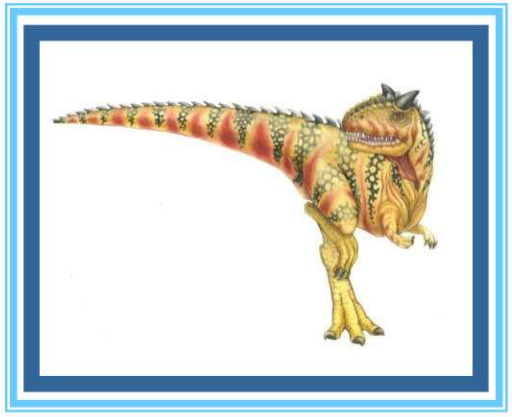
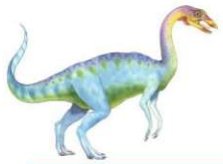
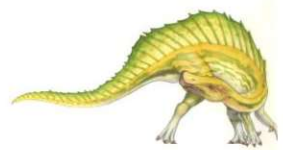
Process Synchronization





Types of Processes

- On the basis of synchronization, processes are categorized as one of the following two types:
- ***Independent Process*** : Execution of one process does not affect the execution of other processes.
- ***Cooperative Process :*** Execution of one process affects the execution of other processes. These process can either directly share a ***logical address space (code, data, memory and resources)*** or be allowed to share data only through files or message. Such processes need to be ***synchronized*** so that their order of execution can be guaranteed.



Process Synchronization

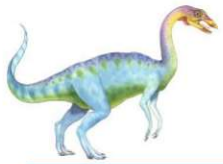
It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.

In order to synchronize the processes, there are various synchronization mechanisms.

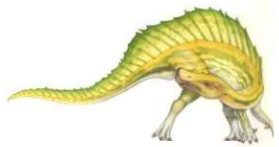
Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time

Process synchronization is basically a way to coordinate processes that use shared resources or data. It is very much essential to ensure synchronized execution of cooperating processes so that will maintain data consistency. Its main purpose is to share resources without any interference using mutual exclusion.



Process Synchronization

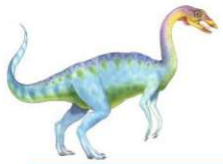
- ***Process Synchronization*** is a technique which is used to coordinate the process that use shared Data.
- Process synchronization problem arises in the case of ***Cooperative process*** also because resources are shared in Cooperative processes.
- ***Need:*** When two or more process concurrently accessing to the shared data, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced ***(data inconsistency, integrity risk).***



Race Condition

At the time when more than one process is either executing the same code or accessing the same memory or any shared variable; In that condition, there is a possibility that the output or the value of the shared variable is wrong so for that purpose all the processes are doing the race to say that my output is correct. This condition is commonly known as a race condition. As several processes access and process the manipulations on the same data in a concurrent manner and due to which the outcome depends on the particular order in which the access of data takes place.

Mainly this condition is a situation that may occur inside the critical section. Race condition in the critical section happens when the result of multiple thread execution differs according to the order in which the threads execute. But this condition is critical sections can be avoided if the critical section is treated as an atomic instruction. Proper thread synchronization using locks or atomic variables can also prevent race conditions.

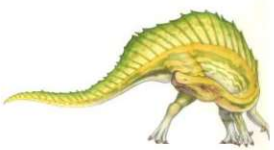


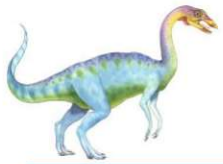
Race Condition (Example 1)

```
P(){
    Read(a)
    a=a+1
    Write (a)
}
```

If a=10 and p1 , p2 executes serially then final value of a will be 12.

If process P1 context switch after Read(a) and then P2 executes (concurrent fashion), inconsistent result (a=11) may occur.



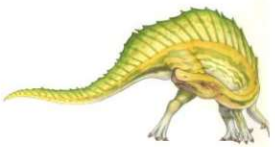


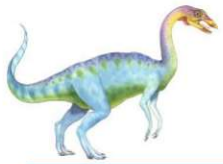
Race Condition (Example 2)

int z = 10;

P1{	*p2(){*
int x = z	*int y = z*
x = x+1	*y = y-1*
z = x	*z=y*
}	*}*

If process P1 and P2 executes serially then result is 10.

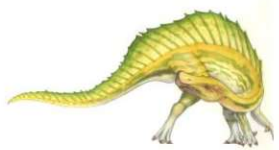


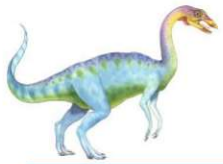


Race Condition (Example 2)

```
int z = 10;

P1{                          |  p2(){
    int x = z                |      int y = z
    x = x+1                  |      y = y-1
    //context switch         |      //context switch
    z = x                    |      z=y
}                            |  }
```

If process P1 and P2 executes above code in concurrent fashion (p1 context switch after first instruction or p2 context switch after first instruction) inconsistent results (11/9) may come.



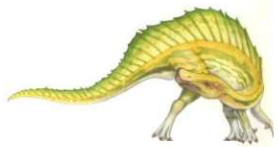


Race Condition

During the concurrent execution of the processes, where several processes access and manipulate the same data concurrently and outcome of the data depends over particular order in which access takes place is called ***RACE condition.***

Or

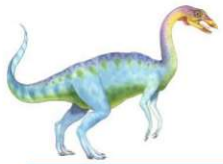
When order of execution can change result is called Race Condition.



