UNIT – IV

CONCURRENCY

Critical Section:

In simple terms a critical section is group of instructions/statements or region of code that need to be executed atomically (read this post for atomicity), such as accessing a resource (file, input or output port, global data, etc.).

In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e. data race across threads), the result is unpredictable.

The access to such shared variable (shared memory, shared files, shared port, etc…) to be synchronized. Few programming languages have built in support for synchronization.

It is critical to understand the importance of race condition while writing kernel mode programming (a device driver, kernel thread, etc.). since the programmer can directly access and modifying kernel data structures.

A simple solution to critical section can be thought as shown below,

```
acquireLock();
Process Critical Section
releaseLock();
```

A thread must acquire a lock prior to executing critical section. The lock can be acquired by only one thread. There are various ways to implement locks in the above pseudo code. Let us discuss them in future articles.

Operating System | Process Synchronization | Introduction

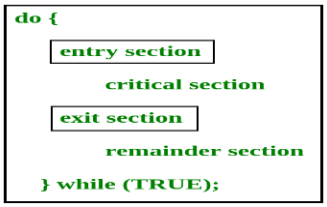
On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Critical Section Problem Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency

of datavariables.



In the entry section, the process requests for entry in the **Critical Section.**

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

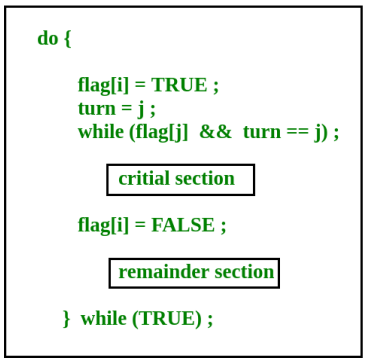
- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is in the critical section, then no other process from outside can block it from entering the critical section.
- **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson's Solution

Peterson's Solution is a classical software based solution to the critical section problem.

In Peterson's solution, we have two shared variables:

- boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
- int turn : The process whose turn is to enter the critical section.



Peterson's Solution preserves all three conditions :

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not blocks other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's Solution

- It involves Busy waiting
- It is limited to 2 processes.

TestAndSet

TestAndSet is a hardware solution to the synchronization problem. In TestAndSet, we have a shared lock variable which can take either of the two values, 0 or 1.

0 Unlock

1 Lock

Before entering into the critical section, a process inquires about the lock. If it is locked, it keeps on waiting till it become free and if it is not locked, it takes the lock and executes the critical section.

In TestAndSet, Mutual exclusion and progress are preserved but bounded waiting cannot be preserved.

Semaphores

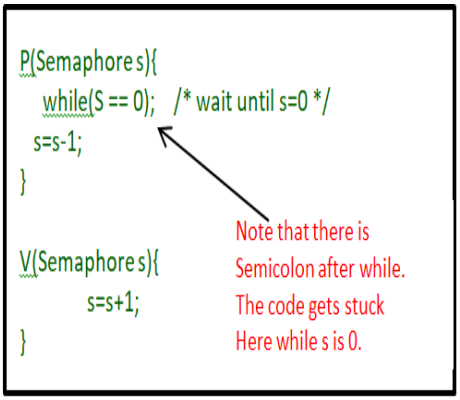
A Semaphore is an integer variable, which can be accessed only through two operations *wait()* and *signal()*. There are two types of semaphores : Binary Semaphores and Counting Semaphores

- Binary Semaphores : They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.
- Counting Semaphores : They can have any value and are not restricted over a certain domain. They can be used to control access a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

Operating System | Semaphores in operating system

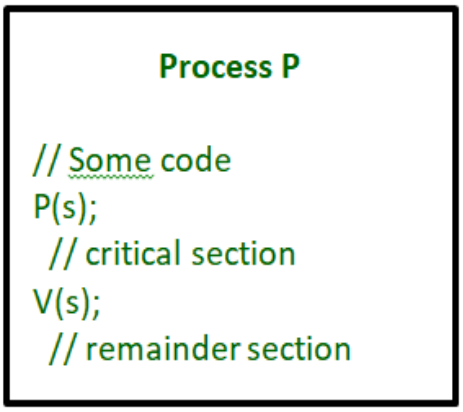
Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multiprocessing environment.The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only. Now let us see how it do so.

First look at two operations which can be used to access and change the value of semaphore variable.

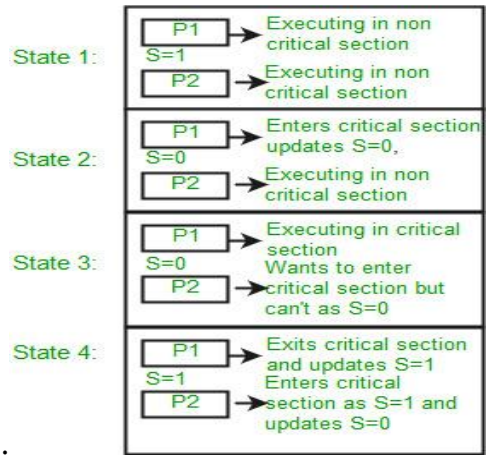


Some point regarding P and V operation

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.
3. A critical section is surrounded by both operations to implement process synchronization.See below image.critical section of Process P is in between P and V operation.



Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details.



The description above is for binary semaphore which can take only two values 0 and 1. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instance is 4. Now we initialize S = 4 and rest is same as for binary semaphore. Whenever process wants that resource it calls P or wait function and when it is done it calls V or signal function. If value of S becomes zero than a process has to wait until S becomes positive. For example, Suppose there are 4 process P1, P2, P3, P4 and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four process calls signal function and value of semaphore becomes positive.

Problem in this implementation of semaphore

Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle. To avoid this another implementation is provided below.

```
P(Semaphore s)
{
   s = s - 1;
   if (s < 0) {

      // add process to queue
      block();
   }
}

V(Semaphore s)
{
   s = s + 1;
   if (s >= 0) {

      // remove process p from queue
      wakeup(p);
   }
}
```

In this implementation whenever process waits it is added to a waiting queue of processes associated with that semaphore. This is done through system call block() on that process. When a process is completed it calls signal function and one process in the queue is resumed. It uses wakeup() system call.

Mutex vs Semaphore

What are the differences between Mutex vs Semaphore? When to use mutex and when to use semaphore? Concrete understanding of Operating System concepts is required to design/develop smart applications. Our objective is to educate the reader on these concepts and learn from other expert geeks.

As per operating system terminology, mutex and semaphore are kernel resources that provide synchronization services (also called as *synchronization primitives*). *Why do we need such synchronization primitives? Won't be only one sufficient?* To answer these questions, we need to understand few keywords. Please read the posts on atomicity and critical section. We will illustrate with examples to understand these concepts well, rather than following usual OS textual description.

The producer-consumer problem:

Note that the content is generalized explanation. Practical details vary with implementation.

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096 byte length. A producer thread collects the data and writes it to the buffer. A consumer thread processes the collected data from the buffer. Objective is, both the threads should not run at the same time.

Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception:

There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is binary semaphore. *But they are not*! The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore.

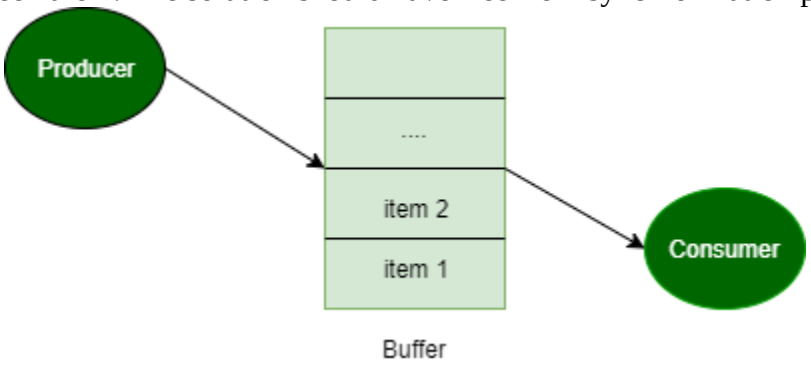
Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

Operating System | Peterson's Algorithm (Using processes and shared memory)

Prerequisite – synchronization, Critical Section

Problem: The producer consumer problem (or bounded buffer problem) describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. Producer produce an item and put it into buffer. If buffer is already full then producer will have to wait for an empty block in buffer. Consumer consume an item from buffer. If buffer is already empty then consumer will have to wait for an item in buffer. Implement Peterson's Algorithm for the two processes using shared memory such that there is mutual exclusion between them. The solution should have free from synchronization problems.



