Lecture 1: Intro

OS is second most imp subject in CS after DSA.

Mainly it is used when we need to directly interact with the hardware/machine.

Topics like process synchronization and memory management are useful for software development.

OS is a software which takes control of the computer after machine is turned on power on tests are completed. It becomes the incharge of the machine.

Applications don't directly interact with the machine. The interact with OS and OS with hardware.

Why do we need manager/incharge?

Abstraction: Without the OS we will have to write code for basic functionalities such as displaying text on screen, moving mouse cursor etc. OS contains the necessary codes for the basic functionalities and makes things easier. This also keeps the hardware abstract.

Resource Management: We have limited resources in our machine and using them effectively is difficult. OS manages the resources.

Protection: It also protects the hardware from applications and applications from each other.

Do we always need a manager?

Let us say we want to operate hardware like lifts and ovens. In such cases the chips are simple and only one or few simple tasks are to be done in such cases we don't need OS.

Desktop OS: Windows, Linux, MacOS.

Mobile OS: Android, iOS.

There are more OS for printers, routers etc.

Three main services that OS provide:

Abstraction.

Resource Management.

Protection.

Lec 02: Types of OS

There can be many basis of classification.

Based on functionality provided by OS.

1. Single Tasking: MS-DOS, Only one process other than OS can exist in memory.
2. Multi Programming and Multi Tasking: Having multiple processes in RAM and assign them in smartly to CPU. Multiprogramming is general idea of managing multiple processes and multitasking is an extended version of multi-programming.
3. Multithreading:
4. Multiprocessing: For a system with multiple processors.

Thread: A thread is the smallest unit of execution/process that can be assigned to CPU. A process can be composed of single or multiple threads. Every OS uses concept of multi-threading nowadays.

Multi-user operating systems: Multiple users can use the same machine as different unique users.

Lec 03: Thread vs Process

Program gets loaded in RAM and then it is called a Process. A process is program in execution.

Pictorial representation of process with single thread. These are segments of a process.

Stack ↓ (Stack grows downwards)
Heap ↑ (Heap grows upward)
Text/Code
Data

If a process is single threaded then it will have only one stack. For multi-threaded processes we have multiple stacks.

Pictorial representation of process with multiple threads.

Stack ↓	Stack ↓	Stack ↓
Heap ↑ (Heap grows upward)		
Text/Code		
Data		

Multiple threads have multiple stacks but same Heap, data and code.

Concurrent and parallel have different meanings with reference to process execution.

More about threads:

1. Faster to create and terminate.
2. Share same address space.
3. Easier to communicate.
4. Context switching is easier.
5. Lightweight.

Lec 04: Multithreading Intro

- Multithreading vs Multitasking
- Some real world examples
- Advantages and Disadvantages

Multitasking: Listening to music and browsing. (Multiple tasks are being done)

Multithreading: Downloading and browsing. (Multiple things are being done within a process)

Real world examples of multithreading:

MS Word: Typing, saving, formatting is done together using multithreading.

IDEs: Error checking is done while the text is formatted.

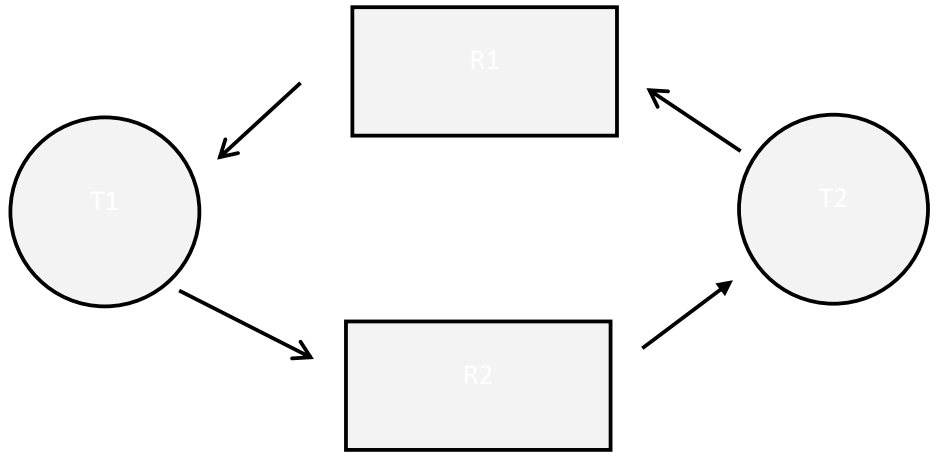
Advantages of multithreading:

1. Parallelism and improved performance
2. More responsiveness
3. Better resource utilization

Threads are also called light weight processes.

Disadvantages of Multithreading:

1. Difficulty in writing, testing and debugging code.
2. Can lead to deadlock and race conditions. (mainly when variables are shared, like in language JAVA)



Deadlock

The above diagram is technically called **The Resource Allocation Graph**.

