UNIT 3

What is a semaphore? Explain how semaphores can be used to deal with n-process critical section.

Ans: semaphore is a hardware-based solution to the critical section problem.

A Semaphore S is a integer variable that, apart from initialization is accessed only through two standard atomic operations: wait() and signal().

The wait () operation is termed as P and signal () was termed as V

Definition of wait () is

```
Wait (S) {
    While  S <= 0
        ;  S--;
    }
```

Definition of signal () is

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

All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly, that is when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Usage:

Operating systems often distinguish between counting and binary semaphores. The value of counting semaphore can range over an unrestricted domain and the binary semaphore also known as mutex locks which provide mutual exclusion can range only between 0 and 1. Binary semaphores are used to deal with critical-section problem for multiple processes as n processes share a semaphore mutex initialized to 1.

```
do {
    Wait (mutex) ;
    // critical section
```

```
            Signal (mutex);
        // remainder section
        } while (TRUE);
```

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. Each process that wishes to use a resource performs a wait () operation on the semaphore. When the count for the semaphore goes to 0, all resources are being used. And the process that wish to use a resource will block until the count becomes greater than 0.

Implementation:

The main disadvantage of the semaphore is that it requires busy waiting. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process spins while waiting for the lock. To overcome the need for busy waiting, we can modify the definition of wait () and Signal () semaphore operations.

When the process executes the wait () operation and finds that the semaphore value is not positive it must wait. Rather that engaging in busy waiting the process can block itself. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state.

The process that is blocked waiting on a semaphore S should be restarted when some other process executes a signal () operation, the process is restarted by a wakeup () operation.

Definition of semaphore as a C struct

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation remove one process from the list of waiting processes and awakens that process.

Wait () operation can be defined as

```
Wait (semaphore *s) {
    S->value--;
    If (s->value <0) {
        Add this process to S->list;
        block (); } }
```

Signal operation can be defined as

```
Signal (semaphore *S) {
S->value++;
If (S-> value <=0) {
    Remove a process P from S-> list;
        Wakeup (P);  } }
```

The block () operation suspends the process that invokes it. The wakeup () operation resumes the execution of a blocked process P.

This is a critical section problem and in a single processor environment we can solve it by simply inhibiting interrupts during the time the wait () and signal () operations are executing and this works in single processor. Where as in multi processor environment interrupts must be disabled on every processor.

<u>Deadlock and Starvation:</u>

The implementation of semaphore with a waiting queue may result in a satiation where two more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such a state is reached that process are said to be deadlocked.

```
        P0              p1
    Wait(S);        wait(Q);
    Wait(Q);        Wait(S);
    .                          .
    .                          .
    .                          .
    Signal( S);                Signal (Q);
    Signal (Q);                Signal (S);
```

P0 executes wait (S) and then P1 executes wait (Q). When p0 executes wait (Q), it must wait until p1 executes Signal (Q) in the same way P1 must wait until P0 executes signal (S). So p0 and p1 are deadlocked.

The other problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore.

<u>Priority inversion:</u>

Scheduling challenges arises when a higher priority process needs to read or modify kernel data that are currently used by lower priority process. This problem is known as priority inversion and it occurs in systems with more than two priorities and this can be solved by implementing priority- inheritance protocol. According to this, all processes that are accessing resources needed by a higher priority process inherit the higher priority until they are finished with the resources.

1. What is deadlock? What are the necessary conditions of deadlock?

Ans: A process requests resources, if the resources are not available at that time; the process enters a waiting state. Sometimes a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

Necessary conditions of deadlock or deadlock characterization:

A deadlock situation can rise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion:** At least one resource must be held in a non-sharable mode, that is, only one processes at a time can use the resource. If another process requests

that resource, the requesting process must be delayed until the resources have been released.

2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No pre-emption:** Resources cannot be pre-empted; that is, resources can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait**: A set {p0,p1,p2....} of waiting processes must exist such that p0 is waiting for a resource held by p1, p1 is waiting for a resource held by p2,....pn-1 is waiting for a resource held by pn, and pn is waiting for a resource held by p0.