Critical Section Problem

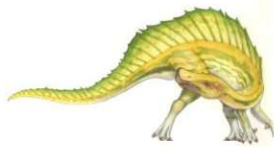
A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes. The entry to the critical section is mainly handled by wait() function while the exit from the critical section is controlled by the signal() function.

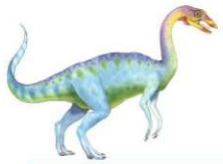
The regions of a program that try to access shared resources and may cause race conditions are called critical section. To avoid race condition among the processes, we need to assure that only one process at a time can execute within the critical section.



Critical Section Problem

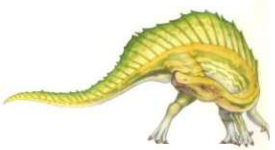
- Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses.
- In ***concurrent programming***, concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the ***shared resource is accessed*** need to be protected in ways that avoid the concurrent access.
- This protected section is the **critical section** or **critical region.** It cannot be executed by more than one process at a time.

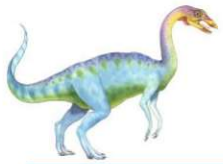




Critical Section Problem

- Every process has a ***reserved segment of code*** which is known as Critical Section. In this section, process can change ***common*** variables, update tables, write files, etc. The key point to note about critical section is that when one process is executing in its critical section, no other process can execute in its critical section.
- No two process can execute in their critical section at same time.
- ***Critical Section Problem:*** *CSP is* to design a protocol that the processes can use to cooperate.

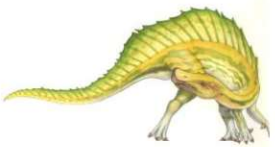


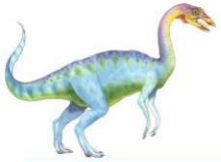


Critical Section

- Each process must request for permission before entering into its critical section and the section of a code implementing this request is the ***Entry Section***, the end of the code is the ***Exit Section*** and the remaining code is the remainder section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

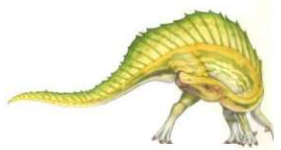


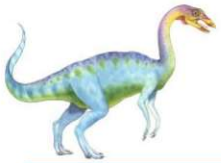


Critical Section Properties

Any solution to the critical section problem must satisfy three requirements:

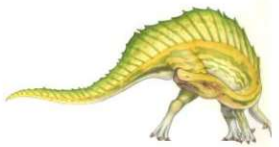
- **Mutual Exclusion** : If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress** : If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.





Critical Section Problem

• **Bounded Waiting** : In bounded waiting, there are limits or bounds on the number of times a process can enter its critical section after a process has made a request to enter its critical section and before that request is granted.



Critical Section problem solution using Turn Variable

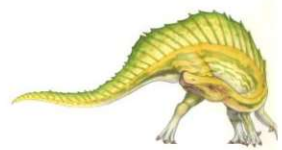
P0

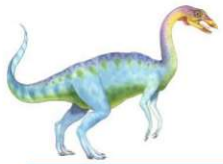
```
while(true)
{
    While (Turn != 0);

    Critical Section

    Turn = 1;

    Remainder Section

}
```

P1

```
while(true)
{
    While (Turn != 1);

    Critical Section

    Turn = 0;

    Remainder Section

}
```

Mutual exclusion is there but progress is not there.
Strict alternation is there between processes.





Producer Process

```
while(True){
        while(counter == Buffer_Size);      // do nothing
        buffer[in] = next_produced;         // in points to next free position
        in = (in+1) % Buffer_Size;
```

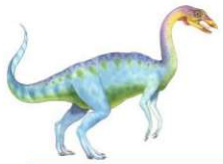
While (Turn != 0) → ***Entry section***

```
        register1 = counter;
        register 1 = register1 +1;
        counter =  register1;
```

Critical Section

Turn = 1 → ***Exit section***

```
}
```



Consumer Process

```
while(True){
        while(counter == 0);                    // do nothing
        next_consumed = buffer[out];            // Consumes item in next_consumed
        out = (out + 1) % Buffer_Size;          // out points to first full position
```

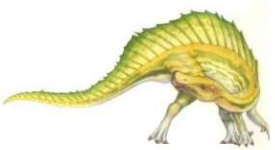
While (Turn != 1) → *Entry section*

```
        register2 = counter;
        register2 = register2 -1;
        counter =  register2;
```

Critical Section

Turn = 0 → *Exit section*

```
}
```



Critical Section Problem (Producer/consumer)

```
while(True){
        while(counter == Buffer_Size);
        buffer[in] = next_produced;
        in = (in+1) % Buffer_Size;
```

While (Turn != 0)

```
        register1 = counter;
        register 1 = register1 +1;
        counter =  register1;
```

Turn = 1

```
}
```

```
while(True){
        while(counter == 0);
        next_consumed = buffer[out];
        out = (out + 1) % Buffer_Size;
```

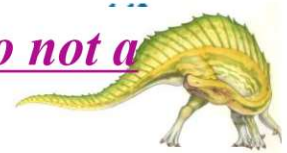
While (Turn != 1)

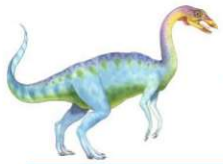
```
        register2 = counter;
        register2 = register2 -1;
        counter =  register2;
```

Turn = 0

```
}
```

For this solution it meets mutual exclusion but progress doesn't meet, so not a valid solution.





