**Classical IPC Problem**

**Producer Consumer Problem:**

Producer Consumer Problem is a classical synchronization problem with presence of more than one process and limited resources in the system. Here, producer produce some items and insert into memory buffer whereas consumer is consuming the item from that memory buffer. Memory buffer is of fixed size and same for both producer and consumer. Producer continue to produce item and store into memory buffer until buffer is not full. If buffer is full then producer is not allowed to store any data into memory buffer. Consumer can only consume a data if there are some items in memory buffer i.e. if memory buffer is not empty. Memory buffer cannot be accessed by produce and consumer at the same time. Following table shows the code section from producer consumer problem:

|  |  |
| --- | --- |
| **Producer**  int count=0;  void producer(void){  int itemP  While(true){  Produce\_item(item P)  Load Rp,m[count]  INCR Rp  Store m[count], Rp  While(count==n);  Buffer[in] = itemP;  In = (in+1)mod n;  Count = count+1;  }  } | **Consumer**  int count;  void consumer(void){  int itemC;  While(true){  Load Rc,m[count]  DECR Rc  Store m[count], Rc  While(count==0);  itemC = buffer[out];  out = (out+1)mod n;  count = count-1;  }  } |

**Some terms used in producer consumer problem:**

**In:** used in producer which represent the address of next empty buffer

**Out:** used in consumer which represent the address of first filled buffer (address of buffer that have item)

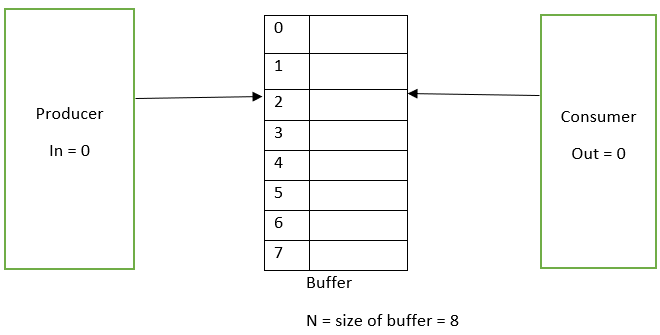
**Count:** it is common for both producer and consumer that counts the number of elements present in buffer. Count is divided into three line of code that shows how work is performed by CPU for incrementing and decrementing the value of count. For producer: value of count is first retrieved and store in CPU’s register (Rp) then the value of register (Rp) is incremented and saved. For consumer: value of count is first retrieved and store in CPU’s register (Rc) then the value of register is decrement and saved.

**Let us consider following two scenarios:**

Suppose: total buffer size (n) = 8 and address: 0-7

**Scenario 1: buffer is empty and no any interruption occur:**

**For Producer:**



At first, count = 0, n=8 (total size of buffer)

Since no any item are stored value of in is also 0. In = 0;

If producer wants to produce an item “x1” then code of producer is executed where condition of while loop i.e. always true is checked and itemP = x1 is tried to insert into buffer. Now, another while loop with condition (count==n) is checked. Here, value of count is 0 and value of n is 8 so 0==8 is false. Therefore, while loop is not evaluated and producer store the item x1 in memory buffer:

Buffer[in]=item x1;

Buffer [0] = x1; => inserted in 0th position of buffer

In = (in +1) mod n => 0+1 mod 8 = 1 (address of next empty buffer)

Count = count +1 have three part

**Part 1:**

Load Rp, m[count] = value of count which is ) is store into Register Rp;

**Part 2:**

Increment Rp = since one item is inserted value of Rp is increment to 1 from 0

**Part 3:**

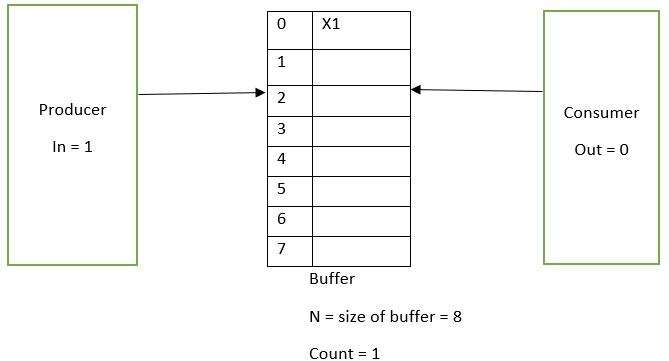
Store m[count], Rp = value of register Rp which is 1 is now store into m[count]

