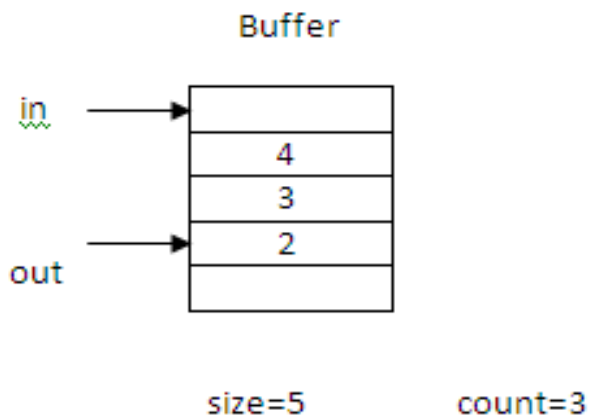
PROCESS SYNCHRONIZATION

Producer – Consumer Problem

- There is a buffer which contains some number of slots.
- One item can be stored into each slot of the buffer.
- *In* and *out* are pointers to the buffer.
- *Count* is a variable which indicates the number of items currently existing in the buffer.
- *Producer* is the process which enters items into the buffer.
- *In* points to the next free slot of buffer into which the producer inserts the next item.
- *In* pointer is moved to the next position after inserting an item into the buffer.
- *Consumer* is the process which takes items from the buffer.
- *Out* points to the slot from which the consumer removes the item.
- *Out* pointer is moved to the next position after deleting an item from the buffer.



Producer Process

The code of producer process is:

```
while (true)
{
        P1:     while (count == sizeofbuffer)
        P2:             ;
        P3:     buffer[in] = item;
        P4:     in = (in + 1) % sizeofbuffer;
        P5:     a = count;
        P6:     a = a + 1;
        P7:     count = a;
}
```

Consumer Process

The code of consumer process is:

```
while(true)
{
        C1:     while (count == 0)
        C2:             ;
        C3:     item = buffer[out];
        C4:     out = (out+1) % sizeofbuffer;
        C5:     b = count;
        C6:     b = b - 1;
        C7:     count = b;
}
```

Serial Execution

With serial execution, processes are executed one after another.

CPU is switched to another process only after completion of the currently running process.

Concurrent Execution (or) Parallel Execution

With concurrent execution, CPU can be transferred to another process during execution of the current process.

When the processes are not accessing common variables:

We get correct output when the processes are executed either serially or concurrently.

When the processes are accessing and modifying common variables:

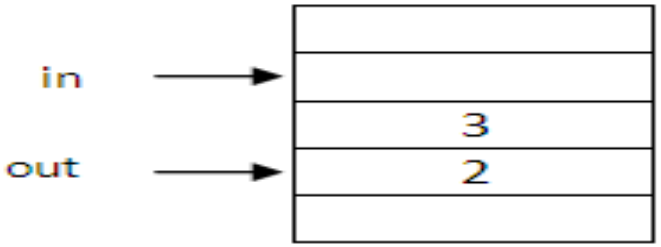
We get correct output in serial execution of the processes.

We may get wrong output in some sequences of concurrent execution of the processes.

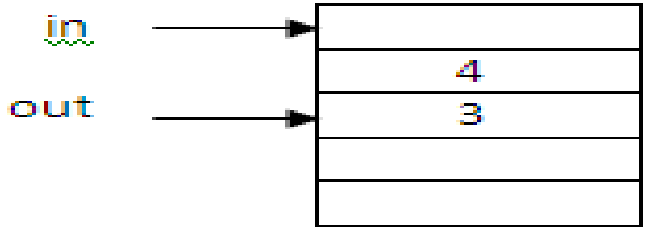
One example concurrent execution of instructions of producer & consumer processes that leads to wrong result is **P1, P2, P3, P4, P5, P6, C1, C2, C3, C4, C5, C6, P7, C7**.

```
while (true)
{
P1:     while (count == sizeofbuffer)
P2:             ;
P3:     buffer[in] = item;
P4:     in = (in + 1) % sizeofbuffer;
P5:     a = count;
P6:     a = a + 1;
P7:     count = a;
}
```

```
while(true)
{
C1:     while (count == 0)
C2:             ;
C3:     item = buffer[out];
C4:     out = (out+1) % sizeofbuffer;
C5:     b = count;
C6:     b = b - 1;
C7:     count = b;
}
```