2. Explain a deadlock situation with resource allocation graph?

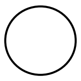
Ans: deadlocks can be described in terms of a directed graph called a system resources-allocation graph. This graph consists of A set of vertices V and a set of edges E.

V is partitioned into two types of nodes:

- P = {P1, P2, …, Pn}, the set consisting of all the processes in the system.
- R = {R1, R2, …, Rm}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi->Rj is called as **requesting edge** and it signifies that process Pi has requested an instance type Rj and is currently waiting for that resource.

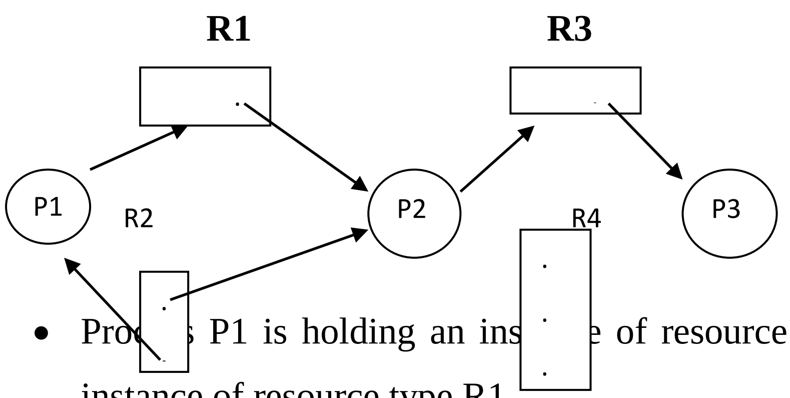
A directed edge Pi -> Rj is called a **assignment edge**, it signifies that resource type Rj has been allocated to process Pi

Process Pi is represented as Circle 

And each resource type Rj is represented as a rectangle

And since each resource type Rj can have more than one instance represented each such instance as a dot with in the rectangle

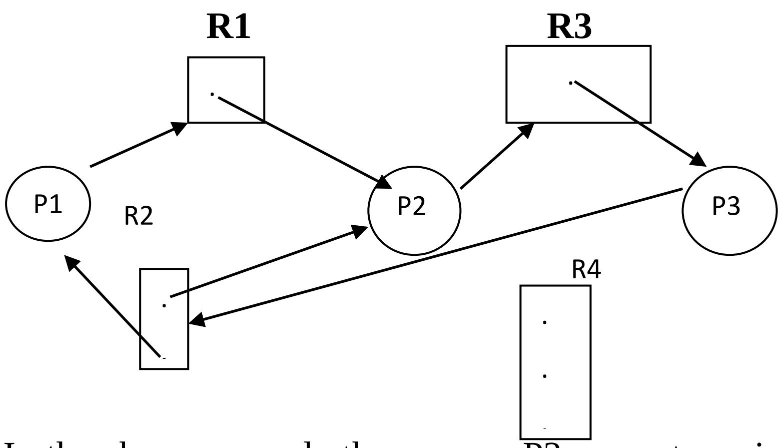
Example of a Resource Allocation Graph



- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1
- Process p2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3
- Process P3 is holding an instance of R3

If the graph contains no cycles then no process in the system is deadlocked. If the graph contains a cycle then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock had occurred. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

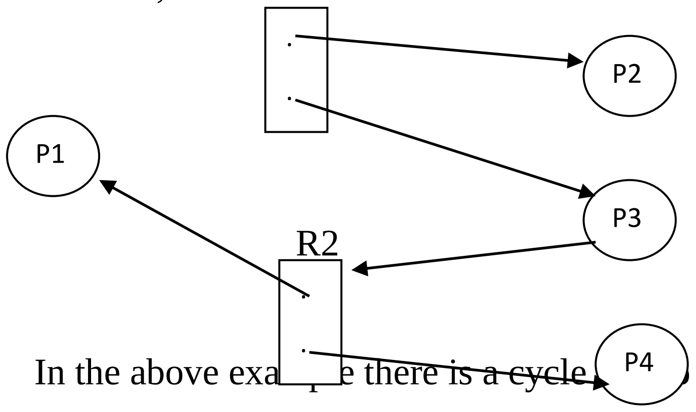


In the above example the process P3 requests an instance of resource type R2, and no resource instance is available, a request edge P3->R2 is added to the graph. At this point two minimal cycles exist in the system