Now for next item:

Count = 1

In = 1

N = 8



**Figure: After producer produce and insert the item**

**For Consumer:**

Count = 1; as count is common for both

Out = 0; //initially value of out is 0

If consumer wants to consume first item x1 then it will enter into consumer() function where while(true) is evaluated which is always true so, another while loop with condition count==0 is evaluated. Value of count is 1 now so 1==0 is false so, it will not enter into while loop and try to consume the item x1,

ItemC = buffer[out]. As value of out is 0

ItemC = buffer[0];

Out = (out+1) mod n = (0+1) mod 8 = 1

Now x1 is removed.

Count = count-1; have three parts

**Part 1:**

Load Rp, m[count] = value of count which is 1 is store into Register Rp;

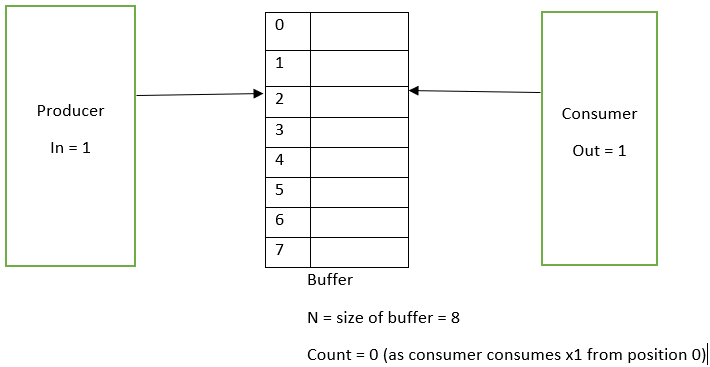
**Part 2:**

Decrement Rp = since one item is took out of buffer value of Rp is decrement to 0 from 1

**Part 3:**

Store m[count], Rp = value of register Rp which is 0 is now store into m[count]

Therefore, value of count=0 for new iteration.



**Figure: after consumer consumes the item from position 0**

**Scenario 2: when interruption occur**

**For Producer:**

Let us consider there are already three item x1, x2 and x3 in buffer position 0, 1 and 2 and suppose that item x4 to be inserted in position 3. So,

Value of in = 3

Count = 3

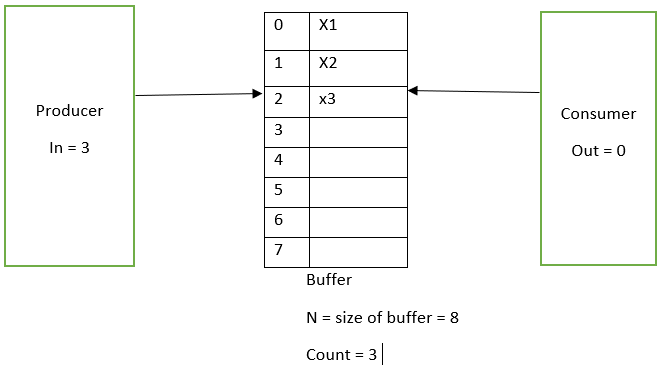


Figure: after three item are inserted

then code of producer is executed where condition of while loop i.e. always true is checked and itemP = x4 is tried to insert into buffer. Now, another while loop with condition (count==n) is checked. Here, value of count is 3 and value of n is 8 so 3==8 is false. Therefore, while loop is not evaluated and producer store the item x4 in memory buffer location 3:

Buffer[in]=item x4;

Buffer [3] = x4; => inserted in 3th index of buffer

In = (in +1) mod n => 3+1 mod 8 = 4 (address of next empty buffer)

Count = count +1 have three part

**Part 1:**

Load Rp, m[count] = value of count which is 3 is store into Register Rp;