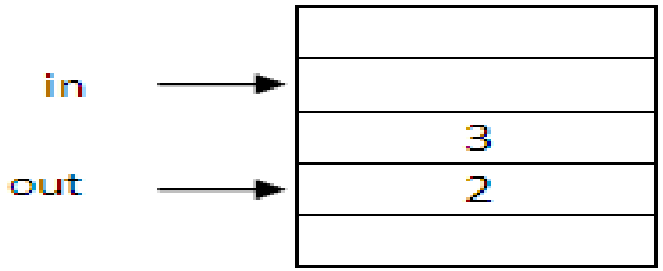
count=2



count=1

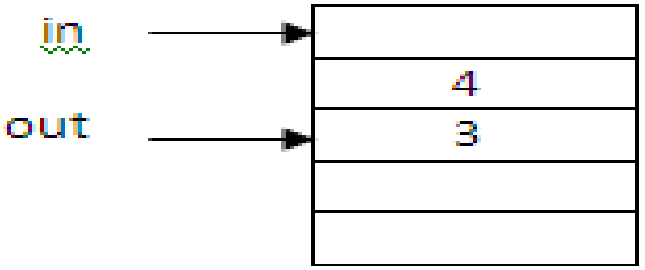
P1, P2, P3, P4, P5, P6, C1, C2, C3, C4, C5, C6, P7, C7

If the position of buffer is



count=2

When the statements of producer and consumer are executed in the order **P1, P2, P3, P4, P5, P6, C1, C2, C3, C4, C5, C6, P7, C7** then the position of buffer is

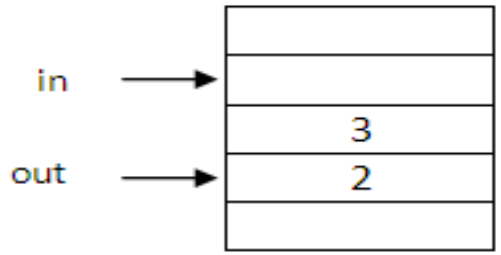


count=1

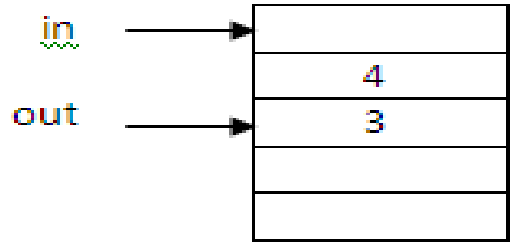
We are getting wrong output i.e. count=1.

```
while (true)
{
P1:     while (count == sizeofbuffer)
P2:             ;
P3:     buffer[in] = item;
P4:     in = (in + 1) % sizeofbuffer;
P5:     a = count;
P6:     a = a + 1;
P7:     count = a;
}
```

```
while(true)
{
C1:     while (count == 0)
C2:             ;
C3:     item = buffer[out];
C4:     out = (out+1) % sizeofbuffer;
C5:     b = count;
C6:     b = b - 1;
C7:     count = b;
}
```



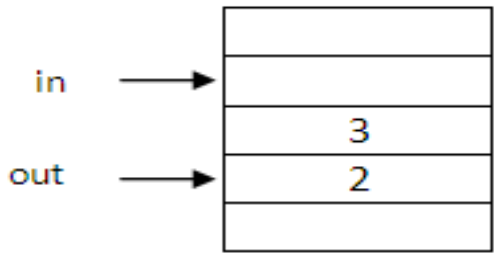
count=2



count=3

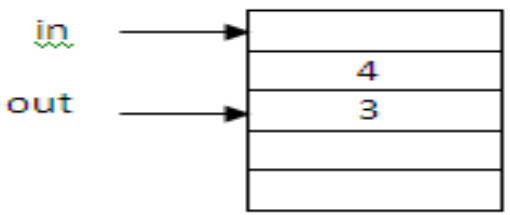
P1, P2, P3, P4, P5, P6, C1, C2, C3, C4, C5, C6, C7, P7

If the position of buffer is



count=2

When the statements of producer and consumer are executed in the order **P1, P2, P3, P4, P5, P6, C1, C2, C3, C4, C5, C6, C7, P7** then the position of buffer is



count=3

Here also, we are getting wrong output i.e. count=3.