P1->R1->P2->R3->P3->R2->P1

P2->R3->P3->R2->P2

Processes P1, P2 and P3 are deadlocked



In the above example there is a cycle but no deadlock.

P1->R1->P3->R2->P1

Process P4 may release its instance of resource type R2 and that resource can be allocated to P3 breaking the cycle.

3. What are the methods to handle and prevent deadlock?

Ans:

- Ensure that the system will never enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system
- used by most operating systems, including UNIX and Windows

For a deadlock to occur each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of deadlock.

Mutual Exclusion – not required for sharable resources and read only files are a good example of sharable resources but must hold for non-sharable resources.

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources

1. Require process to request and be allocated all its resources before it begins execution, or

2. Allow process to request resources only when the process has none – it must release its current resources before requesting them back again

Disadvantages: –

Resource utilization may be low since resources may be allocated but unused for long period

Starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption –

- If a process, that is holding some resources, does request another resource that cannot be immediately allocated to it, then all its resources currently being held are released
- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

This process is often applied to resources whose state can be easily saved and restored later such as CPU registers and memory space.

Circular Wait – impose a total ordering of all resource types

- require that each process requests resources in an increasing order of enumeration, or release resources of higher or equal order 4
- Several instances of one resource type must be requested in a single request
- a witness may detect a wrong order and emit a warning
- Lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically.

4. **Deadlock Avoidance**

   Ans: Avoiding deadlocks requires additional information about how resources are to be requested. The most simple and useful model requires that each process declare maximum number of resources of each type that it may need. With this prior information it is possible to construct an algorithm that ensures the system will never enter into deadlock state.

Deadlock- avoidance algorithm dynamically examines the resource allocation state o ensure that a circular wait condition can never exist and resource allocation state is defined by the number of available and allocated resources and the maximum demands of the process.

Safe state:

A **state is safe** if the system can allocate resources to each process in some order and still avoid a deadlock. A system is in a safe state if there exists a safe sequence.

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

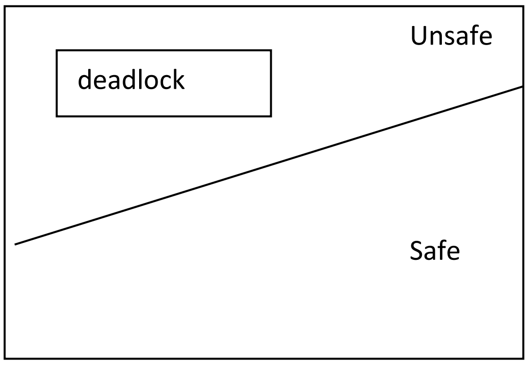
Sequence is safe if for each Pi, the resources that Pi can still request, can be satisfied by currently available resources + resources held by all the Pj, with j < i

Formally, there is a safe sequence such that for all i = 1, 2, ..., n,

Available + Σ1≤ k ≤ i (Allocatedk) ≥ MaxNeedi

1. If Pi resource needs are not immediately available, then P can wait until all P (j < i) have finished.
2. When Pj (j < i) is finished, Pi can obtain needed resources, execute, return allocated resources, and terminate.
3. When Pi terminates, Pi+1 can obtain its needed resources, and so on.

A safe state is not a deadlocked state. A deadlocked state is an unsafe state and not all unsafe states are deadlocks. As long as the state is safe, the operating system acn avoid unsafe states.