**Part 2:**

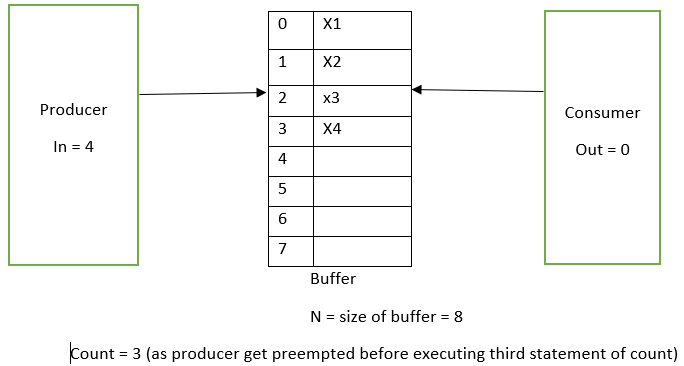
Increment Rp = since one item is inserted value of Rp is increment to 4 from 0

Now, after executing instruction of part 2 suppose the process get preempted due to some interrupt i.e., instruction of part 3 (store m[count], Rp) is not executed. Therefore, value of count is not updated. So,

Count=3

In = 4

Rp of producer= 4 and value of count is not updated so count = 3;



**For Consumer:**

Let us consider that consumer send the request to consume first item (0th position) of memory buffer then:

Out = 0; //initially value of out is 0

Count = 3 // as count is common for both producer and consumer

Consumer will enter into consumer () function where while(true) is evaluated which is always true so, another while loop with condition count==0 is evaluated. Value of count is 3 now so 3==0 is false so, it will not enter into while loop and try to consume the item x1 from 0th position,

ItemC = buffer[out]. As value of out is 0

ItemC = buffer[0];

Out = (out+1) mod n = (0+1) mod 8 = 1

Now x1 is removed from 0th position

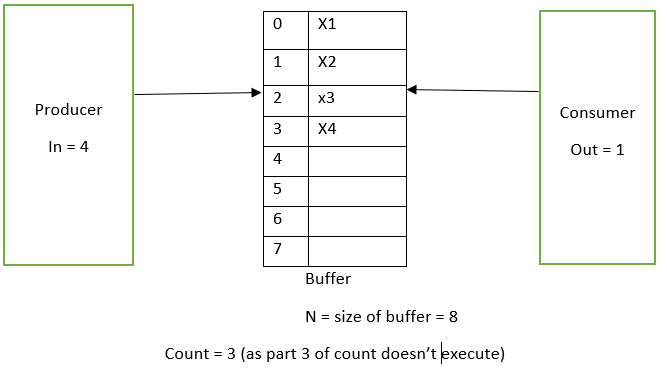
Count = count-1; have three parts

**Part 1:**

Load Rc, m[count] = value of count which is 3 is store into Register Rc;

**Part 2:**

Decrement Rc = since one item is took out of buffer value of Rc is decrement to 2 from 3. After executing part 2 suppose consumer process gets preempted due to some interrupt i.e. instruction of part 3 does not get chance to execute and producer process that gets preempted before sends the request. Then producer which was preempted before will start executing the instruction from where it has left before i.e. part 3 instruction of producer will executed:



**Figure: consumer process after executing 2nd part of count**

**Producer’s remaining Part 3 statement of count:**

M[count], Rp: so the value of count is 4 and value of Rp=4

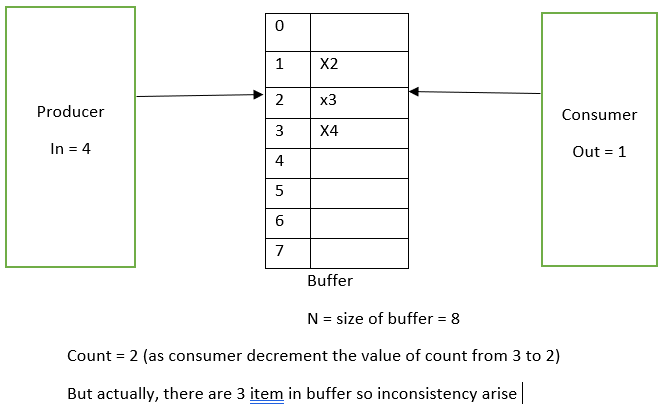
Count = 4;

Now after completing the instruction item x4 is inserted into position 3 and producer process is completed. Consumer process which was preempted now sends request and execute the instruction from position from where it has left i.e. instruction of part 3 of consumer. So,

M[count], Rc: value of rc is 2 so value of count is updated to 2 from 4. As consumer process thinks value of count is 3 because it was activate from the part 2 statement of count (from the state it was left before).