Critical Section problem solution using flag

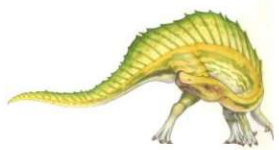
P0

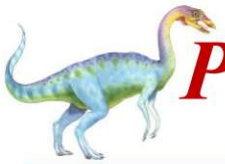
```
while(true)
{
    flag [0] = T;
    while (flag[1]);

    Critical Section

    flag[0]=F;
}
```

P1

```
while(true)
{
    flag [1] = T;
    while (flag[0]);

    Critical Section

    flag[1]=F;
}
```

Mutual exclusion is there but progress is not there system will go in deadlock if context switch happen after first statement flag[0] and p1 and p2 both are interested to get enter into critical section





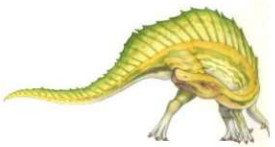
Peterson's Solution (software based solution)

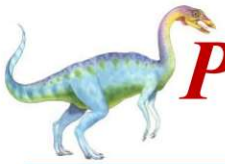
P0

```
while(true)
{
    flag [0] = T;
    turn = 1;
    while (turn==1 && flag[1] ==T);

        Critical Section

    flag[0]=F;

}
```

P1

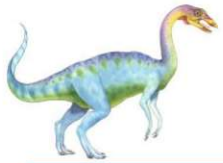
```
while(true)
{
    flag [1] = T;
    turn = 0;
    while (turn==0 && flag[0] ==T);

        Critical Section

    flag[1] = F;

}
```





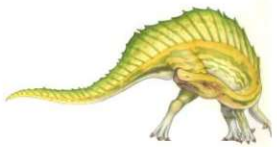
Peterson's Solution (software based solution)

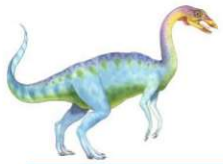
```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```



Peterson's Solution (Drawbacks)

1. ***Peterson's solution*** works only for two processes, but this solution is best scheme in user mode for critical section.
2. Software-based solutions (***Peterson's solution***) are not guaranteed to work on modern computer architectures.





Critical Section Solution

Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned

P1	P2
while (S1 == S2) ;	***while (S1 != S2) ;***
Critica1 Section	Critica1 Section
S1 = S2;	***S2 = not (S1);***

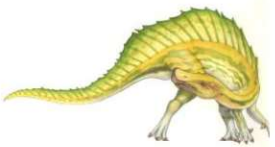
Which one of the following statements describes the properties achieved?

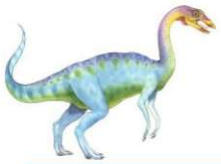
(A) *Mutual exclusion but not progress*

(B) *Progress but not mutual exclusion*

(C) *Neither mutual exclusion nor progress*

(D) *Both mutual exclusion and progress*

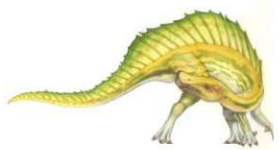


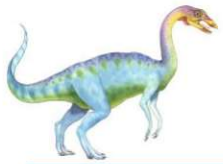


Critical Section Problem

There are two approaches that are commonly used in operating system to handle critical section.

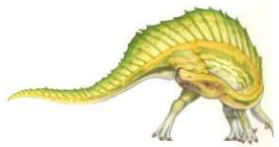
- ***Preemptive Kernel*** − A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- ***Non-Preemptive Kernels*** − A non-preemptive kernel doesn't allow a process running in kernel mode to be preempted.

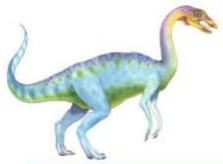




Synchronization Hardware

- Hardware features can make any programming task easier and improve efficiency.
- ***Interrupt prevention*** (no preemption) while a shared variable is being modified, could be a solution to critical section problem on single processor systems. Unfortunately this is not a solution on multiprocessor environment following problems could occur.
  - Time consuming on multiprocessor.
  - System Efficiency will decrease.
  - System clock could be effected.

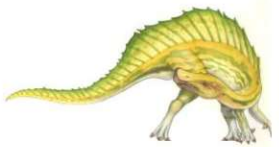


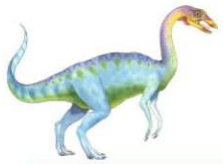


Synchronization Hardware

- ***Locking:*** Protecting critical regions through the use of ***locks***.

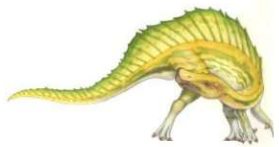
```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

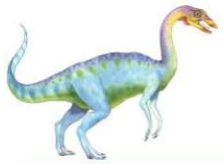




Mutex locks

- Hardware-based solutions to the critical section problem are complicated and inaccessible to application programmers.
- ***Mutex lock:*** It a software tool to solve critical section problem and it is taken from (***Mut****ual* ***Ex****clusion*). Mutex locks are used to protect critical section and to prevent race conditions.
- A process must ***acquire the lock*** before entering a critical section, it ***releases the lock*** when it exits the critical section.
- The ***acquire()*** function will acquire the lock and ***release()*** function will release the lock.



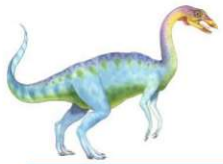


Mutex locks

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

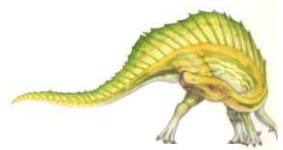
```
acquire(){
        while (!Available);
        available =false;
}

release(){
        available =true;
}
```

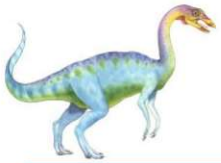
Calls to acquire and release should be atomic usually implemented by hardware instructions.