Peterson's algorithm –

```
// code for producer (j)

// producer j is ready
// to produce an item
flag[j] = true;

// but consumer (i) can consume an item
turn = i;

// if consumer is ready to consume an item
// and if its consumer's turn
while (flag[i] == true && turn == i)

    { // then producer will wait }

    // otherwise producer will produce
    // an item and put it into buffer (critical Section)

    // Now, producer is out of critical section
    flag[j] = false;
    // end of code for producer

    //--------------------------------------------------------
    // code for consumer i

    // consumer i is ready
    // to consume an item
    flag[i] = true;

    // but producer (j) can produce an item
    turn = j;

    // if producer is ready to produce an item
    // and if its producer's turn
    while (flag[j] == true && turn == j)
```

```
    { // then consumer will wait }

    // otherwise consumer will consume
    // an item from buffer (critical Section)

    // Now, consumer is out of critical section
    flag[i] = false;
// end of code for consumer
```

Explanation of Peterson's algorithm –

Peterson's Algorithm is used to synchronize two processes. It uses two variables, a bool array **flag** of size 2 and an int variable **turn** to accomplish it. In the solution i represents the Consumer and j represents the Producer. Initially the flags are false. When a process wants to execute it's critical section, it sets it's flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it's own critical section. After this the current process enters it's critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it's own flag to false, indication it does not wish to execute anymore. The program runs for a fixed amount of time before exiting. This time can be changed by changing value of the macro RT.

Readers-Writers Problem | Set 1 (Introduction and Readers Preference Solution)

Consider a situation where we have a file shared between many people.

- If one of the people tries editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible to him/her.
- However if some person is reading the file, then others may read it at the same time.

Precisely in OS we call this situation as the **readers-writers problem**

Problem parameters:

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write
- Readers may not write and only read

Solution when Reader has the Priority over Writer

Here priority means, no reader should wait if the share is currently opened for reading.

Three variables are used: **mutex, wrt, readcnt** to implement solution

1. **semaphore** mutex, wrt; // semaphore **mutex** is used to ensure mutual exclusion when **readcnt** is updated i.e. when any reader enters or exit from the critical section and semaphore **wrt** is used by both readers and writers
2. **int** readcnt; // **readcnt** tells the number of processes performing read in the critical section, initially 0

Functions for sempahore :

– wait() : decrements the semaphore value.

– signal() : increments the semaphore value.

Writer process:

1. Writer requests the entry to critical section.
2. If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
3. It exits the critical section.

```
do {
   // writer requests for critical section
   wait(wrt);

   // performs the write

   // leaves the critical section
   signal(wrt);

} while(true);
```

Reader process:

1. Reader requests the entry to critical section.
2. If allowed:
    - it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.
    - It then, signals mutex as any other reader is allowed to enter while others are already reading.

- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.

3. If not allowed, it keeps on waiting.

```
do {


  // Reader wants to enter the critical section
  wait(mutex);


  // The number of readers has now increased by 1
  readcnt++;


  // there is atleast one reader in the critical section
  // this ensure no writer can enter if there is even one reader
  // thus we give preference to readers here
  if (readcnt==1)
    wait(wrt);

  // other readers can enter while this current reader is inside
  // the critical section
  signal(mutex);

  // current reader performs reading here
  wait(mutex);   // a reader wants to leave

  readcnt--;

  // that is, no reader is left in the critical section,
  if (readcnt == 0)
     signal(wrt);         // writers can enter

  signal(mutex); // reader leaves

} while(true);
```

Thus, the mutex ‘**wrt**‘ is queued on both readers and writers in a manner such that preference is given to readers if writers are also there. Thus, no reader is waiting simply because a writer has requested to enter the critical section.

Operating System | Reader-Writers solution using Monitors

Prerequisite – Process Synchronization, Monitors, Readers-Writers Problem

Considering a shared Database our objectives are:

- Readers can access database only when there are no writers.
- Writers can access database only when there are no readers or writers.
- Only one thread can manipulate the state variables at a time.

Basic structure of a solution –

```
Reader()
  Wait until no writers
  Access database
  Check out – wake up a waiting writer
Writer()
  Wait until no active readers or writers
  Access database
  Check out – wake up waiting readers or writer
```

- **Once a reader is waiting, readers will get in next.**
- **If a writer is waiting, one writer will get in next.**

Implementation of the solution using monitors:-

1. The methods should be executed with mutual exclusion i.e. At each point in time, at most one thread may be executing any of its methods.
2. Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task.
3. Monitors also have a mechanism for signaling other threads that such conditions have been met.
4. So in this implementation only mutual exclusion is not enough. Threads attempting an operation may need to wait until some assertion P holds true.
5. While a thread is waiting upon a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state.

Producer Consumer Problem using Semaphores | Set 1

Prerequisite – Semaphores in operating system, Inter Process Communication

Producer consumer problem is a classical synchronization problem. We can solve this problem by using semaphores.

A **semaphore** S is an integer variable that can be accessed only through two standard operations : wait() and signal().

The wait() operation reduces the value of semaphore by 1 and the signal() operation increases its value by 1.

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

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

Semaphores are of two types:

1. **Binary Semaphore –** This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement solution of critical section problem with multiple processes.

2. **Counting Semaphore –** Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Problem Statement – We have a buffer of fixed size. A producer can produce an item and can place in the buffer. A consumer can pick items and can consume them. We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item. In this problem, buffer is the critical section.

To solve this problem, we need two counting semaphores – Full and Empty. "Full" keeps track of number of items in the buffer at any given time and "Empty" keeps track of number of unoccupied slots.

Initialization of semaphores –

mutex = 1

Full = 0 // Initially, all slots are empty. Thus full slots are 0

Empty = n // All slots are empty initially

Solution for Producer –

```
do{

//produce an item

wait(empty);
wait(mutex);

//place in buffer

signal(mutex);
signal(full);

}while(true)
```

When producer produces an item then the value of "empty" is reduced by 1 because one slot will be filled now. The value of mutex is also reduced to prevent consumer to access the buffer. Now, the producer has placed the item and thus the value of "full" is increased by 1. The value of mutex is also increased by 1 beacuse the task of producer has been completed and consumer can access the buffer.

Solution for Consumer –

```
do{

wait(full);
wait(mutex);
```

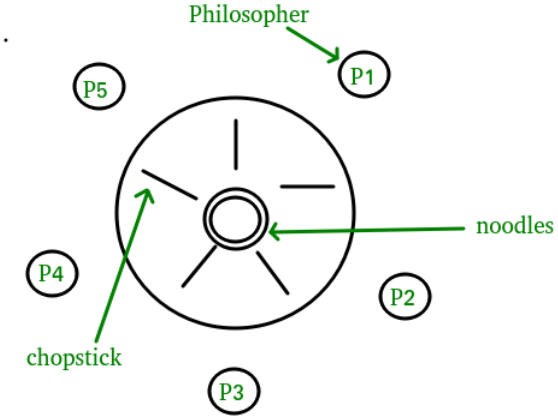
```
// remove item from buffer

signal(mutex);
signal(empty);

// consumes item

}while(true)
```

As the consumer is removing an item from buffer, therefore the value of “full” is reduced by 1 and the value is mutex is also reduced so that the producer cannot access the buffer at this moment. Now, the consumer has consumed the item, thus increasing the value of “empty” by 1. The value of mutex is also increased so that producer can access the buffer now.