Race condition

If we get different outputs for different orders of concurrent execution of statements of processes, then that situation is called as “Race condition”.

The reason for getting wrong results in concurrent execution of processes is accessing and updating some common variables (shared data) in all processes.

Ex: What would be the different values that A, B can possibly take after the execution of the processes P1, P2 concurrently?

Initial values of A and B are 6 and 4 respectively.

Process P1:	Process P2:
I1: A=A-B; I2: B=B+A;	I11: A=A+1; I12: B=B-1;

Concurrent execution sequence3: I1, I11, I2, I12

A=6-4=2

A=2+1=3

B=4+3=7

B=7-1=6

A=3, B=6

Concurrent execution sequence4: I11, I12, I1, I2

A=6+1=7

B=4-1=3

A=7-3=4

B=3+4=7

A=4, B=7

Concurrent execution sequence5: I11, I1, I2, I12

A=6+1=7

A=7-4=3

B=4+3=7

B=7-1=6

A=3, B=6

Concurrent execution sequence6: I11, I1, I12, I2

A=6+1=7

A=7-4=3

B=4-1=3

B=3+3=6

A=3, B=6

The different possible values that A and B can take are

A=3, B=5

A=3, B=6

A=4, B=7

Critical Section

Critical section is part of the process where the process is accessing and updating the common variables (shared data).

In producer process, the critical section is

```
a = count
a = a + 1
count = a
```

In consumer process, the critical section is

```
b = count
b = b - 1
count = b
```

Critical Section Problem

Allowing only one process to execute its critical section at a time.

(or)

If a process is executing its critical section, then at the same time no other process is allowed to execute its critical section.

This restriction on the execution of processes leads to generation of correct output in the concurrent execution of processes.

Solutions for Critical Section Problem (or) Solutions for Implementing Mutual Exclusion

Following are different solutions to the critical section problem

1. Peterson's Solution or Software solution
2. Hardware Solution
3. Semaphore Solution
4. Monitor Solution

Any solution to the critical section problem must satisfy the following three conditions

1) Mutual exclusion
2) Progress
3) Bounded waiting

Mutual Exclusion

At a time, only one process should execute in its critical section.

Progress

All processes should not be in deadlock state. At any time, at least one process should be in the running state.

Bounded waiting

All processes should be given equal chance to execute their critical sections.

Peterson's Solution

It is applicable to two processes only.

Two variables are used

- int turn;
- boolean flag[2];

turn variable indicates whose turn it is to execute its critical section.

turn=1 - indicates that the 1st process (producer process) can execute its critical section.

turn=2 - indicates that the 2nd process (consumer process) can execute its critical section.

flag indicates whether a process is ready to execute its critical section or not.

flag[1]=true - indicates that the 1st process (producer process) is ready to execute its critical section.

flag[2]=true - indicates that the 2nd process (consumer process) is ready to execute its critical section.

Initially, flag[1] and flag[2] are set to false.

The following procedure is used to solve the critical section problem.

When the 1st process (producer process) wants to execute its critical section then the 1st process (producer process) sets flag[1] to true.

If the 2nd process (consumer process) is also ready to execute its critical section, then the 1st process (producer process) allows the 2nd process (consumer process) to execute its critical section by setting the turn value to 2.

The 1st process (producer process) waits until the 2nd process (consumer process) completes its critical section.

Then the 1st process (producer process) executes its critical section.

The 1st process (producer process) sets flag[1] to false after completing its critical section.

Same procedure is followed by the 2nd process (consumer process) when it wants to execute its critical section.