Deadlock: T1 thread holds R1 resource and is waiting for R2. Meanwhile T2 holds R2 and is waiting for R1. Until T1 releases R1, T2 cannot be finished and similarly for T1.

Lec 05: User Threads vs Kernel Threads

User managed threads: The threads created by a process and the kernel is not aware about the threads and the process manages the threads.

Kernel managed threads: Managed by kernel and kernel is aware of everything going on.

	User Managed Threads	Kernel Managed Threads
Management	In user space	In kernel space
Context Switching	Fast	Slow

Blocking	One thread can block all other threads	A thread can block itself only.
Multicore or Multiprocessor	Cannot take advantage of multicore systems. Only concurrent execution on single processor.	Takes full advantage of multicore systems.
Creation/Termination	Fast	Slow

Usually every process have both kind of threads.

One to one: One user thread is mapped to only one kernel thread. On other user thread is mapped to this kernel thread. This is most common. This resembles with purely kernel managed threading system.

Many to one: Multiple user threads are mapped to one kernel thread. This resembles with purely user managed threading system.

Many to many: Multiple user threads are connected to multiple kernel threads. Very uncommon.

Lec 06: Intro to process

Program: File that resides on hard disk which can be executed.

Process: A program in execution.

The binary file (.exe) is loaded into RAM and then run. While it runs it is called a process.

Stack ↓ (Stack grows downwards) (for function calls, local variables)
Heap ↑ (Heap grows upward) (head for dynamically allocated memory)
Text/Code (Instructions to be executed)
Data (static and global variables)

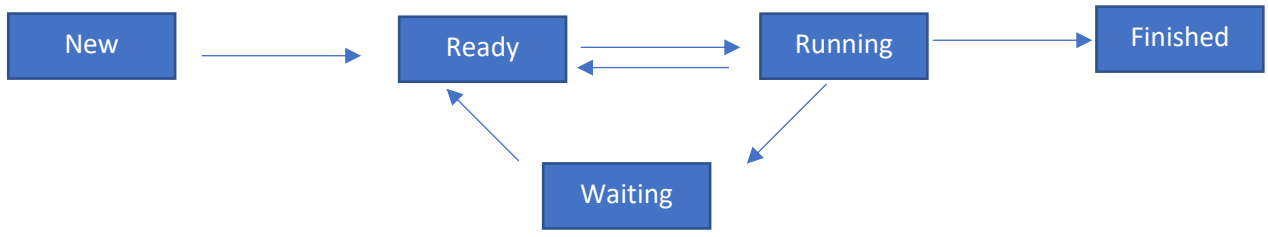
Lec 07: Process States

Single tasking systems (MS – DOS)

New Process-> Goes in Memory -> Finished/Terminated

In such systems processor remains idle for long.

Multiprogramming System: Systems which can have multiple processes in RAM other than OS simultaneously. They have 5 state model.



New State: We just double clicked an exe file. Process control info is created. But the exe file is not yet loaded in RAM, it is still in HD. It is also said that process is not created yet.

Ready State: exe file is partially or completely in RAM. Now it is said that process has been created and ready to be picked by the processor.

When the required part in in RAM. The dispatcher takes it to CPU and the process is said to be dispatched.

If a higher priority process comes or time-out happens the process is sent back to ready state.

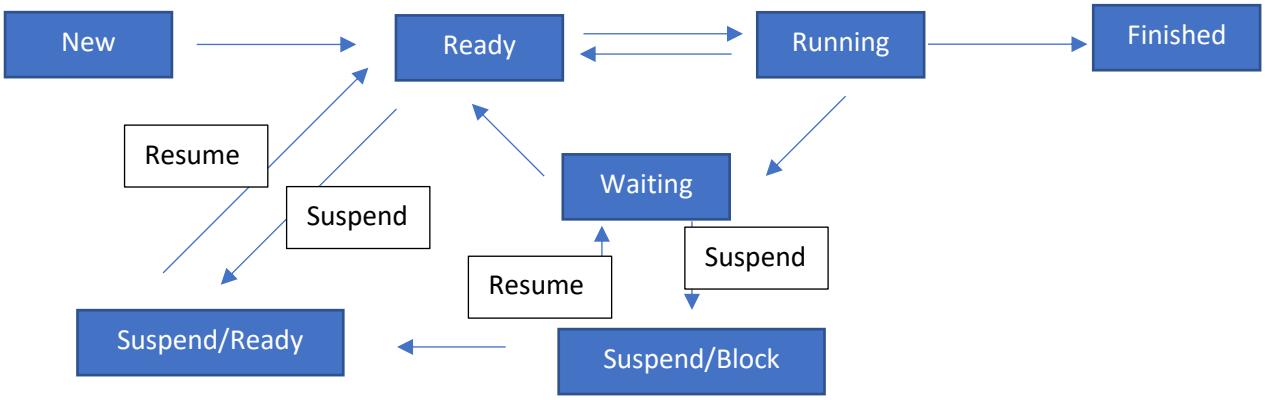
If the process requests I/O word it is sent to waiting state from running state.