Operating System | Dining-Philosophers Solution Using Monitors

Prerequisite: Monitor, Process Synchronization

Dining-Philosophers Problem – N philosophers seated around a circular table



- There is one chopstick between each philosopher
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once

We need an algorithm for allocating these limited resources(chopsticks) among several processes(philosophers) such that solution is free from deadlock and free from starvation.

There exist some algorithm to solve Dining – Philosopher Problem, but they may have deadlock situation. Also, a deadlock-free solution is not necessarily starvation-free. Semaphores can result in deadlock due to programming errors. Monitors alone are not sufficiency to solve this, we need monitors with *condition variables*

Monitor-based Solution to Dining Philosophers

We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. Monitor is used to control access to state variables and condition variables. It only tells when to enter and exit the segment. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

THINKING – When philosopher doesn't want to gain access to either fork.
HUNGRY – When philosopher wants to enter the critical section.
EATING – When philosopher has got both the forks, i.e., he has entered the section.
Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).
Above Program is a monitor solution to the dining-philosopher problem.

We also need to declare

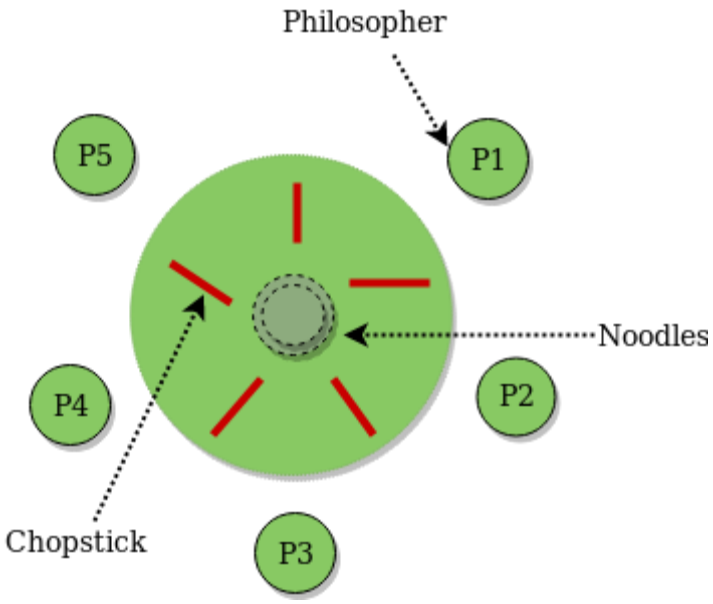
```
condition self[5];
```

This allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs. We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
```

Dining Philosopher Problem Using Semaphores

Prerequisite – Process Synchronization, Semaphores, Dining-Philosophers Solution Using Monitors
The Dining Philosopher Problem – The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Semaphore Solution to Dining Philosopher –

Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
   {  THINK;

      PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);


   }
```

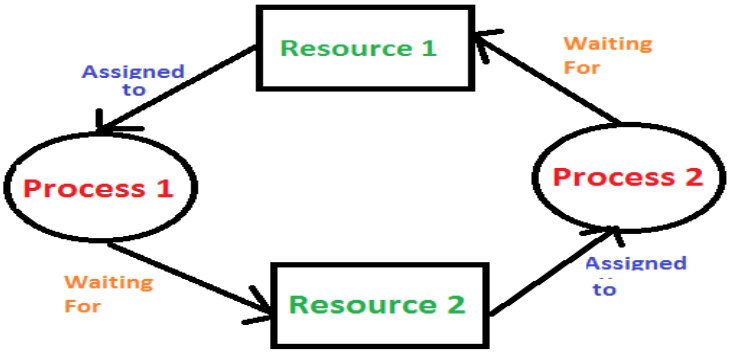
There are three states of philosopher : **THINKING, HUNGRY and EATING**. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

Operating System | Process Management | Deadlock Introduction

A process in operating systems uses different resources and uses resources in following way. 1)Requestsaresource2)Usetheresource2) Releases the resource

PRINCIPLES OF DEADLOCK

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)
Mutual Exclusion: One or more than one resource are non-sharable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.
No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.

Methods for handling deadlock There are three ways to handle deadlock

1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

Deadlock Prevention And Avoidance

Deadlock Characteristics As discussed in the previous post, deadlock has following characteristics.

Mutual Exclusion.

Hold and Wait.

No preemption.

Circular wait.

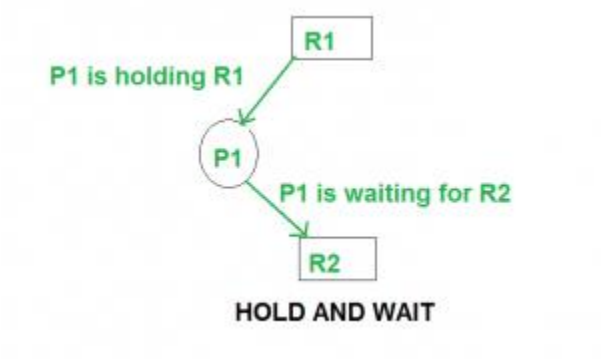
Deadlock Prevention

We can prevent Deadlock by eliminating any of the above four condition.

EliminateMutualExclusion It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tap drive and printer, are inherently non-shareable.

EliminateHoldandwait1. Allocate all required resources to the process before start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remained blocked till it has completed its execution.

2. Process will make new request for resources after releasing the current set of resources. This solution may lead to starvation.



EliminateNoPreemption

Pre-empt resources from process when resources required by other high priority process.

EliminateCircularWait

Each resource will be assigned with a numerical number. A process can request for the resources only in increasing order of numbering. For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

Deadlock Avoidance

Deadlock avoidance can be done with Banker's Algorithm.

Banker's Algorithm

Bankers's Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it check for safe state, if after granting request system remains in the safe state it allows the request and if their is no safe state it don't allow the request made by the process.

InputstoBanker'sAlgorithm

1.Maxneedofresourcesbyeachprocess.

2.Currentlyallocatedresourcesbyeachprocess.

3. Max free available resources in the system. Request will only be granted under below condition.

1. If request made by process is less than equal to max need to that process.

2. If request made by process is less than equal to freely available resource in the system.

Example

Total resources in system:

A B C D

6 5 7 6

Available system resources are:

A B C D

3 1 1 2

Processes (currently allocated resources):

A B C D

P1 1 2 2 1

P2 1 0 3 3

P3 1 2 1 0

Processes (maximum resources):

A B C D

P1 3 3 2 2

P2 1 2 3 4

P3 1 3 5 0

Need = maximum resources - currently allocated resources.

Processes (need resources):

A B C D

P1 2 1 0 1

P2 0 2 0 1

P3 0 1 4 0

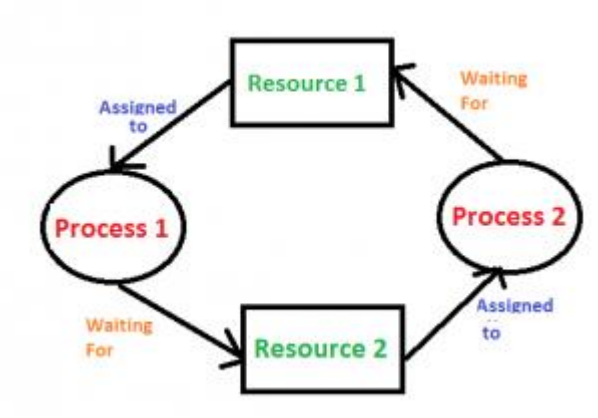
Deadlock Detection And Recovery

In the previous post, we have discussed Deadlock Prevention and Avoidance. In this post, Deadlock Detection and Recovery technique to handle deadlock is discussed.

Deadlock Detection

1. Ifresourceshavesingleinstance:In this case for Deadlock detection we can run an algorithm to check for cycle in the Resource Allocation Graph. Presence of cycle in the graph is the sufficient condition for

deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1–>P1–>R2–>P2. So Deadlock is Confirmed.