Peterson's solution to the critical section problem of producer-consumer is as follows:

Producer Process

```
while (true)
{
        P1:     while (count == sizeofbuffer)
        P2:             ;
        P3:     buffer[in] = item;
        P4:     in = (in + 1) % sizeofbuffer;

                flag[1]=true;
                turn=2;
                while(flag[2]==true && turn==2)
                        ;

        P5:     a = count;
        P6:     a = a + 1;
        P7:     count = a;

                flag[1]=false;
}
```

Consumer Process

```
while(true)
{
        C1:     while (count == 0)
        C2:             ;
        C3:     item = buffer[out];
        C4:     out = (out+1) % sizeofbuffer;

                flag[2]=true;
                turn=1;
                while(flag[1]==true&&turn==1)
                        ;

        C5:     b = count;
        C6:     b = b - 1;
        C7:     count = b;

                flag[2]=false;
}
```

Hardware Solution

A variable named ‘lock’ is used.

boolean lock;

When a process wants to execute its critical section then the process checks the value of lock. If the value of lock is true, then the process is not allowed to execute its critical section. If the value of lock is false, then the process is allowed to execute its critical section. Before executing the critical section, the process must set lock value to true.

After executing the critical section, the process must set lock value to false.

The structure of process is

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

TestandSet()

'TestandSet' is a hardware instruction which performs operations on the 'lock' variable to solve the critical section problem.

This function takes 'lock' variable as input and returns a boolean value.

The definition of TestandSet() function is

```
boolean TestandSet(boolean lock)
{
        boolean temp=lock;
        lock=true;
        return temp;
}
```

TestandSet() is an atomic instruction.

Initial value of ‘lock’ is false.

The solution to critical section problem of producer-consumer is

Producer Process

```
while(true)
{
        P1:     while(count==sizeofbuffer)
        P2:             ;
        P3:     buffer[in]=item;
        P4:     in=(in+1)%sizeofbuffer;

Entry Section   while(TestandSet(lock))
                        ;

        P5:     a=count;
        P6:     a=a+1;          Critical Section
        P7:     count=a;

Exit Section    lock=false;

}
```

Consumer Process

```
while(true)
{
        C1:     while(count==0)
        C2:             ;
        C3:     item=buffer[out];
        C4:     out=(out+1)%sizeofbuffer

Entry Section   while(TestandSet(lock))
                        ;

        C5:     b=count;
        C6:     b=b-1;          Critical Section
        C7:     count=b;

Exit Section    lock=false;

}
```

Swap()

'Swap' is another hardware instruction used to solve the critical section problem. The definition of Swap() instruction is

```
void swap(boolean lock, boolean key)
{
        boolean temp=lock;
        lock=key;
        key=temp;
}
```

Lock is global variable.

key is local variable.

Initial value of lock is false.

The solution to critical section problem of producer-consumer is

Producer Process

```
while(true)
{
        P1:     while(count==sizeofbuffer)
        P2:             ;
        P3:     buffer[in]=item;
        P4:     in=(in+1)%sizeofbuffer;

Entry Section   key=true;
                while(key==true)
                        swap(lock,key);

        P5:     a=count;
        P6:     a=a+1;          Critical Section
        P7:     count=a;

Exit Section    lock=false;

}
```

Consumer Process

```
while(true)
{
        C1:     while(count==0)
        C2:             ;
        C3:     item=buffer[out];
        C4:     out=(out+1)%sizeofbuffer;

Entry Section   key=true;
                while(key==true)
                        swap(lock,key);

        C5:     b=count;
        C6:     b=b-1;          Critical Section
        C7:     count=b;

Exit Section    lock=false;

}
```

Semaphore Solution

Semaphore is an integer variable. There are two types of semaphore

1) Binary semaphore
2) Counting semaphore

Binary semaphore is used to solve the critical section problem.

Binary semaphore is also called as ‘mutex’ variable because binary semaphore is used to implement mutual exclusion.

Initial value of binary semaphore is 1.

Counting semaphore is used to provide synchronous access to a resource by number of processes.