**Count = 2;**

It indicates that there are two items in memory buffer but actually there are 3 items in buffer. So, inconsistency arise.



**Figure: inconsistency in producer-consumer**

**Solution of producer consumer problem using counting semaphore**

Here,

S = semaphore (1: critical section is empty, 0: critical section is busy)

Full: number of filled buffer

Empty: number of empty buffers

Initially:

Full = 0;

Empty = 8; (size of buffer which is 8 in our case)

Semaphore (S) = 0;

|  |  |
| --- | --- |
| **Producer**  Void producer(){  Wait(empty);  Wait(S);  produceItem(item P)  buffer[in] = itemP;  in = in+1 mod n  signal(S)  signal(full) | **Consumer**  Void consumer(){  Wait(empty);  Wait(S);  itemC = buffer[out];  out = out+1 mod n;  signal(S)  signal(empty) |

**For Producer:**

Let us consider there are already three item x1, x2 and x3 in buffer position 0, 1 and 2 and suppose that item x4 to be inserted in position 3. So,

Value of in = 3

Count = 3

Empty: 5 (as there are already 3 item in buffer so 8 – 3 = 5)

Full: 3 (as three item are filled in buffer)

S=1

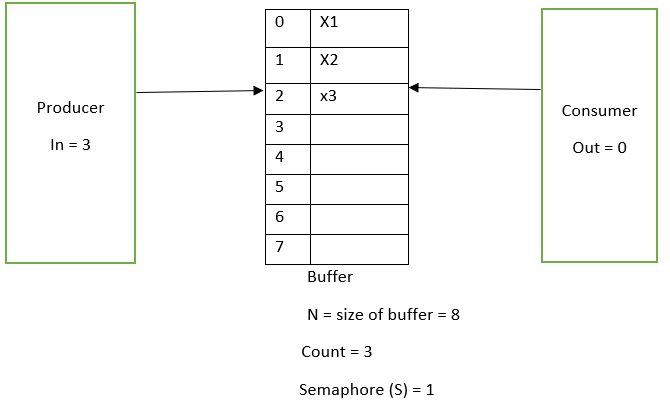


Figure: initial state

As, x4 to be inserted in position 3 so, producer code will be executed:

Wait(empty) will call first which will decrement the value of empty by 1 i.e. value of empty will be 4 from 5.

Wait (s) will call in second which will decrement the value of S by 1. i.e. value of S will be 0 from 1.

buffer[in] = itemP; as value of in is 3 so

buffer[3] = x4;

in = in+1 mod n

in = 3+1 mod 8 = 4;

Count = count +1 have three part

**Part 1:**

Load Rp, m[count] = value of count which is 3 is store into Register Rp;

**Part 2:**

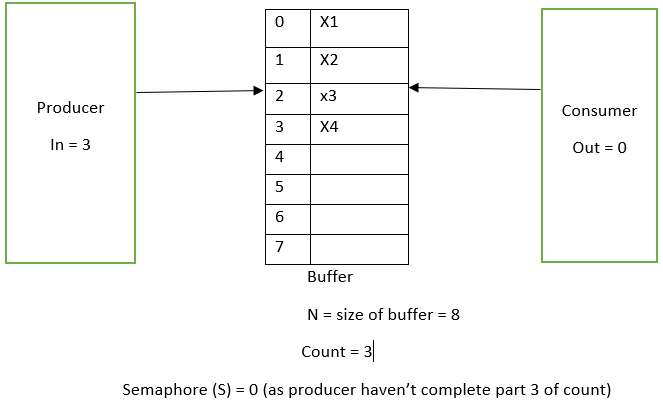
Increment Rp = since one item is inserted value of Rp is increment to 4 from 3