Once the I/O is done, the process is again sent to ready state and then dispatched.

If the process gets finished or aborted, it is sent to finished state.

The seven-state model:

Two more states come out of waiting state.



When all the current processes are in waiting state, CPU is idle. In such cases, one of the processes is sent back to HD to pick a new process from HD.

While waiting the process is in RAM but on suspending the process is in HD.

Process can be directed to suspended state while waiting or while in ready state.

While in any suspended state the process is in HD but suspended state means that process has been through ready state for at least once and has not completed yet.

There is a dedicated place for suspended processes. Swap partition/space is available in Linux for these. In windows, a folder, page file stores these processes. (Things might have changed over time.)

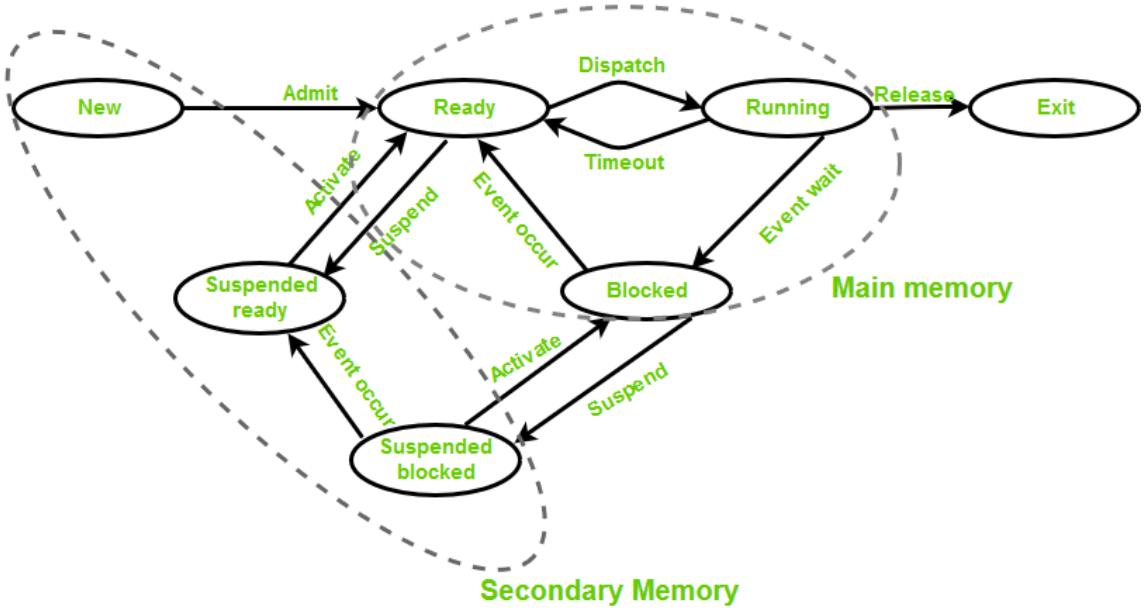


Diagram from GeeksForGeeks

Note: CPU interacts with RAM only. It doesn't interact with HD. CPU doesn't pick up processes. It is responsibility of OS to assign process to CPU.

Lec 08: Process State Block

PCB is a data structure. It is the most central data structure of OS. It is the most secure DS so that id doesn't get corrupted.

It stores info regarding process state. Some of the imp info is:

1. Process Id (Integer)
2. Process state
3. CPU Register
4. Accounts info (how much CPU time has been consumed etc.)
5. I/O Info
6. CPU scheduling info
7. Memory Info (memory blocks allocated to process etc.)

One imp CPU register is program counter which tells the next instruction to be executed.

Lec 09: Process Schedulers

I/O Bound Processes: Processes which do I/O work most of the time.

CPU Bound Processes: Processes which involve CPU most of the time.

1. Long Term Scheduler: Brings process from HD to RAM. Controls the degree of multi programming. Ideally it should bring a good mix of I/O bound and CPU bound processes. Decisions made by it occur on large gap that is why it is named so.
2. Short Term Scheduler: Moves the process from ready to running state. It assigns processor one of the picked processes. It has to make frequent decisions that is why it is named so. It also calls dispatcher. Dispatcher does the process control switch.
3. Medium Term Scheduler: It manages the suspending, blocking and resuming of process. It can be said that it moves processes from HD to RAM and vice versa.

Lec 10: Background for Scheduling Algos

The most crucial part is which process should be picked from ready state.

Most of the algos we study are for short term scheduler or for the jobs in ready Q.