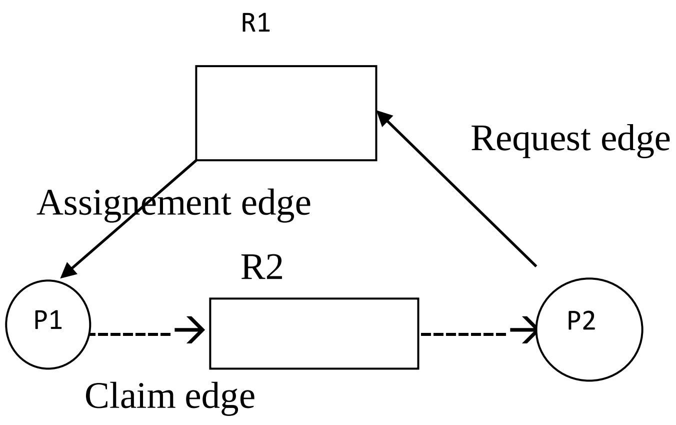


Resource- Allocation- Graph Algorithm

If we have a resource-allocation system with only one instance of each resource, we use a variant of the resource- allocation graph for deadlock avoidance called **claim edge**

- Claim edge Pi → Rj indicated that process Pj may request resource Rj; represented by a dashed line.
- Request edge Pi → Rj when a process requests a resource.
- When a resource is released by a process, assignment edge Rj → Pi reconverts to a claim edge Pi → Rj.

Resources must be claimed a priori in the system. Grant a request only if no cycle created.

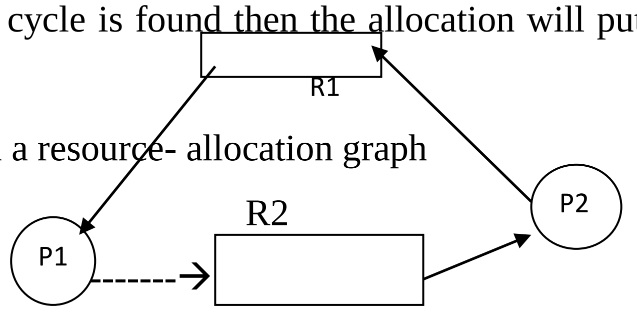


Resource- allocation graph for deadlock avoidance

The requesting edge can be granted only if converting the request edge Pi->Rj to an assignment edge Rj->Pi does not result in the formation of a cycle in the resource allocation graph. Cycle – detection algorithm is used to check the safety. To detect a cycle in this graph requires an order of n^2 operations where n is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found then the allocation will put the system in unsafe state.

Unsafe state in a resource- allocation graph



Bankers Algorithm

- Bankers Algorithm is applicable to systems having multiple instances but it is less efficient than the resource allocation graph.
- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number must not exceed the total number of resources available in the system
- Allocation of resources is done to a requesting process until the process leaves system is in safe state.

Several data structures must be maintained to implement the banker's algorithm.

Let n = number of processes, and m = number of resources types.

- Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available.
- Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type R request at most k instances of resource type Rj .
- Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rj.
- Need: n x m matrix. If Need [i,j] = k, then Pi may need k more instances of Rj to complete its task.

  Need[i, j] = Max[i, j]- Allocation [i, j]

<u>Safety Algorithm</u>

This algorithm is used to find out whether or not a system is in a safe state.

Let Work and Finish be vectors of length m and n, respectively.

Initialize: Work = Available Finish [i] = false for i = 0, 1, …, n-1. 2.

Find an i such that both:

(a) Finish [i] = false

(b) Needi ≤ Work

If no such i exists, go to step 4.

3. Work = Work + Allocation

Finish [i] = true

go to step 2.

4. If Finish [i] == true for all i, then the system is in a safe state.

This algorithm requires an order of m x n^2 operations to determine whether a state is safe.

Resource Resource-Request Algorithm

Request = request vector for process Pi. If Requesti [j] = k then process Pi wants k instances of resource type Rj.