2.Iftherearemultipleinstancesofresources-Detection of cycle is necessary but not sufficient condition for deadlock detection, in this case system may or may not be in deadlock varies according to different situations.

DeadlockRecovery

Traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real time operating systems use Deadlock recovery.

Recovery method

1. Killing the process.

killing all the process involved in deadlock.

Killing process one by one. After killing each process check for deadlock again keep repeating process till system recover from deadlock.

2.ResourcePreemption

Resources are preempted from the processes involved in deadlock, preempted resources are allocated to other processes, so that their is a possibility of recovering the system from deadlock. In this case system go into starvation.

Operating System | Resource Allocation Graph (RAG)

As Banker's algorithm using some kind of table like allocation, request, available all that thing to understand what is the state of the system. Similarly, if you want to understand the state of the system instead of using those table, actually tables are very easy to represent and understand it, but then still you could even represent the same information in the graph. That graph is called **Resource Allocation Graph (RAG)**.

So, resource allocation graph is explained to us what is the state of the system in terms of **processes and resources**. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG, but then you might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and

Graph is better if the system contains less number of process and resource. We know that any graph contains vertices and edges. So RAG also contains vertices and edges. In RAG vertices are two type –

1. Process vertex – Every process will be represented as a process vertex. Generally, the process will be represented with a circle.

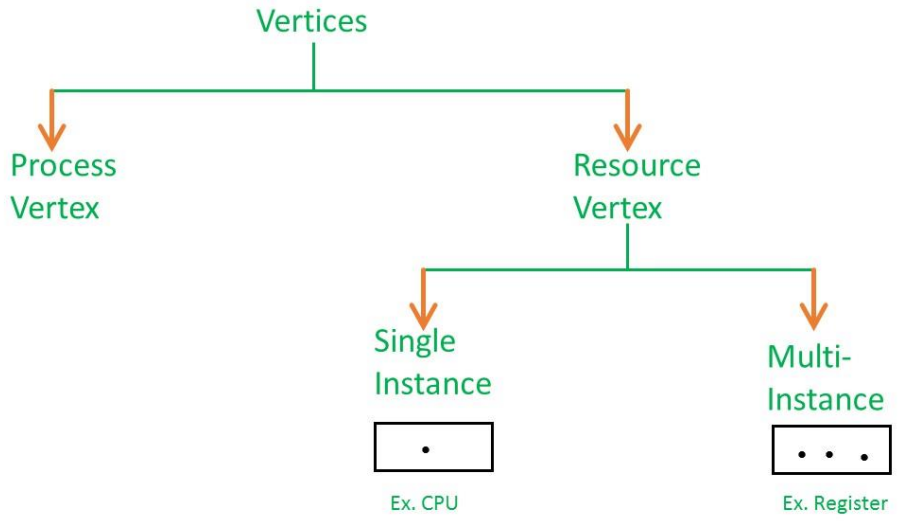
2. Resource vertex – Every resource will be represented as a resource vertex. It is also two type –

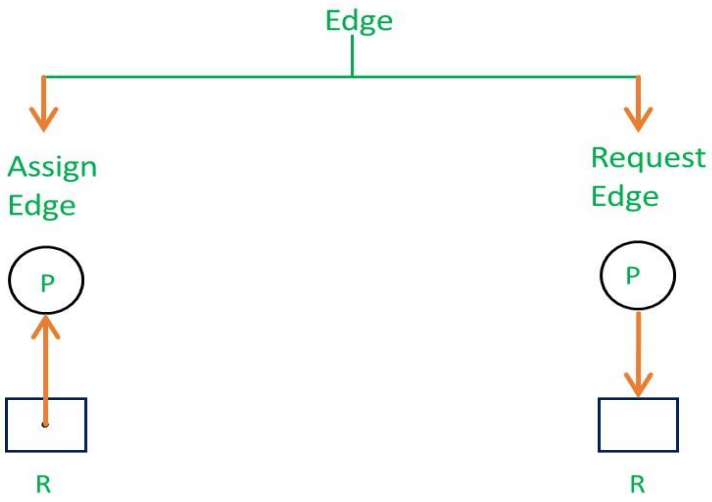
- **Single instance type resource –** It represents as a box, inside the box, there will be one do .So the number of dots indicate how many instances are present of each resource type.
- **Multi-resource instance type resource –** It also represents as a box, inside the box, there will be many dots present.

Now coming to the edges of RAG.There are two types of edges in RAG –

1. Assign Edge – If you already assign a resource to a process then it is called Assign edge.

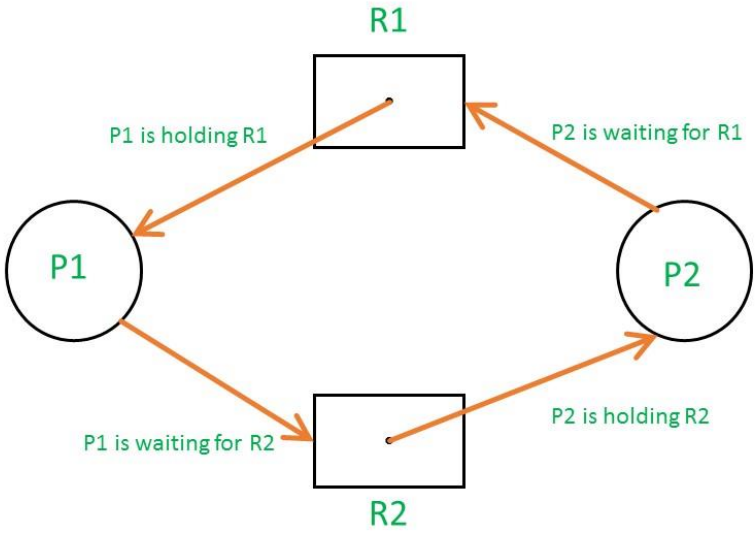
2. Request Edge – It means in future the process might want some resource to complete the execution, that is called request edge.





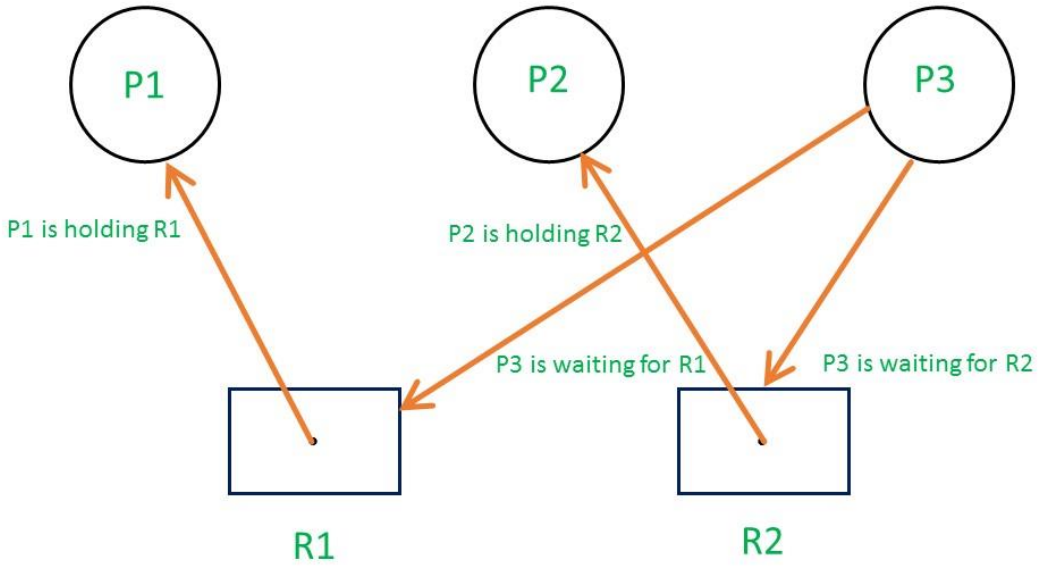
So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG) –