Now, after executing instruction of part 2 suppose the process get preempted due to some interrupt i.e., instruction of part 3 (store m[count], Rp) is not executed. Therefore, value of count is not updated. So,

Count=3

In = 4

Rp of producer= 4



**Figure: after producer executes part 2 of count and gets preempted**

For Consumer:

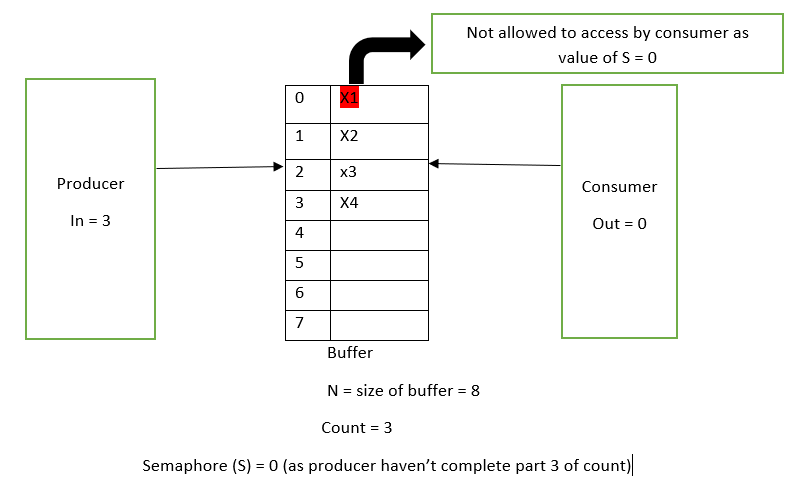
Let us consider that consumer send the request to consume first item (0th position) of memory buffer then:

Out = 0; //initially value of out is 0

Count = 3 // as count is common for both producer and consumer

S = 0; //as producer have decremented the value of S and haven’t come out of critical section.

Since, value of S = 0, it means that critical section is busy so consumer process is not allowed to decrement the value of S. So, the consumer process will not allowed to enter into critical section until producer process execute the last statement and calls signal (s) .



**Figure: consumer trying to access item x1**

Therefore, when producer process is executing then consumer process is not allowed to enter into critical section, hence, mutual exclusion is hold.

**Dining philosopher problem:**

Dining philosopher problem is a classical inter process communication and synchronization problem for multiple process. Here, scenario is that there are N philosophers sitting around a circular dining table eating rice and discussing philosophy. The problem is that each philosopher needs 2 fork to eat and there are N fork, one between each 2 philosophers. In our case let us suppose there are 5 philosopher sitting on dining table and there are 5 fork, one between each 2 philosopher. The job is to think about philosophy and to eat. The philosopher can only eat if both right and left chopstick of philosopher is available i.e. one can eat if there are two fork available. If one of the fork (left or right) or both (left and right) fork is not available then he puts down the available fork and start thinking again.

Void philosopher (void){

While(true){

Thinking();

takeFork(i); //left fork

takeFork((i+1)%N); //right fork

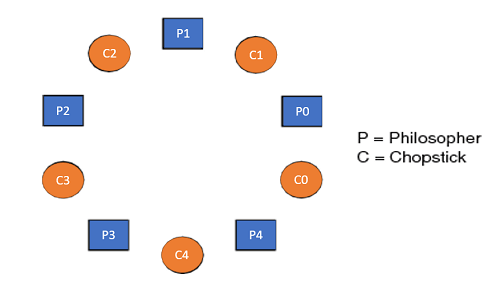
eat();

putFork(i); //put leftFork

putFork((i+1)%N); //put right fork

}

}



Let us suppose, there are 5 philosopher be consider as process and denoted by P0,P1,P2,P3,P4 and five fork be denoted by c0,c1,c2,c3 and c4;

**Case 1: Normal flow**

Suppose, P0 process request to eat then it will try to lift left fork i.e. execute take\_fork(i). as p0 process has request value of ‘I’ is 0 then:

TakeFork(i) = takeFork(0) = C0

takeFork((i+1)%n)) = takeFork((0+1)modN) = takeFork(1) = c1;