Mutex locks disadvantage

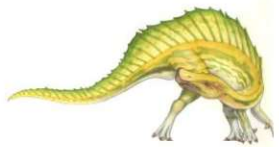
- ***Busy waiting:*** While a process is in its critical section any other process that tries to enter its critical section must loop continuously in the call to acquire(), this type of mutex lock is also called a spinlock because the process "spins" while waiting for lock to become available.
- This continual looping is clearly a problem in real multiprogramming systems where single CPU is shared among many processes.
- Busy waiting ***waste CPU cycles*** that some other process might be able to use productively.

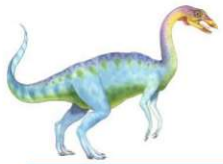




Mutex locks Advantage

- ***Busy waiting/Spinlock:*** No context switch is required, when a process must wait on lock, and context may take considerable time.
- On multiprocessor systems one thread can spin on one processor while another thread performs its critical section on another processor.



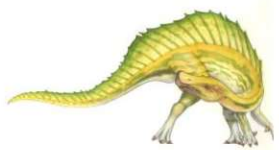


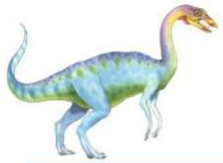
Semaphore

A ***semaphore (S)*** is an ***integer variable*** that, apart from initialization, is accessed only through two standard atomic operations: ***wait(S)*** and ***signal(S).*** The wait() operation was originally termed as ***P*** and signal operation is termed as ***V***.

Applications:

- Semaphore are used to solve the critical section problem
- Process Synchronization
- Resource management.





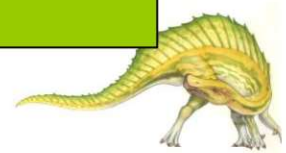
Semaphore

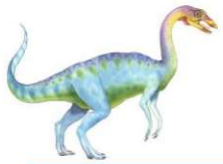
Wait(S) / P: The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

Signal(S) / V: The signal operation increments the value of its argument S

```
wait(S){
        while (S<=0);
        S--;
}
```

```
signal(S){
        S++;
}
```

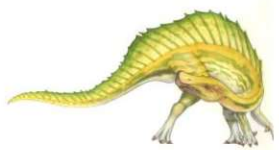


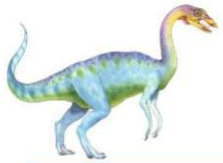


Types of Semaphore

There are two main types of semaphores:

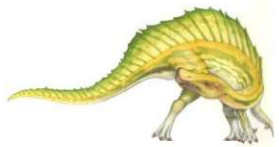
- ***Binary Semaphores:*** This is also known as mutex lock and their value can range between 0 and 1.
- It is used to implement the solution of critical section problem with multiple processes. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.
- ***Counting Semaphores:*** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

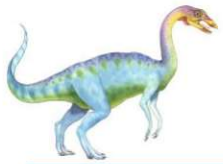




Types of Semaphore

- ***Counting semaphores*** are used to coordinate the resource access, where the semaphore count is the number of available resources. Each process that wishes to use resource need to perform wait() on semaphore, when process release the resource it will perform signal() on semaphore. When count of the semaphore goes to 0 all resources are being used. After that wish to use a resource will block until count becomes greater than 0.



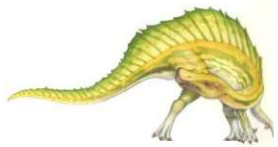


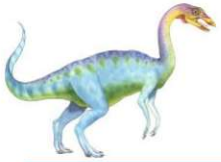
Types of Semaphore

- ***Semaphore can be used to solve various synchronization problems.***

Example: Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we want that S2 should be execute only after s1 has completed.

S1:	***wait (S);***
signal(S)	***S2***

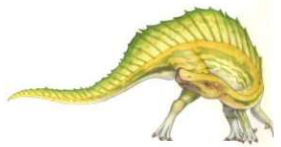




Semaphores without busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - a list of processes list

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.



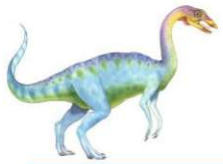
- Implementation of wait:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

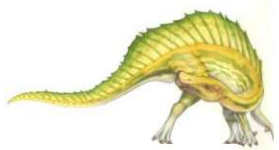
Semaphores without busy waiting

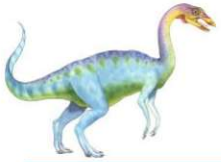


Advantages of Semaphores

Some of the advantages of semaphores are as follows:

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

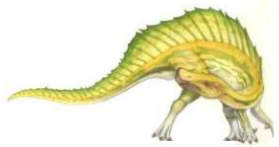


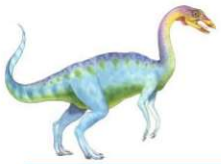


Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.





Semaphores Example - 1

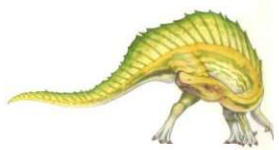
A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution?