1. If Requesti ≤ Needi go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. 2. If Requesti ≤ Available, go to step 3. Otherwise Pi must wait, since resources are not available.
3. Pretend to allocate requested resources to Pi by modifying the state as follows:

   Available = Available - Requesti;

   Allocationi = Allocationi + Requesti;

   Needi = Needi – Requesti;

   If safe ⇒ the resources are allocated to Pi. If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

5. Deadlock Detection

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock.

Single Instance of Each Resource Type: If all resources have only a single instance, then we define a deadlock detection algorithm that use a variant of the resource- allocation graph called a **wait- for graph.**

- An edge from Pi to Pj in wait- for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.
- An edge Pi $\rightarrow$ Pj exists in a wait- for graph if and only if the corresponding resource- allocation graph contains two edges Pi $\rightarrow$Rq and Rq $\rightarrow$Pj for some resource Rq.

<u>Diagram in page no 302 from text book</u>

Deadlock exists in the system if and only if the wait- for graph contains a cycle

To detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type:

- Available: A vector of length m indicates the number of available resources of each type.
- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.
- Request: An n x m matrix indicates the current request of each process. If Request [i j] = k, then process Pi is requesting k more instances of resource type Rj.

Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively Initialize: (a) Work = Available (b)For i = 1,2, …, n, if Allocationi ≠ 0, then Finish[i] = false; otherwise, Finish[i] = true.

2. Find an index i such that both:

(a)Finish [i] == false

(b)Request i ≤ Work If no such i exists, go to step 4

3. Work = Work + Allocationi

Finish [i] = true

go to step 2.

4. If Finish [i] == false, for some i, $0 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish [i] == false, then P is deadlocked.

This algorithm requires an order of m x n^2 operations to detect whether the system is in deadlocked state.

6. Recovery from Deadlock

Ans: There are two options for breaking a deadlock.

- One is simply to abort one or more processes to break the circular wait.
- Second one is to pre-empt some resources from one or more of the deadlocked processes.

To eliminate deadlocks by aborting a process, we use one of the two methods.

Process Termination

- Abort all deadlocked processes. This method clearly break the deadlock cycle, but at great expense
- Abort one process at a time until the deadlock cycle is eliminated: since after each process is aborted, a deadlock- detection algorithm must be invoked to determine whether any processes are still deadlocked
- In which order should we choose to abort?
    1. Priority of the process.
    2. How long process has computed, and how much longer to completion.
    3. Resources the process has used.
    4. Resources process needs to complete.
    5. How many processes will need to be terminated?
    6. Is process interactive or batch?

Resource Preemption

Successively pre-empt some resources and give them to other processes until the deadlock cycle is broken.

If pre-emption is required to deal with deadlocks, then three issues need to be addressed.

- **selecting a victim:** which resources and which processes are to be pre-empted

1. How to minimize the cost?
2. what types and how many resources are held
3. how much time has run
4. how much time to end

- **Rollback**: if we pre-empt a resource from a process, what should be done with that process

  1. rollback a process to a safe state, and restart it (some resources are released)
  2. Two implementations –
  3. totally rollback: simple but expensive as far as necessary – (OS should maintain "checkpoints ")
  4. checkpoint: a recording of the state of a process to allow rollback.

- **Starvation**

  Not always select the same process for pre-emption

8. What is address binding. Explain Logical versus and physical address space?

Ans: After the instruction has been executed on the operands, results may be stored back into the memory. The memory unit sees only the stream of memory addresses; it does not known how they are generated and what they are for.

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.

Registers that are built into the CPU are generally accessible within one cycle of the CPU block.

Each process has a separate memory space and range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. To ensure this, protection is provided by using two registers called base and limit register.

The base register holds the smallest legal physical memory address and limit register specifies the size of the range. The base and limit registers can be loaded only by the operating system.

Address Binding

Program resides on a memory as an executable file, and that program must be brought into memory and placed within a process. The processes on the disk that are waiting to be brought into memory for execution from the input queue. After the successful execution of process, it terminates and its memory space is declared available.

Binding of instructions and data to memory addresses can be done at any step:

- **Compile Time:** If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load Time**: Must generate relocatable code if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).