- Different Queues: Ready Q, Job Q, I/O Q (each I/O device has its own Q).
- Short Term Scheduler and Dispatcher: STS picks one of the processes from ready Q and dispatcher comes into light. Once a process is picked, Dispatcher does the context switch, mode switch and tells from where a program should begin (a program might come from suspended state so it should not run from beginning).

When does STS picks a process?

1. When some process moves from running to waiting state.
2. When some other process moves from running to ready state.
3. When a new/existing process moves to ready state.
4. When a process is aborted or terminated.

Pre-emptive cases

Time related terms in Scheduling Algos:

1. Arrival Time (Point): The time at which the process enters into the ready queue is called the arrival time.
2. Completion Time (Point): The Time at which the process enters into the completion state or the time at which the process completes its execution, is called completion time.
3. Burst Time (Period): The total amount of time required by the CPU to execute the whole process is called the Burst Time. This does not include the waiting time. It is confusing to calculate the execution time for a process even before executing it hence the scheduling problems based on the burst time cannot be implemented in reality.
4. Turn Around Time (Period): The total amount of time spent by the process from its arrival to its completion, is called Turnaround time.
5. Waiting Time (Period): The Total amount of time for which the process waits for the CPU to be assigned is called waiting time.
6. Response Time (Period): The difference between the arrival time and the time at which the process first gets the CPU is called Response Time.

Goals of Scheduling Algos:

1. Max CPU utilization
2. Max throughput (throughput is number of jobs finished in unit time.)

3. Min Turn Around Time
4. Min Waiting Time
5. Min Response Time
6. Fair CPU Allocation (No Starvation)

Starvation: If a process has been waiting for a long time to get picked and executed then it is said to be starving.

Almost every time we care much about averages of the time.

GANTT Chart: Representation of CPU utilization by process with time stamps.

Lec 11: FCFS Scheduling

First Come First Serve. It is non pre-emptive. It is not batch processing.

The process which comes first gets executed first.

Advantages:

1. Simple and easy to implement.

Disadvantages:

1. Non-pre-emptive.
2. Convoy effect.
3. Average waiting time is not optimal.

Lec 12: SJF (Non Pre-Emptive)

Considers the jobs in increasing order of their Burst time.

Lec 13: SJF (Pre-Emptive)

Also called shortest remaining time first algo.

At every second, the process which has shortest remaining time is picked and executed.

Advantages:

1. Min average waiting time among all scheduling algos.

Disadvantages:

1. May cause high waiting time and response time for CPU bound processes.
2. Impractical, because the burst time can only be known after a process is aborted or terminated. However, there are ways to guess the CPU time of the process but still they would be guesses.

Lec 14: Priority Scheduling

It can be pre-emptive or non pre-emptive.

Every process has a priority and the process which has highest priority is picked.

It is very common.

Average times depend on arrival time and priorities of process.

Advantages:

1.

Disadvantages:

1. Starvation of low priority processes. Starvation of processes can be handled with a technique called aging. Aging means priority of process increases with time.

How can we manually assign priorities?

One simple way: Earlier is the deadline, more should be the priority. This is static way of assigning priority. There are more and better ways of assigning priorities.

Aging is a dynamically priority assigning technique.

FCFS and SJF both are priority scheduling algos. In FCFS, processes are prioritized on the basis of arrival time. In SJF, remaining time is considered for priority.

Lec 15: Round Robin Scheduling Algo

One of the most asked and popular scheduling algo.

Terminologies:

1. Time Quantum: Max time for which a process can be executed.

Idea: We maintain a circular ready queue. Pick the process from the front. Execute it not more than for the fixed time. If it does not finish in fixed time, push it back with its remaining time. Move to next process.

Average waiting time can be high.

Response time is generally good.

Deciding time quantum is difficult. Small time quantum will lead to too many context switches. Large time quantum will make it FCFS.

Lec 16: Multilevel Queue Scheduling

Concept: Divide the ready queue into multiple queues, use different scheduling algo for different queues. Put different processes in different queues. Then we have to arrange queues as well on the basis of some scheduling algo.

Commonly the queues are arranged on the basis of priorities or round robin algo.

Another way of running multiple queues is using two queues, one for foreground processes and on for background. Background queue is generally for lower priority processes.

The multi-level queue scheduling is used with feedback in Windows and Mac OS. Linux has improved itself and uses better scheduling algo.

In multilevel queue scheduling with feedback, process can be moved from one queue to another. It is one of the hardest, most useful and most flexible algos to implement.

Lec 17: Deadlocks

When a process runs it uses resources. Process runs in its own address space. Processes request the OS to get the resources and it is the responsibility of OS to manage the resources. The resources might be sharable or non-sharable.

Deadlock has been discussed earlier with diagram.