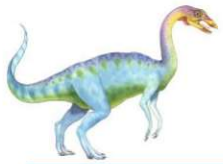
(A) -2

(B) -1

(C) 1

(D) 2





```
Process P:
while (1) {
W:
    print '0';
    print '0';
X:
}
```

```
Process Q:
while (1) {
Y:
    print '1';
    print '1';
Z:
}
```

Synchronization statements can be inserted only at points W, X, Y and Z.

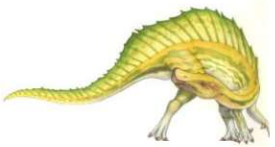
Which of the following will always lead to an output staring with '001100110011'?

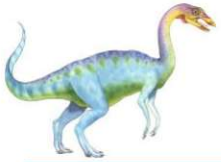
(A) ***P(S)*** at ***W***, ***V(S)*** at ***X***, ***P(T)*** at ***Y***, ***V(T)*** at ***Z***, ***S and T initially 1***

(B) ***P(S)*** at ***W***, ***V(T)*** at ***X***, ***P(T)*** at ***Y***, ***V(S)*** at ***Z***, ***S initially 1, and T initially 0*** ⟸

(C) ***P(S)*** at ***W***, ***V(T)*** at ***X***, ***P(T)*** at ***Y***, ***V(S)*** at ***Z***, ***S and T initially 1***

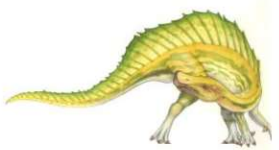
(D) ***P(S)*** at ***W***, ***V(S)*** at ***X***, ***P(T)*** at ***Y***, ***V(T)*** at ***Z***, ***S initially 1, and T initially 0***

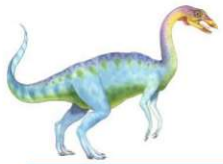




Bounded Buffer Problem

- There is a buffer of n slots and each slot is capable of storing ***one unit*** of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.
- A producer tries to ***insert*** data into an **empty** slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.
- There needs to be a way to make the producer and consumer work in an independent manner.





Solution of Bounded Buffer Problem

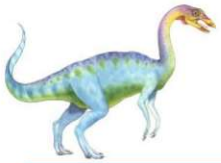
- ***mutex =1***, a **binary semaphore** which is used to acquire and release the lock.
- ***empty***, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- ***full***, a **counting semaphore** whose initial value is 0

The structure of the **producer** process

```
do {
        //  produce an item in nextp

        wait (empty);
        wait (mutex);

        //  add the item to the  buffer

        signal (mutex);
        signal (full);

} while (TRUE);
```

The structure of the **consumer** process

```
do {
        wait (full);
        wait (mutex);

        //  remove an item from
        //  buffer to nextc

        signal (mutex);
        signal (empty);

        //  consume the item in nextc
} while (TRUE);
```



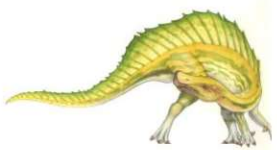
Readers Writers Problem

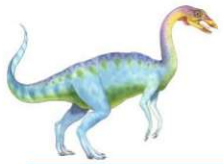
A data set is shared among a number of concurrent processes

Readers – only read the data set; they do not perform any updates

Writers – can both read and write.

- ***Problem*** – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.





Solution of Readers Writers Problem

Shared Data:

Semaphore ***mutex*** initialized to ***1***.

Semaphore ***wrt*** initialized to ***1***.

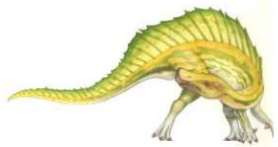
Integer ***readcount*** initialized to ***0***.

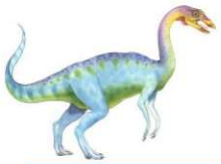
The structure of a ***writer*** process

```
while (true) {
        wait (wrt) ;
        write operation
        signal (wrt) ;
}
```

The structure of a ***reader*** process

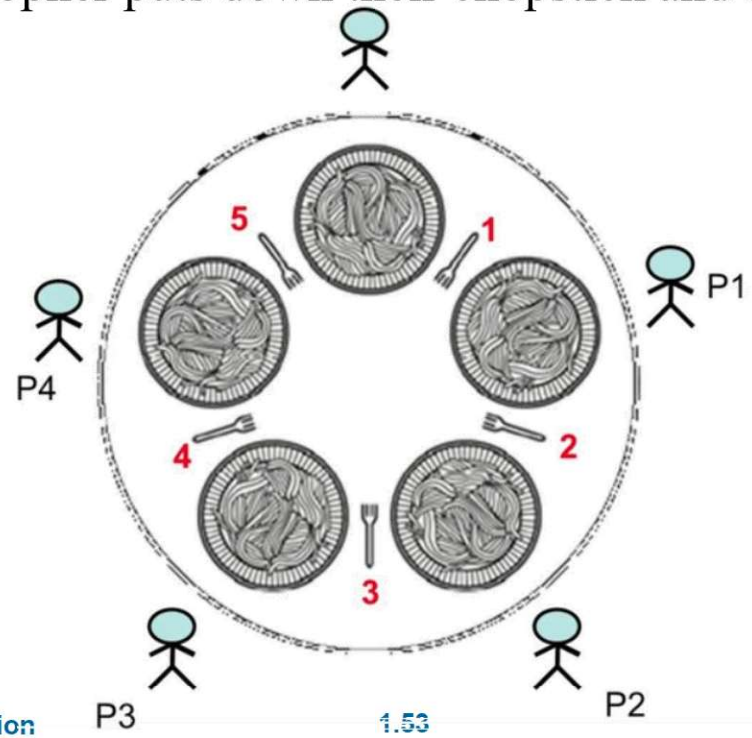
```
while (true) {
    wait (mutex) ;
    readcount ++ ;
    if (readercount == 1)
        wait (wrt) ;
    signal (mutex)

        Read Operation

    wait (mutex) ;
    readcount  - - ;
    if (redacount  == 0)
        signal (wrt) ;
    signal (mutex) ;
}
```