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

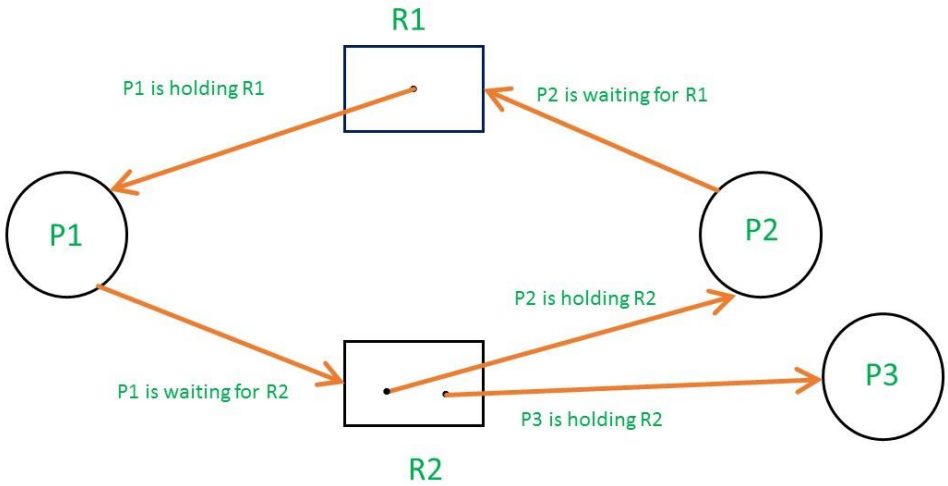


SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency.

So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG) –



MULTI INSTANCES WITHOUT DEADLOCK

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state.So to see the state of this RAG, let's construct the allocation matrix and request matrix.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	0	0

- The total number of processes are three; P1, P2 & P3 and the total number of resources are two; R1 & R2.

Allocation matrix –

- For constructing the allocation matrix, just go to the resources and see to which process it is allocated.
- R1 is allocated to P1, therefore write 1 in allocation matrix and similarly, R2 is allocated to P2 as well as P3 and for the remaining element just write 0.

Request matrix –

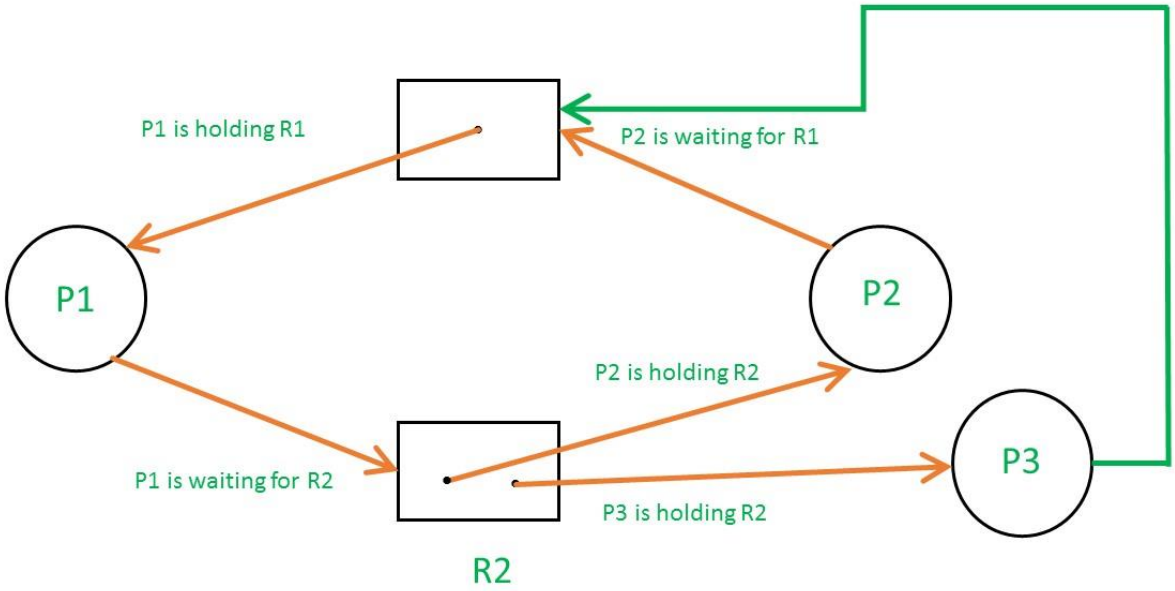
- In order to find out the request matrix, you have to go to the process and see the outgoing edges.
- P1 is requesting resource R2, so write 1 in the matrix and similarly, P2 requesting R1 and for the remaining element write 0.

So now available resource is = (0, 0).

Checking deadlock (safe or not) –

```
Available = 0  0   (As P3 does not require any extra resource to complete the execution and after
  P3        0  1    completion P3 release its own resource)

_____________________

New Available = 0  1  (As using new available resource we can satisfy the requirment of process P1
     P1         1  0   and P1 also release its prevous resource)

_____________________

New Available = 1  1 (Now easily we can satisfy the requirement of process P2)
     P2         0  1

_____________________

New Available =  1  2
```

So, there is no deadlock in this RAG.Even though there is a cycle, still there is no deadlock.Therefore in multi-instance resource cycle is not sufficient condition for deadlock.



MULTI INSTANCES WITH DEADLOCK

Above example is the same as the previous example except that, the process P3 requesting for resource R1. So the table becomes as shown in below.

Process	Allocation		Request	
	Resource		Resource	
	R1	R2	R1	R2
P1	1	0	0	1
P2	0	1	1	0
P3	0	1	1	0

So,the Available resource is = (0, 0), but requirement are (0, 1), (1, 0) and (1, 0).So you can’t fulfill any one requirement.Therefore, it is in deadlock.

Therefore, every cycle in a multi-instance resource type graph is not a deadlock, if there has to be a deadlock, there has to be a cycle.So, in case of RAG with multi-instance resource type, the cycle is a necessary condition for deadlock, but not sufficient.

Operating System | Deadlock detection algorithm

If a system does not employ either a deadlock prevention or deadlock avoidance algorithm then a deadlock situation may occur. In this case-

- Apply an algorithm to examine state of system to determine whether deadlock has has occurred or not.
- Apply an algorithm to recover from the deadlock. For more refer- Deadlock Recovery

Deadlock Detection Algorithm:

The algorithm employs several time varying data structures:

- **Available-** A vector of length m indicates the number of available resources of each type.
- **Allocation-** An n*m matrix defines the number of resources of each type currently allocated to a process. Column represents resource and resource represent process.
- **Request-** An n*m matrix indicates the current request of each process. If request[i][j] equals k then process P_i is requesting k more instances of resource type R_j.

We treat rows in the matrices Allocation and Request as vectors, we refer them as $Allocation_i$ and $Request_i$.

Steps of Algorithm:

1. Let *Work* and *Finish* be vectors of length m and n respectively. Initialize *Work= Available*. For *i=0, 1, …., n-1*, if $Allocation_i$ = 0, then *Finish[i]* = true; otherwise, *Finish[i]*= false.
2. Findanindexisuchthatboth
   a)*Finish[i]==false*
   b)$Request_i <= Work$
   If no such *i* exists go to step 4.
3. $Work = Work + Allocation_i$
   Finish[i]=true
   Go to Step 2.
4. If *Finish[i]== false* for some i, 0<=i<n, then the system is in a deadlocked state. Moreover, if *Finish[i]==false* the process P_i is deadlocked.

For example,

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

1. In this, Work = [0, 0, 0] & Finish = [false, false, false, false, false]
2. *i=0* is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
3. Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] & Finish = [true, false, false, false, false].
4. *i=2* is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
5. Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] & Finish = [true, false, true, false, false].
6. *i=1* is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
7. Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] & Finish = [true, true, true, false, false].
8. *i=3* is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
9. Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] & Finish = [true, true, true, true, false].

10. *i=3* is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].
11. Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] & Finish = [true, true, true, true, true].
12. Since Finish is a vector of all true it means **there is no deadlock** in this example.