Initial value of counting semaphore is equal to the number of instances of the resource.

A queue is associated with each semaphore variable.

Two operations are defined on a semaphore variable

1) wait()
2) signal()

The definition of wait() operation is

```
wait(s)
{
        s=s-1;
        if(s<0)
        {
                block the process;
                Insert the process in the queue associated with the semaphore
        variable s;
        }
}
```

The definition of signal() operation is

```
signal(s)
{
        s=s+1;
        if(s<=0)
        {
                remove a process form the queue associated with the semaphore
        variable s;
                restart the removed process;
        }
}
```

To solve the critical section problem, a process should call wait() operation on the semaphore variable before executing its critical section and signal() operation after executing its critical section.

```
wait(S)
 Critical Section
signal(S)
```

Solution to the critical section problem of producer-consumer is

Producer Process

```
while (true)
{
        P1:     while (count == sizeofbuffer)
        P2:             ;
        P3:     buffer[in] = item;
        P4:     in = (in + 1) % sizeofbuffer;

                wait(s);

        P5:     a = count;
        P6:     a = a + 1;
        P7:     count = a;

                signal(s);
}
```

Consumer Process

```
while(true)
{
        C1:     while (count == 0)
        C2:             ;
        C3:     item = buffer[out];
        C4:     out = (out+1) % sizeofbuffer;

                wait(s);

        C5:     b = count;
        C6:     b = b - 1;
        C7:     count = b;

                signal(s);
}
```

Problems with Semaphore

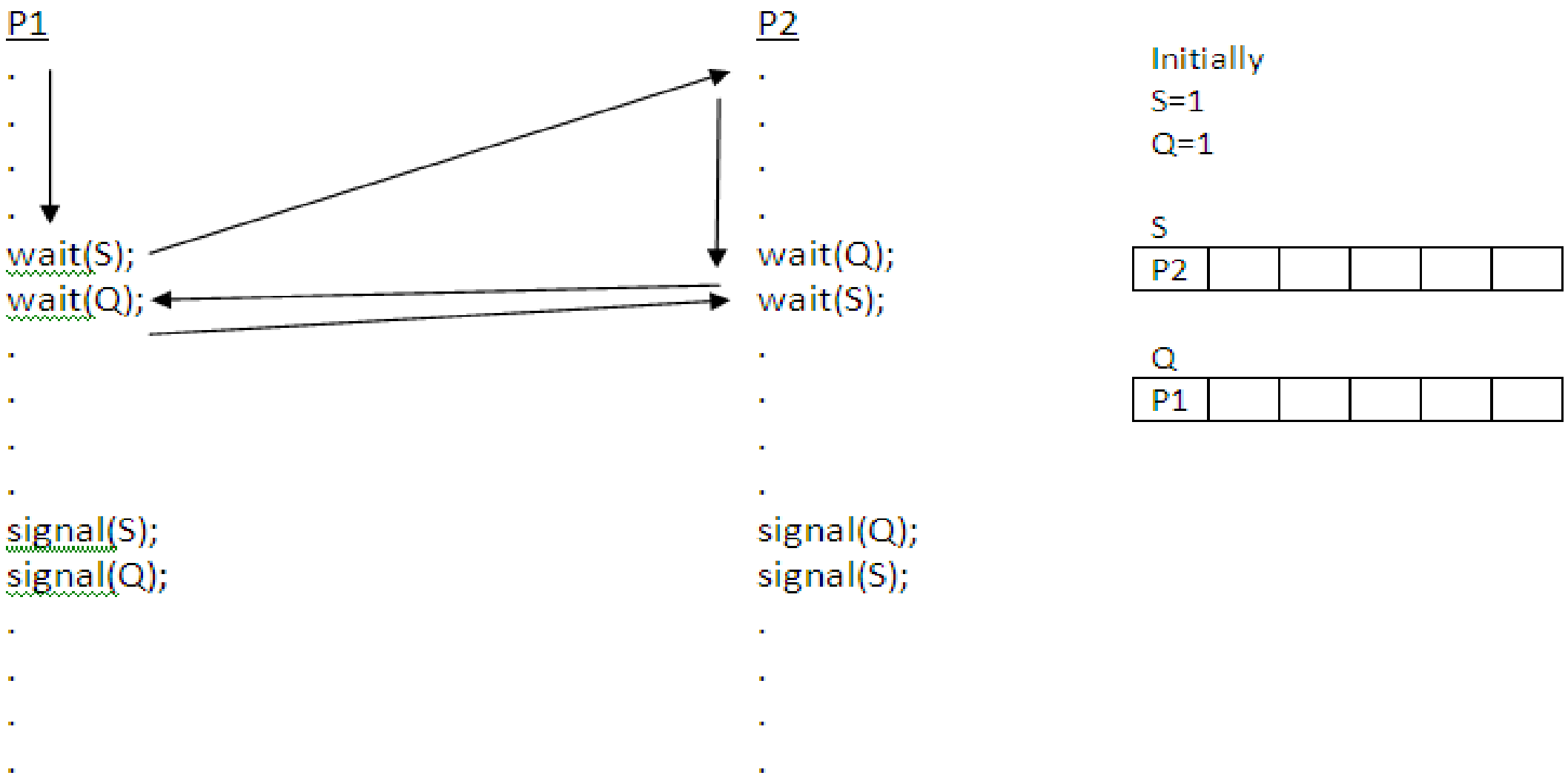
When the semaphore is used to solve the critical section problem then there is a possibility of occurring either deadlock or starvation or both.