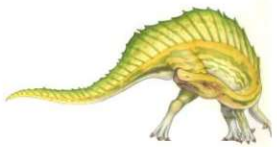


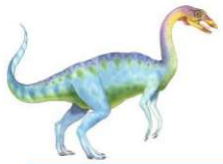


Dining-Philosophers Problem

- There are five silent philosophers (P1 – P5) sitting around a circular table, and they ***eat and think alternatively***.
- There is a bowl of rice for each of the philosophers and 5 chopsticks (1 – 5).
- To be able to eat, a ***philosopher needs to have chopstick in both his hands***. A hungry philosopher may only eat if there are both chopsticks available.
- After eating a philosopher puts down their chopstick and begin thinking again.





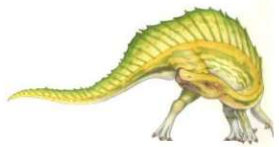


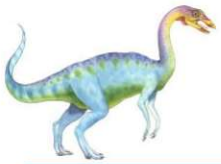
Dining-Philosophers Problem Solution

- A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick.
- A chopstick can be picked up by executing a ***wait*** operation on the ***semaphore*** and released by executing a ***signal*** semaphore.
- The structure of the chopstick is shown below:

semaphore chopstick [5];

- Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.

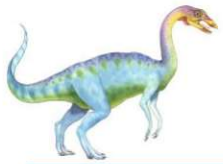




Dining-Philosophers Problem Solution

semaphore chopstick [5];

```
do {
        wait( chopstick[i] );
        wait( chopstick[ (i+1) % 5] );

        Eating the Rice

        signal( chopstick[i] );
        signal( chopstick[ (i+1) % 5] );

        Thinking

} while(1);
```

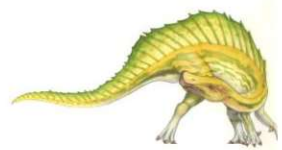


Difficulty with the solution

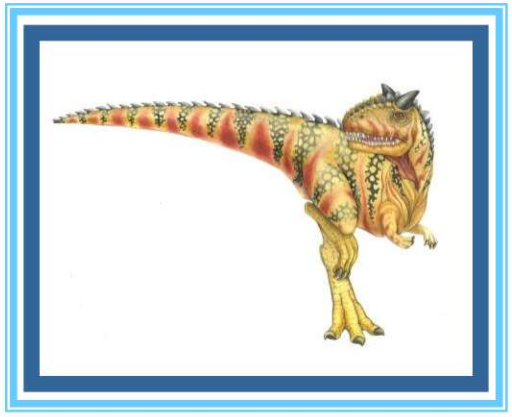
The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a ***deadlock***. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

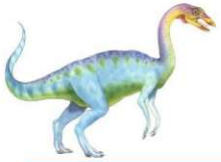
Some of the ways to ***avoid deadlock*** are as follows:

- ***There should be at most four philosophers on the table.***
- ***An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.***
- ***A philosopher should only be allowed to pick their chopstick if both are available at the same time.***



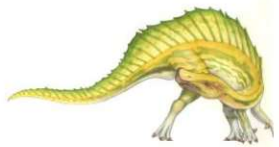
Deadlocks

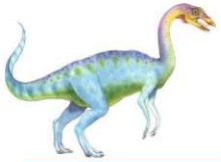




Contents

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

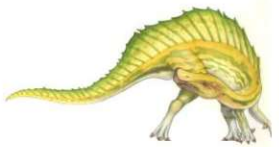


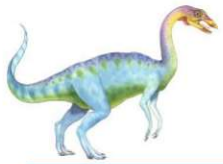


Resource Use

- A process in operating systems uses different resources and uses resources in following way.

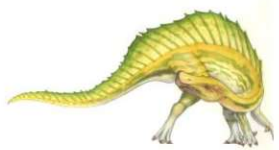
1. ***Requests** a resource*
2. ***Use** the resource*
3. ***Releases** the resource*