Program for Banker's Algorithm | Set 1 (Safety Algorithm)

Prerequisite: Banker's Algorithm

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Following **Data structures** are used to implement the Banker's Algorithm:

Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently allocated **'k'** instances of resource type $\mathbf{R_j}$
- Need [i, j] = Max [i, j] – Allocation [i, j]

$Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.

Banker's algorithm consist of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length 'm' and 'n' respectively.
   Initialize: Work= Available
   Finish [i]=false; for i=1,2,……,n
2. Findanisuchthatboth
   a)Finish[i]=false
   b)Need_i<=work

   if no such i exists goto step (4)
3. Work=Work+Allocation_i
   Finish[i]=true go to step(2)
4. IfFinish[i]=true for all i, then the system is in safe state.

Safe sequence is the sequence in which the processes can be safely executed.

UNIT - V

File Systems | Operating System

A file is a collection of related information that is recorded on secondary storage. Or file is a collection of logically related entities. From user's perspective a file is the smallest allotment of logical secondary storage.

ATTRIBUTES	TYPES	OPERATIONS
Name	Doc	Create
Type	Exe	Open
Size	Jpg	Read
Creation Data	Xis	Write
Author	C	Append
Last Modified	Java	Truncate
protection	class	Delete
		Close

FILE TYPE	USUAL EXTENSION	FUNCTION
Executable	exe, com, bin	Read to run machine language program

FILE TYPE	USUAL EXTENSION	FUNCTION
Object	obj, o	Compiled, machine language not linked
Source Code	C, java, pas, asm, a	Source code in various languages
Batch	bat, sh	Commands to the command interpreter
Text	txt, doc	Textual data, documents
Word Processor	wp, tex, rrf, doc	Various word processor formats
Archive	arc, zip, tar	Related files grouped into one compressed file
Multimedia	mpeg, mov, rm	For containing audio/video information

FILE DIRECTORIES:

Collection of files is a file directory. The directory contains information about the files, including attributes, location and ownership. Much of this information, especially that is concerned with storage, is managed by the operating system. The directory is itself a file, accessible by various file management routines.

Information contained in a device directory are:

- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed
- Date last updated
- Owner id
- Protection information

Operation performed on directory are:

- Search for a file
- Create a file
- Delete a file

- List a directory
- Rename a file
- Traverse the file system

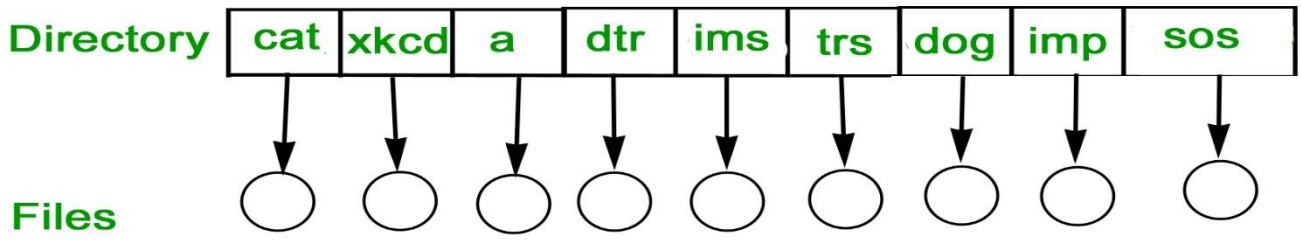
Advantages of maintaining directories are:

- **Efficiency:** A file can be located more quickly.
- **Naming:** It becomes convenient for users as two users can have same name for different files or may have different name for same file.
- **Grouping:** Logical grouping of files can be done by properties e.g. all java programs, all games etc.

SINGLE-LEVEL DIRECTORY

In this a single directory is maintained for all the users.

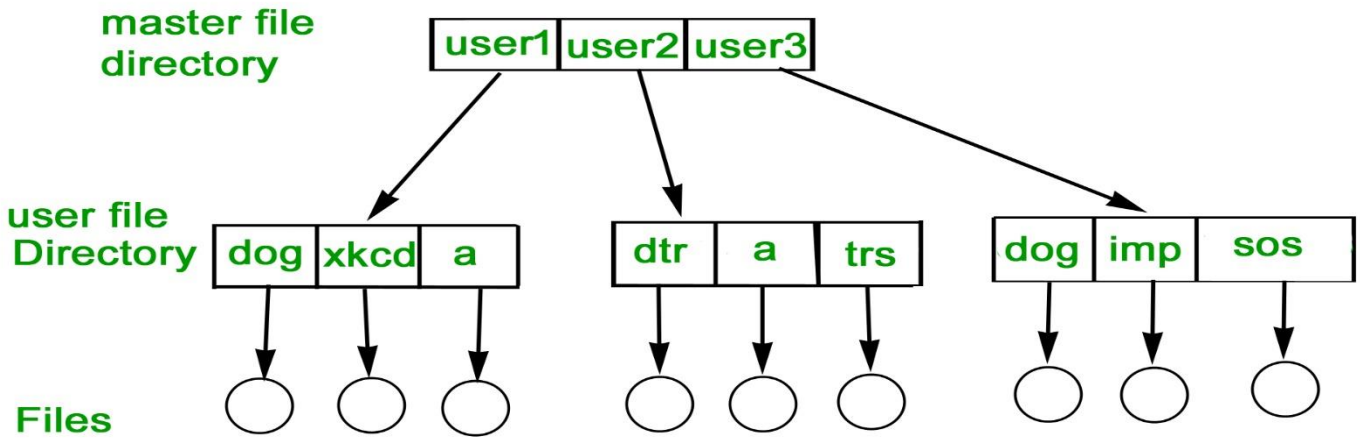
- **Naming problem:** Users cannot have same name for two files.
- **Grouping problem:** Users cannot group files according to their need.



TWO-LEVEL DIRECTORY

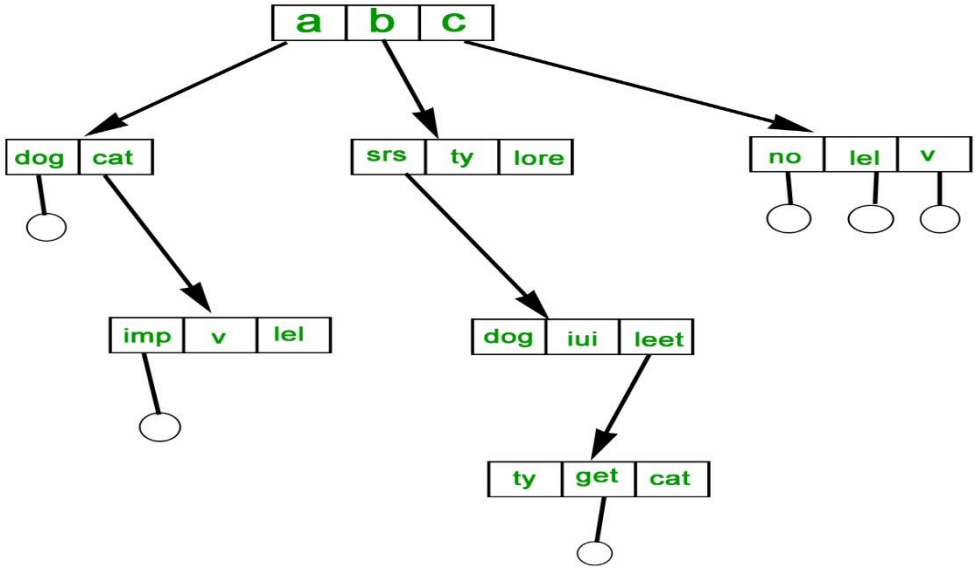
In this separate directories for each user is maintained.

- Path name:Due to two levels there is a path name for every file to locate that file.
- Now,we can have same file name for different user.
- Searching is efficient in this method.



TREE-STRUCTURED DIRECTORY :

Directory is maintained in the form of a tree. Searching is efficient and also there is grouping capability. We have absolute or relative path name for a file.



File Allocation Methods

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

The main idea behind these methods is to provide:

- Efficient disk space utilization.
- Fast access to the file blocks.