Four **necessary** conditions for deadlock:

1. Mutual Exclusion: Deadlocks are possible only if the resources cannot be shared among the processes.
2. Hold and Wait: Processes must be holding some resources and must be waiting for some resources.
3. No pre-emption: This pre-emption is different from process pre-emption. It means that an assigned resource cannot be taken until the process is finished/aborted.
4. Circular Wait: In circular wait, processes wait in circle for each other to finish.

All these 4 conditions are together responsible for deadlock.

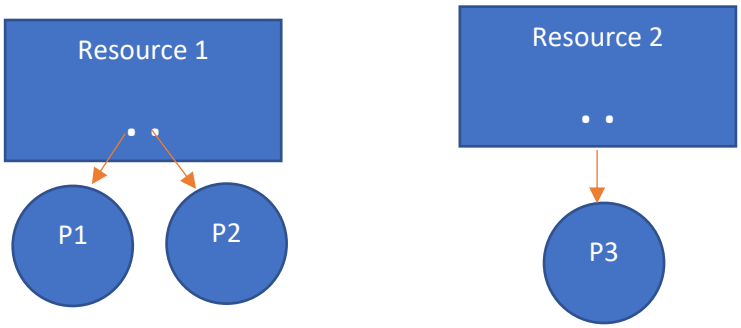
Real world example of deadlock: Trains coming towards each other.

The Resource Allocation Graph is used in DBMS.

In Resource Allocation Graph:

1. Process is represented in circle.
2. Resource is represented in rectangle.
3. There might be multiple instances of a resource. Multiple instances are shown by using dots in graph.

   This means that resource has three instances.
4. Arrow from process to resource means that process is waiting for the resource.
5. Arrow from resource to process means that the resource is allocated to process.



Two instances of R1 allocated to different processes. Both the instances of R2 allocated to same process.

Lec 18: Methods of handling deadlocks

1. Deadlock Prevention: Making sure that one of the necessary four conditions always fails. This is achieved by setting rules for processes or for resources. Before sending any request to access the resource it is made sure that it must not create deadlock. OS in this case has nothing to do for deadlock.
2. Deadlock avoidance: Processes can send all requests. It is responsibility of OS to check if granting the request will cause deadlock or not. OS can use any algo for the check. Banker's algo is one such algo.
3. Detection and Recovery: OS periodically checks if deadlock has happened or not. If a deadlock is found, one of the processes is killed to free the processes.
4. Ignore the deadlock: If a type of deadlock is very rare. OS let the deadlock happen and user has to reboot the system. Ignoring deadlock was a method used by windows and Linux earlier but nowadays OS use advanced ways to deal with them.

Deadlock handling techniques might vary from resource to resource.

Lec 19: Deadlock Prevention

Prevent/Eliminate any of the necessary conditions for deadlock.

1. Mutual Exclusion: It cannot be completely eliminated practically. It is done by a method called Spooling.
   For example: the printers have their own job queue and each print request is added to that queue.
   Spooling is not a full proof solution because the job queue might get full.
2. Hold and Wait: There are two ways to eliminate Hold and Wait.
   a. Knowing which resources are going to be used by the process. This way is quite impossible.
   b. A process while sending a request for a resource must release all the other resources it is holding. This way is also not very useful because the released resources by this process might be useless for the other processes.
3. No pre-emption: OS takes away the resources allocated to a process. This might be problematic because the process might be in the middle of something.

4. Circular Wait: We number the resources. Then we make a rule to assign the resources to the processes. For example: If a process has taken resource with number n, it cannot acquire any resource which has number lesser than n. This prevents cycle.

Lec 20: Deadlock Avoidance (Banker's Algorithm)

	Allocated		Max Required	
	R0	R1	R0	R1
P0	0	1	7	5
P1	2	0	3	2
P2	3	0	9	0
P3	1	1	2	2
P4	0	0	4	3

Total instances of R0 = 10 and R1 = 5

Deadlock avoidance works if the max required for each process is known before. Knowing which resources are required and in what quantity before the execution of process is impossible. But let us work on this.

Let us say P3 requests for <1, 0>. Should the OS grant the request?

Make the resource allocation diagram by assuming that the resources have been allocated to P3.

Now try to generate a safe sequence. If a safe sequence can be generated then it is said that safe state can be achieved after resource allocation and the resources will actually be allocated.

Safe Sequence:

- A permutation of the processes
- in which when the processes are executed one by one in sequence
- and after execution of every process the resources allocated to the process are released
- and can be used by other processes.
- If all the processes can be executed,
- then the sequence is called, Safe Sequence.

Algo to find Safe Sequence:

Create a Need Matrix.

Must remember: For generating safe sequence we must assume that the resources have been allocated.