Deadlock: a set of processes is said to be in deadlock state if each process in the set is waiting for another process in the set.

An example for the occurrence of deadlock with usage of semaphore

Assume that there are two processes P1 and P2 and two binary semaphore variables ‘S’ and ‘Q’.

The two processes are executing the wait() and signal() operations on the semaphore variables ‘S’ and ‘Q’ as shown below:


P1
P2
wait(S);
wait(Q);
wait(Q);
wait(S);
signal(S);
signal(Q);
signal(Q);
signal(S);
Initially
S=1
Q=1
S
P2
Q
P1

When the wait() operations on ‘S’ and ‘Q’ are executed in the order shown in above diagram then both process P1 and P2 will go to the waiting state.

Process P1 can be restarted when P2 executes signal(Q).

Similarly, process P2 can be restarted when process P1 executes Signal(S).

Because the processes are already blocked there is no way of executing signal(Q) and signal(S) by P2 and P1 respectively.

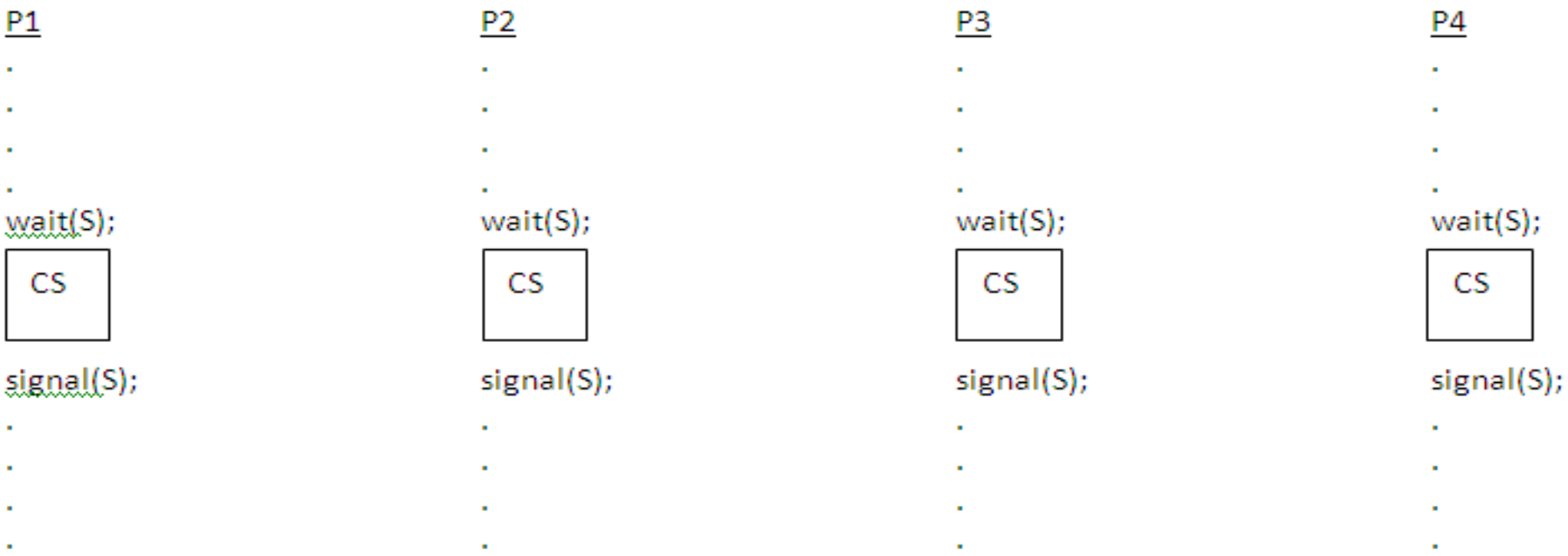
So, P1 and P2 are permanently blocked.