So, P0 process takes fork C0 and C1 and start eating Eat(); After eating it will put down the forks as:

putFork (i) = putFork(0) = C0;

putFork((i+1)%n)) = takeFork((0+1)%n)) = C1

Now, P1 request to eat then it will try to lift left fork and then right for as:

Takefork(i) = takeFork(1) = C1;

Takefork((i+1)modn) = takefork((1+1)mod n) = 2 = C2

Now, P2 will start eating using eat() and after eating it will put down the left and then right fork as:

putFork(i) = putFork(1) = C1;

putFork(i+1) mod n) = putFork(2 mod n) = 2 = C2;

This is a normal case in which philosopher (process) request and consume in serial manner. Now, lets talk about another case .

**Case 2: interruption occur:**

Void philosopher (void){

While(true){

Thinking();

takeFork(i); //left fork

takeFork((i+1)%N); //right fork

eat();

putFork(i); //put leftFork

putFork((i+1)%N); //put right fork

}}

Suppose, P0 process request to eat then it will try to lift left fork i.e. execute take\_fork(i). as p0 process has request value of ‘I’ is 0 then:

TakeFork(i) = takeFork(0) = C0;

While going to take next fork (right fork C1), suppose P1 requested to eat then P1 will execute same code and try to take left fork as takeFork(i) = takeFork(1) = 1 = C1. Here, P0 is waiting to take his right fork C1. P0 will only gets his right fork C1 when process P1 will get his both left and right fork i.1 C1 and C2, starts eating (eat()) and left its both left and right fork C1 and C2. It means that if more than one philosopher (process) will request to eat at same time then there is a chance of **race condition.**

**Solution of Dining philosopher problem using Semaphore.**

Here semaphore is used to represent chopstick and wait() or down() operation is used for picking fork and signal() operation is used for putting down the fork. As semaphore represents total number of fork then in our case number of fork is 5 so value of semaphore is also 5.

S[i] = S[5] = S0, S1,S2,S3,S4 and all are initialize with value 1. [S0=1. S1=1,S2=1, S3 =1, S4=1]

Now the code section is:

Void philosopher (void){

While(true){

Thinking();

Wait(takeFork(Si); //left fork

Wait(takeFork((Si+1)%N)); //right fork

eat();

Signal(putFork(i)); //put leftFork

Signal(putFork((i+1)%N)); //put right fork

}}

**Scenario for fork required;**

Now, suppose that P0 request to eat then it will required two fork (left and right) to eat so it will required two semaphore (S0 and S1) i.e.

Wait(takeFork(Si) = Wait(takeFork(S0) = S0;

Wait(takeFork((Si+1)%N)) = Wait(takeFork((0+1)%5)) = S1

suppose that P1 request to eat then it will required two fork (left and right) to eat so it will required two semaphore (S1 and S2) i.e.

Wait(takeFork(Si) = Wait(takeFork(S1) = S1;

Wait(takeFork((Si+1)%N)) = Wait(takeFork((1+1)%5)) = S2 and so on for other process

|  |  |
| --- | --- |
| Process | Fork Required |
| P0 | S0, S1 |
| P1 | S1, S2 |
| P2 | S2, S3 |
| P3 | S3, S4 |
| P4 | S4, S0 |

**Solution:**

If P0 request to eat then it will require two semaphore s0 and s1. Value of semaphore are 1 i.e. S0 = 1 and S1 = 1.

To eat it will call following function:

Wait(takeFork(Si)) = wait(takeFork(S0) )=wait(S0) so value of S0 will be changed to 0 from 1.

Wait(takeFork((Si+1)%N)) = Wait(takeFork((0+1)%5)) = S1 so value of S1 will be changed to 0 from 1.

Now: S0 = 0 , S1=0. It means that both semaphore S0 and S1 is busy.

**Suppose, P1 request to eat then it will execute:**

Wait(takeFork(Si)) = wait(takeFork(S1) )=wait(S1)

Now the value of S1 is 0 (P0 is alredy using) so it will not allow to take the fork and will have to wait.