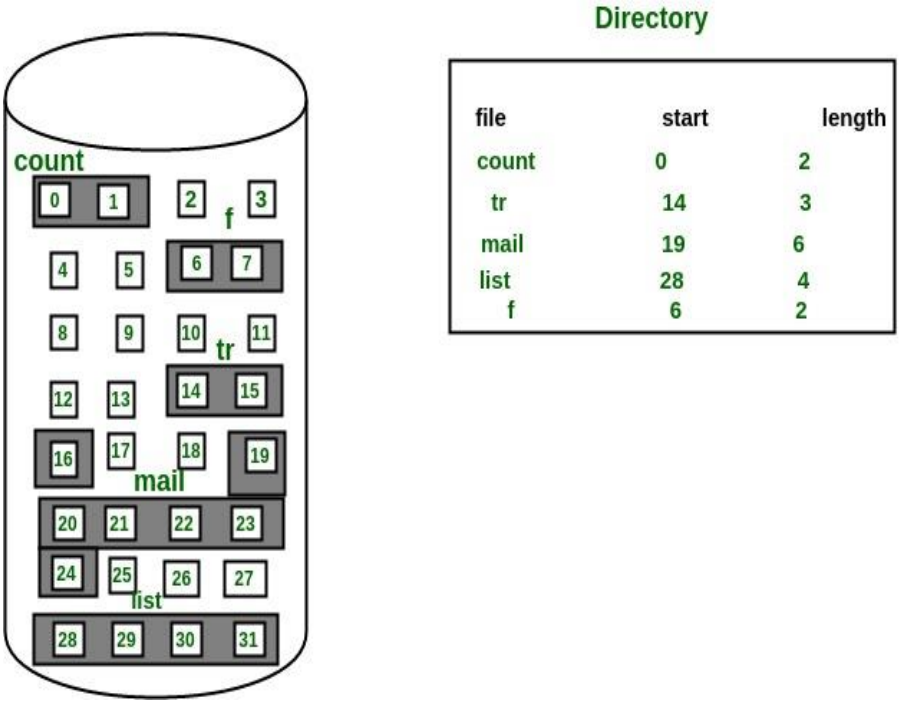
All the three methods have their own advantages and disadvantages as discussed below:

1. Contiguous Allocation

In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1.* This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.
The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.



Advantages:

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:

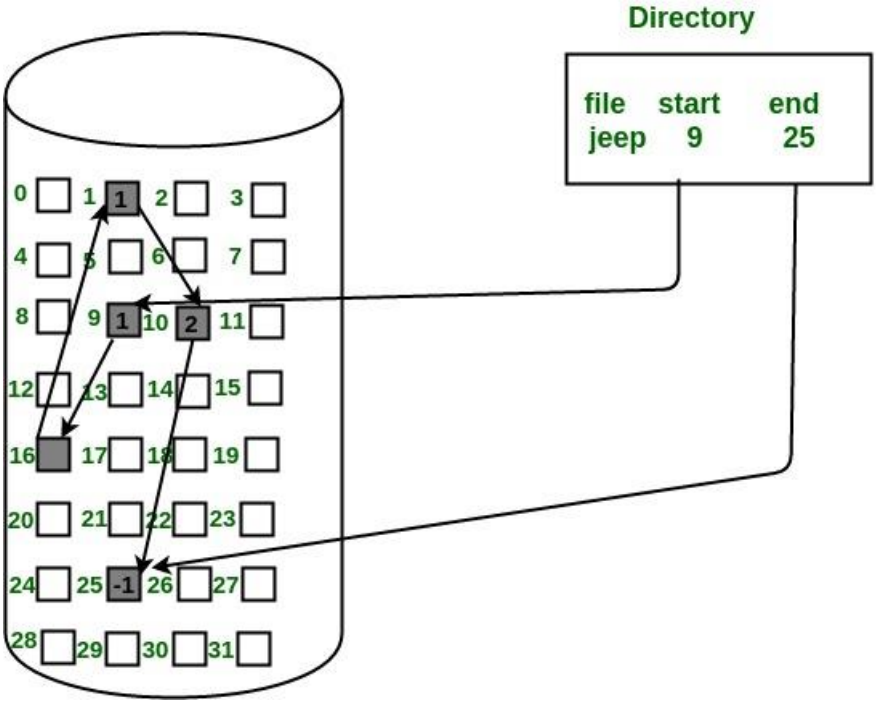
- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

2. Linked List Allocation

In this scheme, each file is a linked list of disk blocks which **need not be** contiguous. The disk blocks can be scattered anywhere on the disk.

The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.



Advantages:

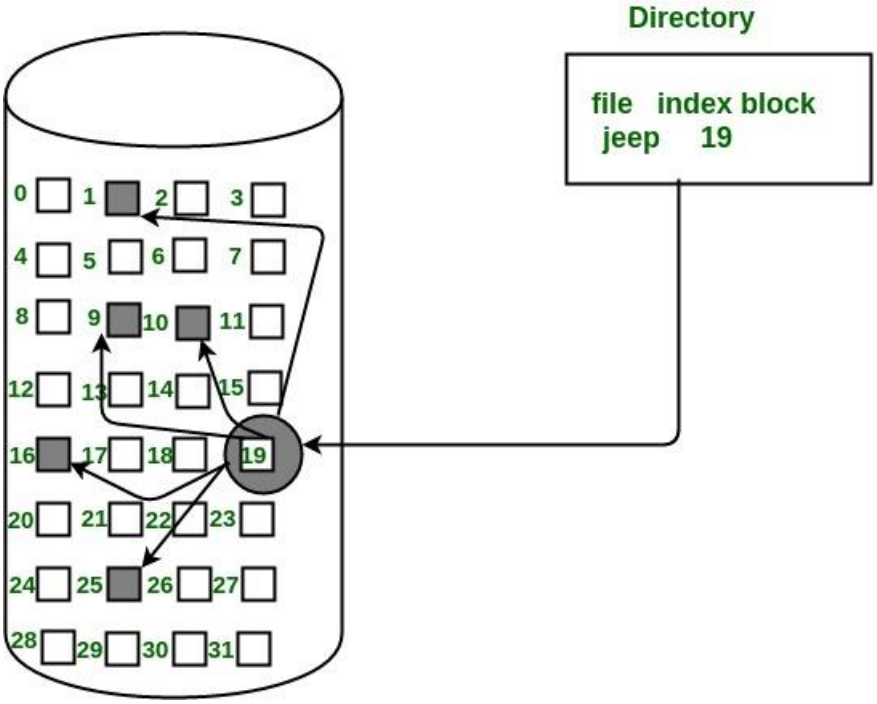
- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.
- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

Disadvantages:

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We can not directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

3. Indexed Allocation

In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directory entry contains the address of the index block as shown in the image:



Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

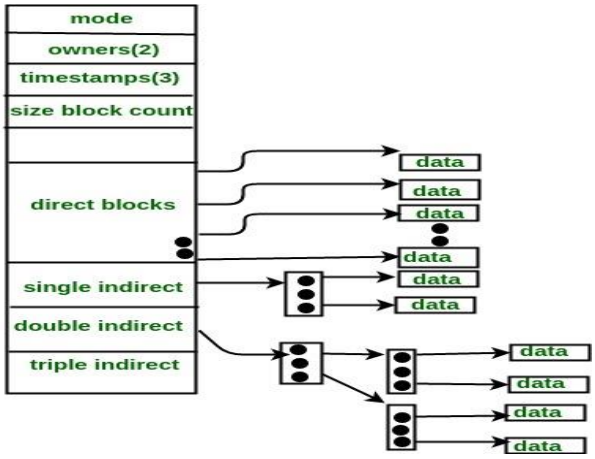
Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

For files that are very large, single index block may not be able to hold all the pointers. Following mechanisms can be used to resolve this:

1. **Linked scheme:** This scheme links two or more index blocks together for holding the pointers. Every index block would then contain a pointer or the address to the next index block.
2. **Multilevel index:** In this policy, a first level index block is used to point to the second level index blocks which inturn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.
3. **Combined Scheme:** In this scheme, a special block called the **Inode (information Node)** contains all the information about the file such as the name, size, authority, etc and the remaining space of Inode is used to store the Disk Block addresses which contain the actual file *as shown in the image below*. The first few of these pointers in Inode point to the **direct blocks** i.e the pointers contain the addresses of the disk blocks that contain data of the file. The next few pointers point to indirect blocks. Indirect blocks may be single indirect, double indirect or triple indirect. **Single Indirect block** is the disk block that does not contain the file data but the disk address of the blocks that contain the file data. Similarly,

double indirect blocks do not contain the file data but the disk address of the blocks that contain the.