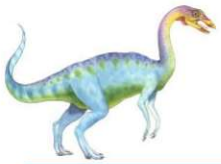




Deadlock

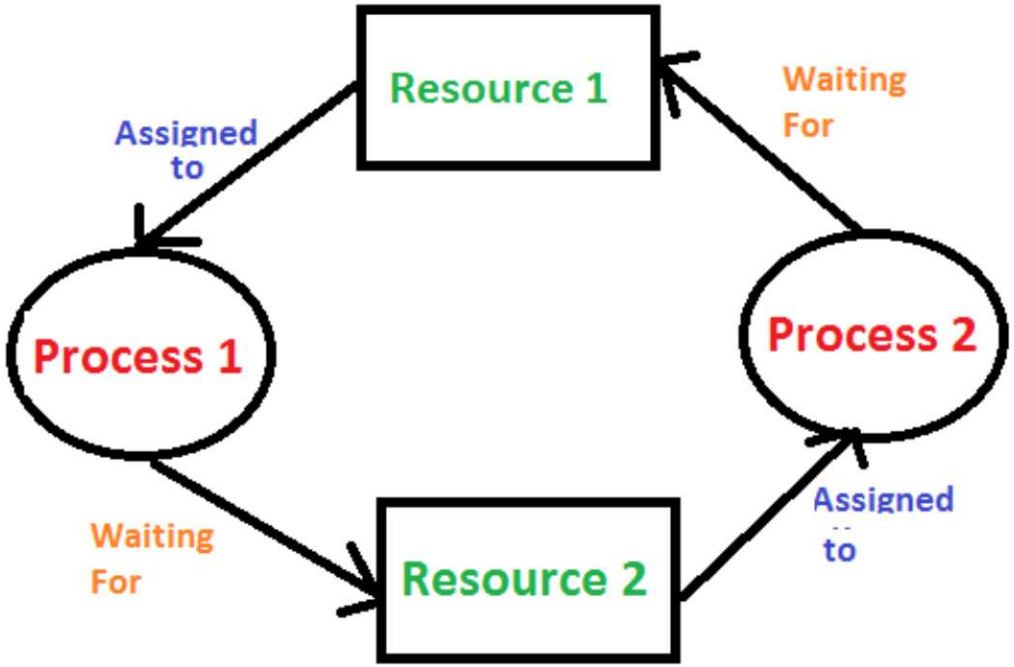
In multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources, if the resources are not available at that time, the process enters in waiting state. ***Sometimes a waiting process is never again able to change state, because the resources it has requested are held by other processes. This situation is called a deadlock.***

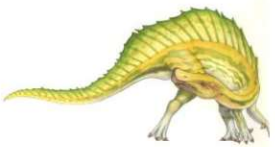


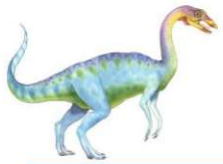


Deadlock

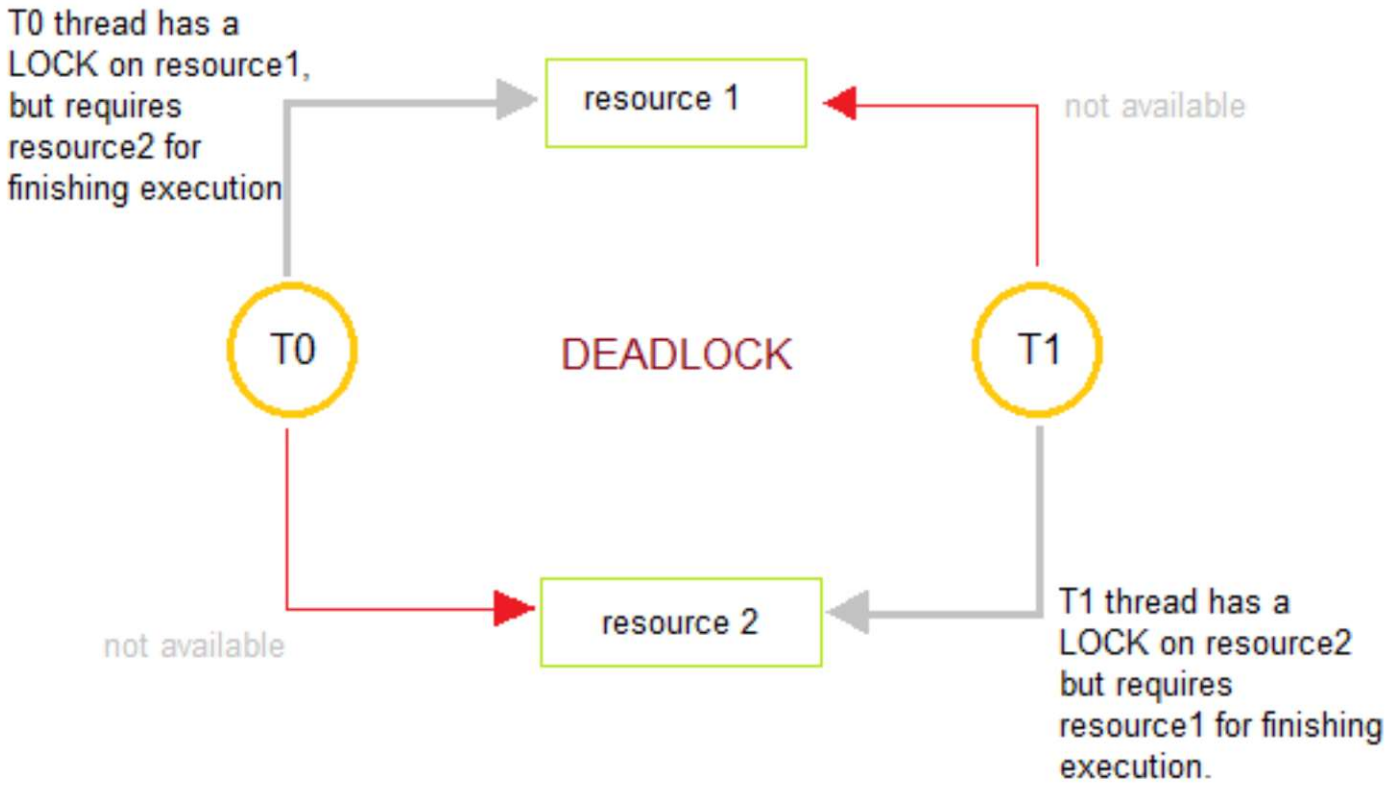
- A set of ***blocked processes*** each holding a resource and waiting to acquire a resource held by another process in the set. Resource may be either physical resource (I/O devices, tape drives, CPU cycles, memory) or logical resources (semaphor