Suppose, P2 request to eat then it will require S2 and S3. The value of S2 and S3 is 1 and execute:

Wait(takeFork(Si)) = wait(takeFork(S2) )=wait(S2);

Initially the value of S2 is 1 and now it will be change to 0.

Wait(takeFork((Si+1)%N)) = Wait(takeFork((2+1)%5)) = S3

Now value of S3 is changed to 0 and P2 can call eat () function. P0 is already in eat() which means that P0 has not came out of critical section P2 accessed the critical section. This means mutual exclusion does not hold. But actually, P0 and P2 are independent process to each other as P0 need S0 and S1 to eat and P2 needs S2 and S3. There is no requirement of common spoon between P0 and P1. So, P0 and P2 can access critical section at same time. This is a special case in dining philosopher problem.

**Note:**

* Two or more Dependent process (that exchange data between each other) cannot access critical section at same time.
* Independent process can access critical section.

Following figure shows the condition for dependent and independent process on dining philosopher problem:

P0 (S0,S1)=>P1(S1,S2) : dependent process as both need S1

P0(S0,S1)=>P2(S2,S3): independent process as both doesn’t need common spoon

If all the process simultaneously made request (P0=>P1=>P2=>P3=>P4) then no one will be able to eat. This condition is called deadlock situation:

|  |  |
| --- | --- |
| Process | Fork Required |
| P0 | S0, S1 |
| P1 | S1, S2 |
| P2 | S2, S3 |
| P3 | S3, S4 |
| P4 | S4, S0 |

When P0 takes S0 and before taking S1 if it gets preempted then S1 will be taken by P1 and if P1 gets preempted before it gets S2 then S2 will take by P2. Following this process, all the process have to wait for indefinite time. This is deadlock condition.

**Solving Deadlock condition:**

To solve deadlock condition, change the orientation of spoon of any one process:

|  |  |  |
| --- | --- | --- |
| Process | Previous Fork Required | New Fork Orientation |
| P0 | S0, S1 | S0, S1 |
| P1 | S1, S2 | S1, S2 |
| P2 | S2, S3 | S2, S3 |
| P3 | S3, S4 | S3, S4 |
| P4 | S4, S0 (change orientation) | S0, S4 |

If P0 request to eat then it will require two semaphore s0 and s1. Value of semaphore are 1 i.e. S0 = 1 and S1 = 1.

To eat it will call following function:

Wait(takeFork(Si)) = wait(takeFork(S0) )=wait(S0) so value of S0 will be changed to 0 from 1.

Before executing second statement of wait ( taking right spoon) Po gets preempted.

Now: S0 = 0 , S1=1. It means that S0 is busy and S1 is empty

**Suppose, P1 request to eat then it will execute:**

Wait(takeFork(Si)) = wait(takeFork(S1) )=wait(S1)

Now it will change the value of S1 to 0 from 1 and moves to execute second statement of wait. Before executing second statement of wait (taking right spoon) P1 gets preempted then

S1= 0, S2 = 1 means that S1 is busy and S2 is empty.

**Following this sequence, when P4 arrives it will have following condition:**

S1 =0, S1 = 0, S2= 0, S3 = 0, S4=1

As we have changed the orientation of P4, it will first try to use right spoon that means try to execute 2nd part of wait:

Wait(takeFork((i+1)mod n) = takeFork(4+1)mod 5) = takeFork(S0);

Here, value of S0 is already 0 so process P4 will gets blocked (it will not get chance to execute both part of wait). Now the value of semaphore are: S1 =0, S1 = 0, S2= 0, S3 = 0, S4=1

Process P3 required S3 (left) and S4 (right) spoon to eat. Here, S3 is already occupied by P3 and value of S4 is 1. So, it can call 2nd statement wait(takeFork(i+1)mod n)= (3+1)mod 5 = S4 and eat. Now, value of S0 to S4 all is 0. After completing its eat(), P3 will call signal() function to release its left (S3) and Right (S4) spoon i.e. S3=1, S4=1. When S3 will be 1, P2 can call eat(). When P2 ends, S2 spoon will be free and P1 can call eat and so on. In this way deadlock can be removed.