Logical Versus Physical Address Space

- Logical address – generated by the CPU; also referred to as virtual address
- Physical address – address seen by the memory unit, the one loaded into the memory address register of the memory

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- The set of all logical addresses generated by a program is a logical address space and the set of all physical addresses corresponding to these physical addresses is a physical address space
- The run- time mapping from virtual to physical addresses is done by a hardware device called memory- management unit(MMU).

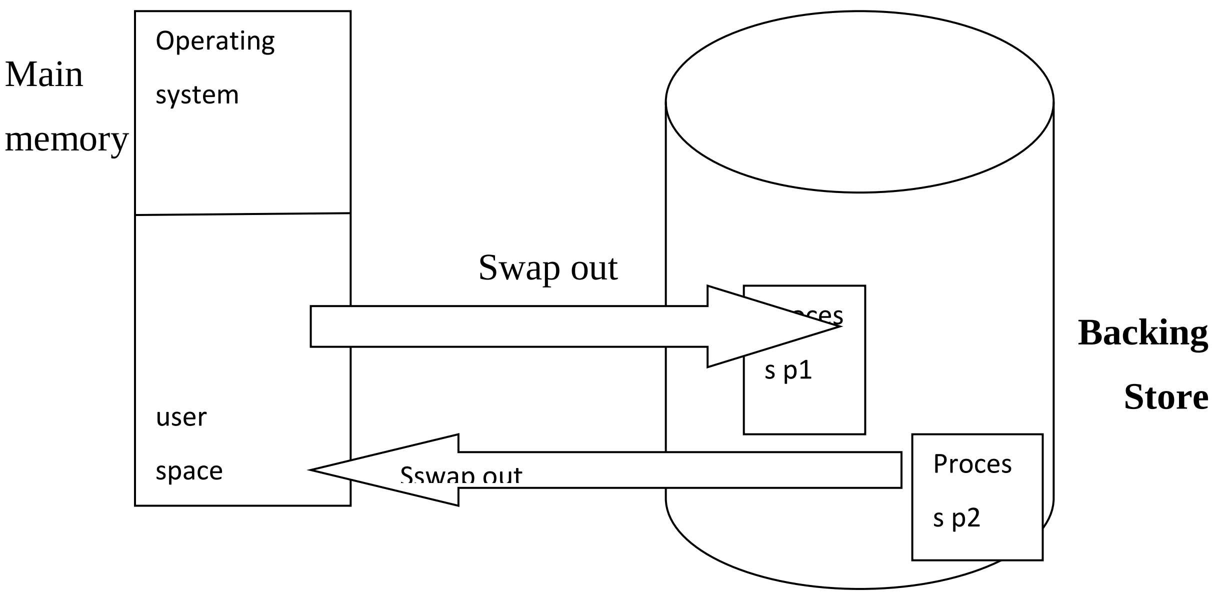
9. What is swapping? Explain in detail.

Ans: A process must be in memory to be executed and can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution

Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

A variant of this swapping policy is used for priority based scheduling algorithms. If a higher priority process arrives and wants service the memory manager can swap out the lower priority process and then load and execute the higher- priority process. This type of swapping called Roll out and Roll in

Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed



- The process that is swapped out will be swapped in to the same memory by the method of **address binding**
- If binding is done at **assembly or load time**, then the process cannot be easily moved to a different location
- In **execution time** swapped process can be moved into different memory space.
- The backing store accommodate copies of all memory images for all users and provides direct access to these memory images
- The system maintains a **ready queue** consisting of all processes whose memory images on the backing store or in memory and are ready to run.
- The dispatcher checks to see whether the next process in the queue is in memory, if it is not the dispatcher swaps out a process currently in memory and swaps in the desired process.
- The context- switch time is high
- The total transfer time is directly proportional to the amount of memory swapped.
- There are other constraints for swapping a process.
- To swap a process it should be completely idle.

The two main solutions for this problem is never swap a process with Pending I/O , or execute I/O operations only into operating system buffers.