungry for philosopher 4 and there is test i which is invoked. So, in the test i, we see that the first condition is met for philosopher 4, because P 4 is indeed in a hungry state; however, the state left is in the eating state right. Therefore, this entire if condition evaluates to false, and therefore, execution does not enter this if loop, rather it just keeps the if part of it. Now we go back to unlock and down.

So what we see now is that the semaphore value corresponding to P 4 as a value of 0. So, when down is invoked as we know it would lead to the process getting blocked. So, thus philosopher P 4 will get blocked, so this P 4 will continue to be blocked as long as P 3 is eating. Then after a while P 3 decides to put down the forks, and sets its state back to the thinking state, then it would invoke test with the left philosopher this is with respect to P 2, which in our case it is not very interesting so we will not look at that. But what is interesting is the test right and the right here corresponds to P 4 so this will invoke here.

And what we see is the state of P 4 so remember that test of right so this is invoked with i having the value of 4. Thus we see that the state of 4 is hungry, because P 4 is hungry the left and the right is not eating, therefore, this condition will be set to true and execution comes into this if. Consequently, the value of state for the philosopher 4 would be set to eating and the semaphore value is set to 1 for P 4.

Now setting the value of 1 for the semaphore will cause the wakeup to occur. So P 4 which was blocked on the semaphore would wake up. And as a result of this the value of the semaphore gets decremented to 0. Thus the P 4 philosopher would wake up and start to eat.

Thus, we see that the semaphores had efficiently ensured that the different philosophers could share the five forks, which they have in common. And it will ensure that every philosopher would get to eat eventually. So, there are other variance and solutions for the dining philosophers' problem which are very interesting to read, but we will not go into that for this particular course.

Thank you.