	Allocated		Max Required		Need Matrix	
	R0	R1	R0	R1	R0	R1
P0	0	1	7	5	7	4
P1	2	0	3	2	1	2
P2	3	0	9	0	6	0
P3	1+1	1+0	2	2	0	1
P4	0	0	4	3	4	3

SafeSequence = {} //initialize empty safe sequence

```
While(all the processes are not added to safe sequence)

{

   a. Find Pi such that need(i) <= available
   b. If (no such Pi can be found)
      Return false;
   c. Else
      Available+=Allocated //we are updating Available because this
      process will be finished and resources will be available for all
      upcoming processes.
      Add Pi to the SafeSequence

}
```

For the above problem, the SafeSequence is P1 P3 P4 P0 P2.

Before jumping to finding safe sequence do some basic checks such as, is the process asking for resources more than available or the resource is asking for more than max.

Banker's Algo is developed by Djiktra. It is the only algo for Deadlock Avoidance.

Drawbacks:

1. Requires max required resources.
2. Assumes that resources will be released.

Note: Failing to generate SafeSequence does not guarantee that Deadlock will happen, because the processes need lesser than or equal to max (max is the upper limit). Also, there are more things to consider which we did not.

Lec 21: Deadlock Detection and Recovery

It has two steps:

1. Detection: All resources involved have only one instance and there is a cycle in resource allocation diagram, then there is a deadlock. If there are multiple instances then cycle does not guarantee deadlock. If there are multiple instances, then we use Modified Banker's Algo. We don't consider the resources which do not hold any resources.
2. Recovery:
    a. We can pre-empt resources: It might be problematic because the process might be in middle of something. There might chances of starvation too because the resources might be away from the process for too long.
    b. Kill one or more processes: Kill a process and check if the resources are enough or not.

Lec 22: Process Synchronization

Processes are of two types:

1. Independent: They run independently.
2. Co-operative: They interact with other processes.

OS has to manage the communication between the processes.

Inter-process communication may happen on same device or across multiple devices using computer networks.

Generally, people think that process synchronization happens when we have multiple processors.

In single processor devices concurrent execution may lead to inter-process communication.

Con-current execution: In round robin, processes get executed one by one. When multiple processes run on intervals, this execution is called con-current execution.

Inter-process communication happens in a device using shared memory. Global Variables are examples of shared memory.

```
//Global Variables

Int size = 10;

Char buffer[size];

Int in=0, out = 0;

Int count = 0;

//Producer is process 1

Void producer(){

While(true)

{

While(count==Size){

//if no free space is available producer waits

}

Buffer[in]=producerItem();
```

```
In = (in+1)%size;

Count++; //IMP LINE

}

}

//Consumer is process 2

Void consumer(){

While(true)

{

While(count==0){//consumer waits infinite if the count is 0 }

consumeItem(buffer[out]);

out = (out+1)%size;

count--; //IMP LINE

}

}
```

Understanding IMP LINES:

Reg = CPU register

Count++ is interpreted as:

1. Reg = count
2. Reg = reg +1
3. Count = reg

Count-- is interpreted as:

1. Reg = count
2. Reg = reg-1
3. Count = reg

Let us say, count is 8 and count++ was executing in P1 and as soon as statement 2 i.e. Reg = reg +1 executed, P1 was pre-empted. It means that count was not updated.

Now, the CPU is executing P2 and when it comes to count--, it would see that count is 8 because the count has not been updated in P1. Now, P2 makes the count to 7 and CPU goes back to P1.

When P1 resumes, the count sets to 9, because the incremented value was stored in CPU register.

Due to all this, there were inconsistent values of count for both the processes.

Race Condition: When the output depends on the sequence of execution of instructions in a concurrent/multi-programming/multi-threading environment, the condition is called Race Condition.

Lec 23: Critical Section

```
//Global Variable

Int balance = 100;

//Process 1

Void deposit (int x){

Balance = balance + x; //Critical Section

}

//Process 2

Void withDraw (int x)

{

Balance = balance - x; //Critical Section

}
```

Terminologies:

1. Critical section: The part of code in which we access shared variables is called critical section. The rest of the part is called remainder section or non-critical section.

The inconsistency is due to partial execution of critical section of one process and then execution of another process. We need to put some logic so that only one of the critical section gets executed completely and then only the another critical section gets executed.

The logic has two parts. One has to be before the critical section which allows/denies the process to enter the critical section, this is called entry section. And another has to be after the critical section which controls the exit of process from critical section, this is called exit section.

There might be more than two processes.

Lec 24: Goals of Synchronization

1. Mutual Exclusion: Only one process is allowed to enter critical section.
2. Progress: The processes which do not wish to execute their critical section must not block other processes.
3. Bounded Waiting (Fair): It means fairness. Any process should not be waiting for too long to enter the critical section.
4. Performance: The logic must not be slow. There are two ways to lock the processes so that critical section is accessed by one process only.
    a. Hardware Locking Mechanism: Better than Software Locking Mechanism.
    b. Software Locking Mechanism:

Mutual Exclusion and Progress must be achieved by any synchronization mechanism, others are optional.