Operating System | File Directory | Path Name

Prerequisite – File Systems

Hierarchical Directory Systems –

Directory is maintained in the form of a tree.Each user can have as many directories as are needed so, that files can be grouped together in natural way.

Advantages of this structure:

- Searching is efficient
- Groping capability of files increase

When the file system is organized as a directory tree, some way is needed for specifying file names.

Two different methods are commonly used:

1. **Absolute Path name –** In this method, each file is given an **absolute path** name consisting of the path from the root directory to the file. As an example, the path **/usr/ast/mailbox** means that the root directory contains a subdirectory usr, which in turn contains a subdirectory ast, which contains the file mailbox. Absolute path names always start at the root directory and are unique.

   In UNIX the components of the path are separated by /. In Windows the separator is \. **Windows** \usr\ast\mailbox

   UNIX /usr\ast\mailbox
2. **Relative Path name –** This is used in conjunction with the concept of the **working directory** (also called the **current directory**).A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For **example**, if the current working directory is /usr/ast, then the file whose absolute path is /usr/ast/mailbox can be referenced simply as mailbox.Inother words,the UNIX command

   cp/usr/ast/mailbox/usr/ast/mailbox.bak

   andthecommand-**cpmailboxmailbox.bak**

   do exactly the same thing if the working directory is /usr/ast.

When to use which approach? Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read /usr/lib/dictionary to do its work. It should use the full, absolute path name in this case

because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is.

Of course, if the spelling checker needs a large number of files from /usr/lib, an alternative approach is for it to issue a system call to change its working directory to /usr/lib, and then use just dictionary as the first parameter to open. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

DISK SCHEDULING ALGORITHMS

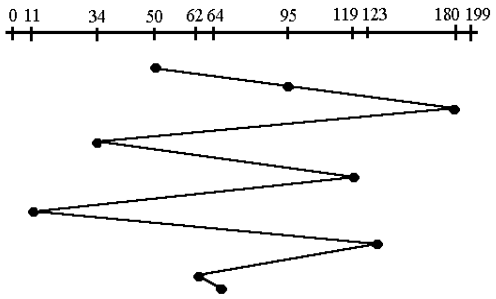
In operating systems, seek time is very important. Since all device requests are linked in queues, the seek time is increased causing the system to slow down. Disk Scheduling Algorithms are used to reduce the total seek time of any request.

The purpose of this material is to provide one with help on disk scheduling algorithms. Hopefully with this, one will be able to get a stronger grasp of what disk scheduling algorithms do.

TYPES OF DISK SCHEDULING ALGORITHMS

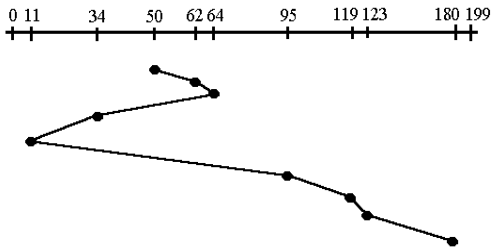
Although there are other algorithms that reduce the seek time of all requests, I will only concentrate on the following disk scheduling algorithms:
First Come-First Serve (FCFS)
Shortest Seek Time First (SSTF)
Elevator (SCAN)
Circular SCAN (C-SCAN)
LOOK
C-LOOK
These algorithms are not hard to understand, but they can confuse someone because they are so similar. What we are striving for by using these algorithms is keeping Head Movements (# tracks) to the least amount as possible. The less the head has to move the faster the seek time will be. I will show you and explain to you why C-LOOK is the best algorithm to use in trying to establish less seek time.

Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199 let us now discuss the different algorithms.

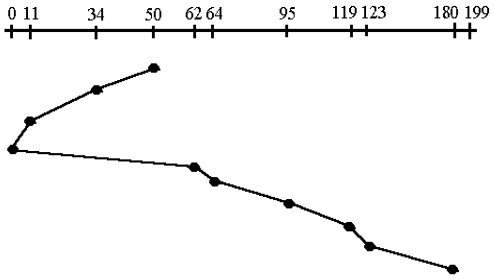


1. *First Come -First Serve (FCFS)* All incoming requests are placed at the end of the queue. Whatever number that is next in the queue will be the next number served. Using this algorithm doesn't provide the best results. To determine the number of head movements you would simply find the number of tracks it took to move from one request to the next. For this case it went from 50 to 95 to 180 and so on. From 50 to 95 it moved 45 tracks. If you tally up the total number of tracks you will find how many tracks it had to go through before finishing the entire request. In this example, it had a total head movement of 640 tracks. The disadvantage of this algorithm is noted by the oscillation from

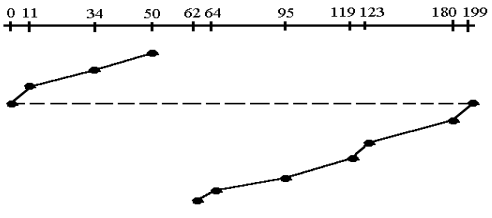
track 50 to track 180 and then back to track 11 to 123 then to 64. As you will soon see, this is the worse algorithm that one can use.



2. *Shortest Seek Time First (SSTF)*
In this case request is serviced according to next shortest distance. Starting at 50, the next shortest distance would be 62 instead of 34 since it is only 12 tracks away from 62 and 16 tracks away from 34. The process would continue until all the process are taken care of. For example the next case would be to move from 62 to 64 instead of 34 since there are only 2 tracks between them and not 18 if it were to go the other way. Although this seems to be a better service being that it moved a total of 236 tracks, this is not an optimal one. There is a great chance that starvation would take place. The reason for this is if there were a lot of requests close to eachother the other requests will never be handled since the distance will always be greater.

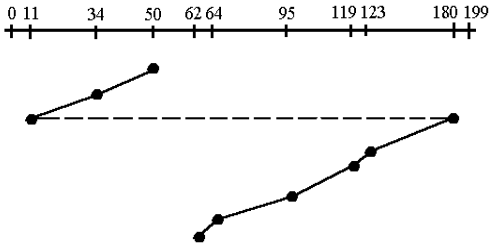


3. Elevator (SCAN)
This approach works like an elevator does. It scans down towards the nearest end and then when it hits the bottom it scans up servicing the requests that it didn't get going down. If a request comes in after it has been scanned it will not be serviced until the process comes back down or moves back up. This process moved a total of 230 tracks. Once again this is more optimal than the previous algorithm, but it is not the best.



4. Circular Scan (C-SCAN)

Circular scanning works just like the elevator to some extent. It begins its scan toward the nearest end and works it way all the way to the end of the system. Once it hits the bottom or top it jumps to the other end and moves in the same direction. Keep in mind that the huge jump doesn't count as a head movement. The total head movement for this algorithm is only 187 track, but still this isn't the mose sufficient.



5. C-LOOK
This is just an enhanced version of C-SCAN. In this the scanning doesn't go past the last request in the direction that it is moving. It too jumps to the other end but not all the way to the end. Just to the furthest request. C-SCAN had a total movement of 187 but this scan (C-LOOK) reduced it down to 157 tracks.

From this you were able to see a scan change from 644 total head movements to just 157. You should now have an understanding as to why your operating system truly relies on the type of algorithm it needs when it is dealing with multiple processes.