This situation is called a “deadlock state”.

Starvation is continuously waiting for something.

An example for the occurrence of starvation with the usage of semaphore

Assume that there are four processes P1, P2, P3 and P4. There is a semaphore variable ‘S’. The four processes are calling wait() and signal() operations on the semaphore variable ‘S’ as shown below.



Assume that process P1 is started for the first time.

Process P1 will enter its critical section as no other process is in its critical section.

While process P1 is executing in its critical section, if control is transferred to process P2 then P2 also tries to enter its critical section.

When P2 tries to enter its critical section then P2 is blocked and placed in the queue associated with semaphore variable ‘S’ as shown below

Now, if the control is transferred to P3 then P3 also tries to enter its critical section.

When P3 tries to enter its critical section then P3 is blocked and placed in the queue associated with semaphore variable ‘S’ as shown below

Now, if the control is transferred to P4 then P4 also tries to enter its critical section.

When P4 tries to enter its critical section then P4 is blocked and placed in the queue associated with semaphore variable ‘S’ as shown below

Now, if the control is transferred to P1 then P1 completes its critical section and executes signal(S).

When P1 executes signal(S) operation then one of the processes in the queue is removed and restarted.

If the processes from the queue are removed in LIFO order, then P4 is removed and restarted.

Now, P4 enters its critical section and executes signal(S) after completing its critical section.

When P4 executes signal(S) then P3 is removed and restarted.

Process P2 is continuously waiting in the queue. This situation is called as ‘starvation’.

If the processes in the waiting queue are removed in Last In First Out (LIFO) order, then the process that enters the queue for the first time may be continuously waiting to execute its critical section.

Monitor

Monitor is same as a class in ‘Java’ language. The structure of monitor is

```
Monitor Nameofmonitor
{
        Declaration of shared variables;
        Declaration of condition variables;

        Procedure1(list of parameters)
        {
                ------
        }
        Procedure2(list of parameters)
        {
                -----
        }
        -----
        ------
        Initializationcode( )
        {
                ----
        }
}
```

In a monitor, zero or more shared variables are declared using syntax rules of java language.

Similarly, zero or more condition variables are declared with the following syntax

condition listofvariables;

Zero or more procedures are defined. Procedures do not return any value.

Shared variables are used to store values and can be accessed only by the procedures of monitor.

Procedures of the monitor can access and perform operations on the shared variables of the monitor.

Initialization code is like a constructor in a java class and is used for initializing the shared variables of the monitor.

Condition variables are used to solve the critical section problem.

A queue is associated with each condition variable.

Two operations are defined on any condition variable

1. wait( )
2. signal( )

When any process calls wait() operation on any condition variable then the execution of that process will be stopped and will be placed into the queue associated with that condition variable.