Lec 25: Overview of Synchronization Mechanisms

1. Disabling Interrupts: The Synchronization problems happen because of race condition. What if a process before entering the critical section declares that it must not be interrupted while

it is in critical section. This sounds good. But this works only for single processor devices. There is one more problem that the process might stay in critical section for too long.

2. Locks (Mutex): This mechanism is basic building block for synchronization mechanisms. Locks can be implemented in software and hardware both. A lock is made, process enters critical section, does the work and releases the lock (there might be pre-emption when the process is in critical section). While the lock is on, only the processes which have critical section have to wait until the critical section is executed completely.
   Hardware Locks are better than software.
   Software locks: Peterson's Implementation, Bakery Algo.
3. Semaphore: It is higher level mechanism. When a process enters the critical section, it causes wait and when the process exits the critical section it causes signal. Wait and signal must be atomic. Semaphores are built on Locks.
4. Moniters: They are managed in JAVA. They are software mechanisms. The shared variables are put into a class and synchronized methods are implemented to handle the variables. They are built on top of Semaphores.

Applications of Synchronizations:

There are two types of synchronization:

1. Process synchronization
2. Thread synchronization: They are used a lot.

Lec 26: Lock for Synchronization

```
int amount = 100;
bool lock = false;

//Process 1
void deposit(int x){
    while(lock==true){
        //if lock is true then keep waiting
    }
    //----------------------------------------Line A
    lock = true;//setting lock true to make other process wait
    amount = amount + x;
    lock = false;//setting lock false to send signal
}

//Process 2
void withdraw(int x){
    while(lock==true){
        //if lock is true then keep waiting
    }
    //----------------------------------------Line B
    lock = true;//setting lock true to make other process wait
    amount = amount - x;
    lock = false;//setting lock false to send signal
}
```

The above code implements lock but does not make mutual exclusion. What if the P1 is pre-empted at Line A and P2 starts execution. The lock is false at this moment but P1 just entered the critical section.

So we have to add one more thing that is supported by hardware called test_and_set or TSL Lock.

Note: The code is written in C/CPP style for demonstration only. The code/process might be in some another language.

```
int amount = 100;
bool lock = false;

bool test_and_set(bool *ptr){
    bool old = *ptr;
    *ptr = true;
    return old;
}

//Process 1
void deposit(int x){
    while(test_and_set(lock)){
        //if lock is true then keep waiting
    }
    //----------------------------------------
    lock = true;//setting lock true to make other process wait
    amount = amount + x;
    lock = false;//setting lock false to send signal
}

//Process 2
void withdraw(int x){
    while(test_and_set(lock)){
        //if lock is true then keep waiting
    }
    //----------------------------------------
    lock = true;//setting lock true to make other process wait
    amount = amount - x;
    lock = false;//setting lock false to send signal
}
```

The basic problem of synchronization is solved now.

Lec 27: Semaphore

Most interesting OS topic.

Problems with the lock based mechanism:

1. Busy Waiting: The process was wating in an infinite loop. Think of this as people standing outside the govt office. The people are not doing anything except waiting.
2. No Queue: The processes do not maintain a queue.

3. This lock based solution is feasible for simple problems only. If the resources have multiple instances then the lock based solution would be very complex.

A semaphore is a count variable and a queue.

```
struct Sem{
    int count;
    queue<Process> q;
}
```

Whenever a process is using resource the count is decreased.

If count becomes lesser than 0, we add the process to queue. Before acquiring the resource we call the wait function and after releasing the resource we call the signal function.

-ve count indicates number of people in q.

+ve count indicates number of resources available.

```
Sem s(3, empty);
void wait(){
    s.count--;
    if(s.count<0){
        1. Add the caller process to q
        2. sleep(Pi)
    }
}

void signal(){
    s.count++;
    if(s.count<=0){
        1. Remove a process from q
        2. WakeUp(Pi)
    }
}
```

Original implementation of Semaphore by Dijktra:

```
s=3
P(){
    while(s==0); //Dijktra used busy waiting
    s=s-1;
}
V(){
    s=s+1;
}
```
//P and V have some meaning in Dutch so Dijktra used them.

Lec 28: Binary Semaphores

```
struct BinSem{
    bool val;
```

```
    Queue q;
}

BinSem s(true, empty);//Global

void wait(){
    if(s.val==1) s.val=0;
    else{
        1. Put the process P in q
        2. Sleep(P)
    }
}

void signal(){
    if(q is empty){
        s.val = 1;
    }else{
        1. Pick a process from q
        2. WakeUp(P)
    }
}
```

Some more points about semaphores:

1. The wait and signal must be atomic. Atomicity is achieved using locks.
2. The Kernel has to implement them.

If a process is atomic that means either the whole process is executed or whole process is skipped. There is no chance of partial execution.