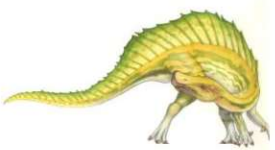


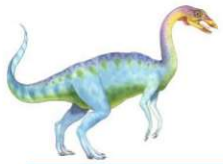




Deadlock


T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

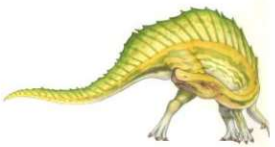


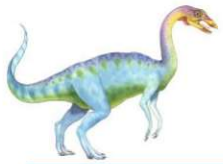


Deadlock Examples

- ***Example***
  - System has 2 ***tape drives***.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- ***Example***
  - semaphores ***A*** and ***B***, initialized to ***1***

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

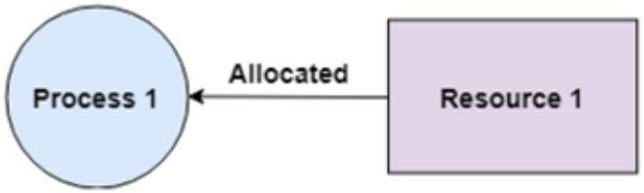


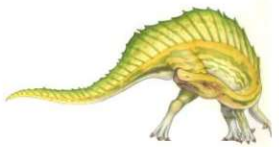


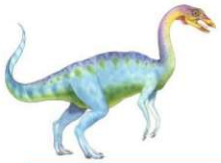
Necessary Conditions

Deadlock can arise if four conditions hold simultaneously. All four conditions must hold for a deadlock to occur

- ***Mutual exclusion:*** At least one resource must be held in a ***nonsharable mode***, that is ***only one process at a time can use a resource***. If another process request the resource, the requested process must be delayed until the resource has been released.

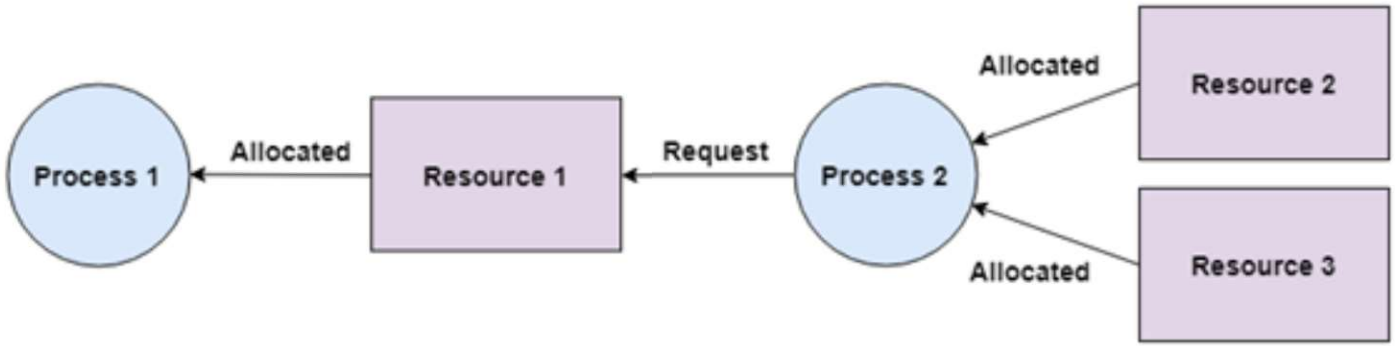


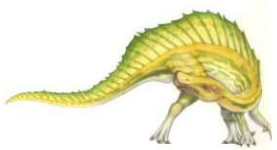


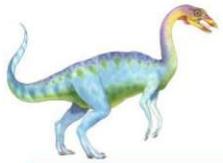


Necessary Conditions

- ***Hold and wait:*** a process holding at least one resource is waiting to acquire additional resources held by other processes.

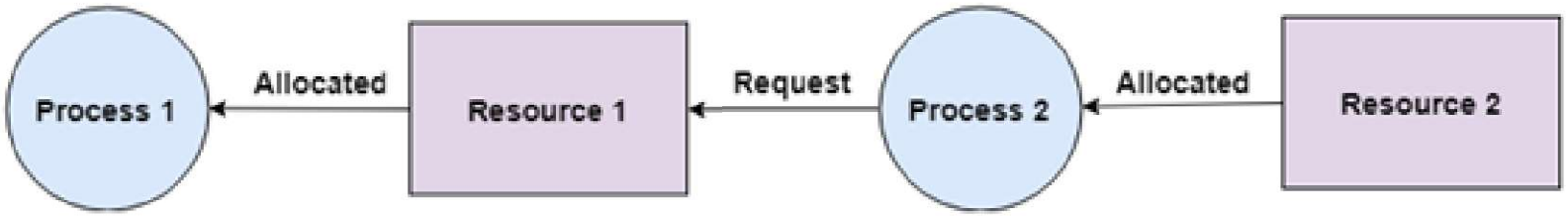


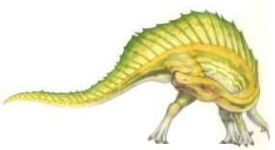


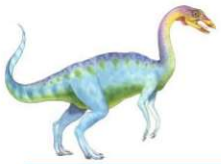


Necessary Conditions

- ***No preemption:*** a resource can be released only voluntarily by the process holding it, after that process has completed its task.







Necessary Conditions

- ***Circular wait:*** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.