When any process calls signal() operation on any conditional variable then a waiting process from the queue of the condition variable will be restarted.

At any time, only one process is allowed to call a procedure of the monitor.

When a process say P1 invokes a procedure of the monitor and the procedure is executing then at the same time if any other process say P2 tries to call a procedure of the monitor then the process P2 should wait.

With the above property of monitor, mutual exclusion is implemented automatically in the monitor.

Monitor solution for critical section problem of Producer – Consumer

```
Monitor PC
{
        int sizeofbuffer, Buffer[], in, out, count, item;
        condition full, empty;

        insert(int x)
        {
                if(count==sizeofbuffer)
                        wait(full);
                Buffer[in]=x;
                in=(in+1)%sizeofbuffer;
                count=count+1;
                signal(empty);
        }
```

```
        delete()
        {
                if(count==0)
                        wait(empty);
                item=Buffer[out];
                out=(out+1)%sizeofbuffer;
                count=count-1;
                signal(full);
        }
        Initialization()
        {
                sizeofbuffer=5;
                in=1;
                out=0;
                count=0;
        }
}
```

Producer Process

```
while(true)
{
        printf(“enter item\n”);
        scanf(“%d”,&item);
        PC.insert(item);
}
```

Consumer Process

```
while(true)
{
                PC.delete();
}
```

Classic problems of synchronization

1. Bounded buffer or producer-consumer problem
2. Readers-writers problem
3. Dining philosopher's problem

Readers-Writers Problem

There is a database that can be shared by number of processes.

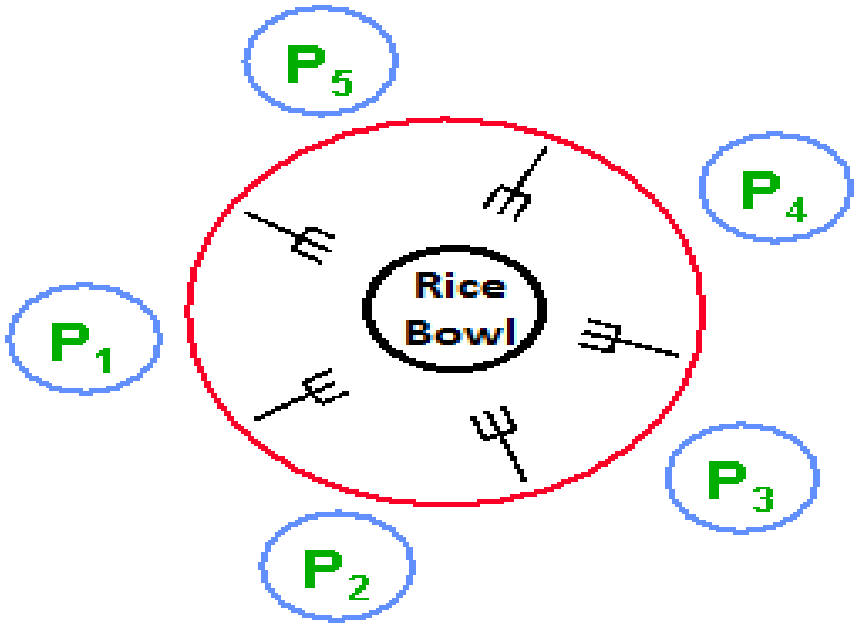
Some processes named ‘readers’ only read data from the database.

Other processes named ‘writers’ write data to the database.

At a time, any number of readers can read data from the database without any conflict.

But when a writer is writing data then other writers and readers are not allowed to access the database.

Dining Philosopher's Problem



There are 5 philosophers.

A philosopher at any time is in either thinking or hungry or eating state.

There is a table surrounded by 5 chairs which are used by the 5 philosophers.

There is a rice bowl in the center of table.

There are 5 plates and 5 spoons (chopsticks) and are placed on the table as shown in diagram.

A philosopher can eat food only when the two (left and right) chopsticks (spoons) are available for him.

A chopstick can be used by only one philosopher at a time.

In this problem, we need to implement mutual exclusion for the 5 chopsticks.