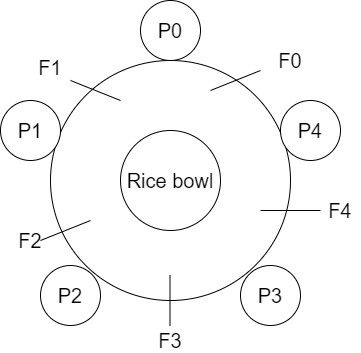
Dining Philosopher Problem | Operating System – M03 P08

This is a multipart blog article series, and in this series I am going to explain you the concepts of operating system. This article series is divided into multiple modules and this is the third module which consists of 10 articles.

In this article we will see about dining philosopher problem, various cases and there solutions in detail.

**Dining philosopher problem**

* To understand this problem let’s take an example.
* There is a dining table on which 5 philosophers are stetted and in between every philosopher there is a “fork” to eat the rice from rice bowl placed in center of the table.
* Every philosopher has two state “think” and “eat”.



Void philosopher(void)

{

While(true)

{

Thinking();

Take\_fork(i);

Take\_fork((i+1)%N);

EAT();

Put\_fork(i);

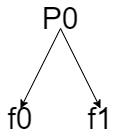
Put\_fork((i+1)%N);

}

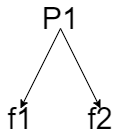
}

*Case 1:-* Po comes

* This will first complete the thinking() function, and then it will pick the left fork, which is in this case will be f0 (because i=0). And then it will take right fork which is f1 (because i=1).
* Now after taking both fork it will start eating and eat() function will execute.
* When eating is done (means eat() is executed completely) it will put left fork first and then the right fork.



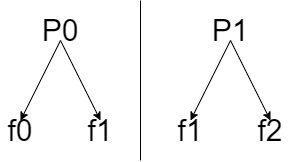
* Now P1 will come and do the same thing as done by P0.



* This is the normal case in it every philosopher comes one by one and this program is fine.

*Case 2: -*

* Suppose P0 comes first and take the left fork (f0), but before he could take the right fork (f1), P1 came and take the left fork (f1), now P0 is waiting for the right fork (f1).



* P0 can get f1 fork only if P1 get f2 (right fork) and then execute eat() and put f1 fork and f1 become free.
* So, if more than one philosopher came at same time then it will cause race condition.
* Now to remove the race condition we use the concept of semaphore (binary semaphore)
* We will use array of semaphore, five semaphore as we have 5 philosophers and fork S0, S1, S2, S3, S4
* All semaphore initialize with 1 value.

Void philosopher(void)

{

While(true)

{

Thingikng();

Wait (take fork (Si));

Wait (take fork(Sci + 1) % N)

Eat();

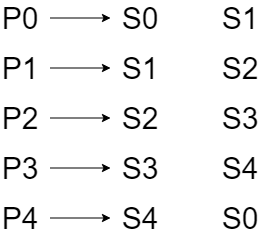
Signal(put fork(i))

Signal(out fork(i+1)%N)

}

}

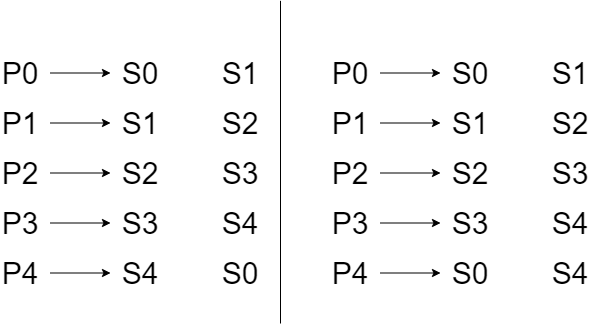
* Now when P0 come and if it want to take two fork then it requires two values of semaphore (S0, S1)
* Then if P1 comes to get executed properly it also require two values of semaphore (S1, S2)



* Now suppose P0 want to eat so he executed the wait() operation and came into critical section and start eating (executing eat()) due to which the value of S0 and S1 become 0 from 1 while other semaphore values at this point is 1.
* Suppose while P0 is in critical section and P1 arrives and it requires S1 and S2 but as we know the value of S1 is 0 so P1 get blocked.
* Now at the same time P2 also arrives and it requires S2 and S3 value to be 1 to get into critical section, as the condition is true so P2 will successfully enter the critical section and start eating (execute eat())
* But as we know that first condition of process synchronization is “mutual exclusion” accordingly to which at a time only one process can enter critical section. But this “Dining philosopher problem” is a special case for that.
* P0 and P2 are independent and do not have dependency because p0 need f0 and f1 while P2 needs f2 and f3, Due to which more than one process can enter critical section (because both process are independent.)

*Case 3: -*

* Suppose value of all semaphore is initialized with 1.
* P0 arrives and it takes the left fork (f0) and get preempt. Then P1 arrives and take left fork (f1) and also get preempt. Similarly P2, P3 and P4 take f2, f3 and f4 fork (left fork) respectively. And all the processes get preempt just before they were going to pick right fork.
* Now every philosopher had taken the left fork but no one have the right fork. Value of all semaphore is 0
* Now every process is blocked and they cannot unblock if they try they get blocked again because value of all semaphore 0. This is the “Dead lock condition”
* The only solution to this problem is that we can change the sequence of fork of P4. Which means P4 will first will take the right fork instead of left fork (f0) and then left fork instead of right fork (f4)



* So, when the P4 comes it cannot take any fork and get blocked due to which P3 can take the right fork (f4) as S4 is 1 already, and so on. By this we can overcome the deadlock state.
* For P4 this will be the sequence of entry section code.

Wait(take fork(S(i+1)%N)

Wait(take fork(Si)

So this was all about dining philosopher problem. Hope you liked it and learned something new from it.

If you have any doubt, question, quires related to this topic or just want to share something with me, then please feel free to contact me.