Lec 29: Monitors

Monitors are also built on top of locks.

```
Class AccountUpdate{
    private int bal;
    void synchronized deposit(int x){
        bal = bal + x;
    }
    void synchronized withdraw(int x){
        bal = bal - x;
    }
}; //The code is C/CPP styled for understanding only
```

JAVA is used for Monitors and the shared variables are put into a class. Synchronized methods are made using the synchronized keyword in JAVA to manage the variables.

If a class has one or more synchronized functions then the class is called synchronized class.

Lec 30: Priority Inversion

Priority inversion is a problem asked very often in interviews and exams.

Low Priority Process (L)

Entry Section
Critical Section
Exit Section

High Priority Process (H)

Entry Section
Critical Section
Exit Section

L starts running, comes to critical section and gets pre-empted.

Now, H starts running but it has to wait for L to finish. High priority is waiting for Lower Priority. This is called priority inversion.

This can be simply solved by making some changes that is a high priority process is waiting for some result by lower priority, then L must not be pre-empted.

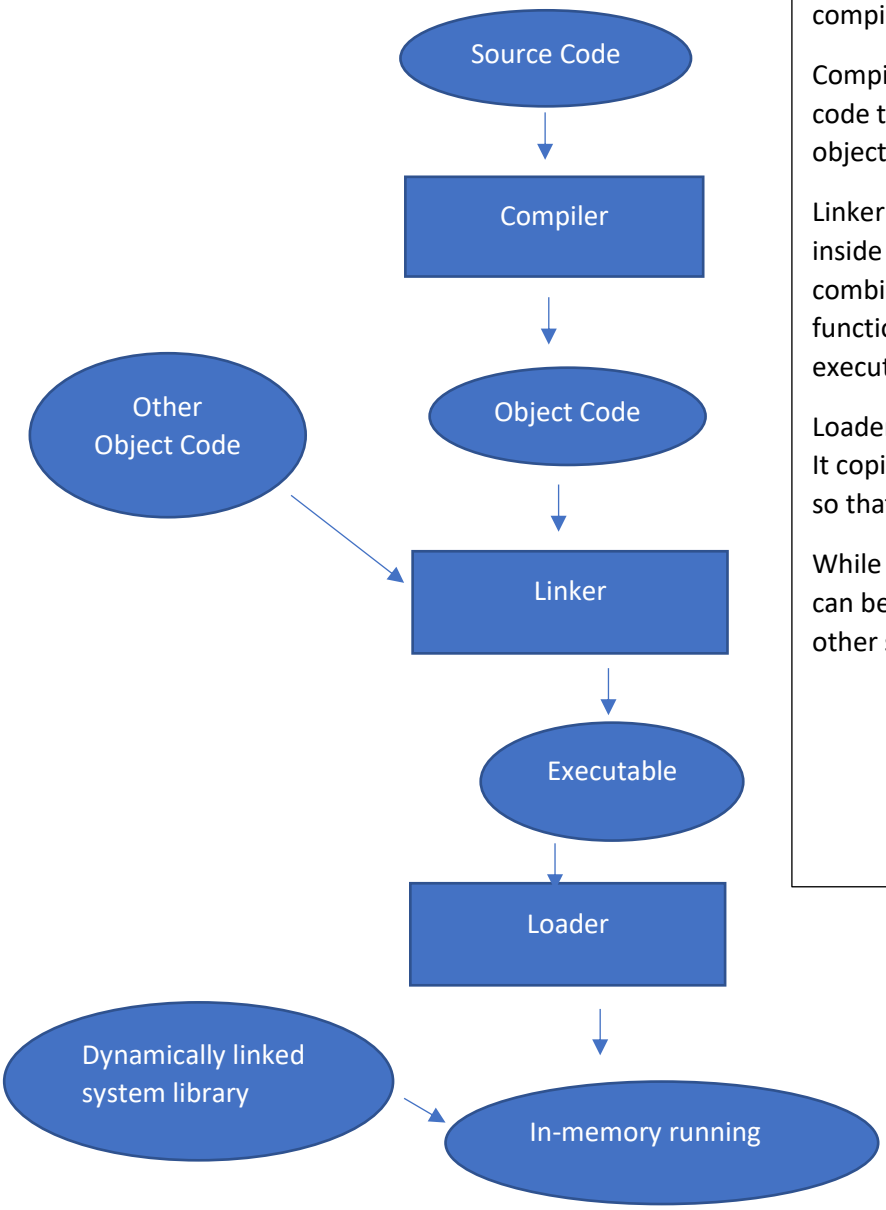
But, what if there is one more process M having medium priority which does not share the critical section. In case of non pre-emption of L, M would also have to wait even if does not share the critical section.

Or if L gets pre-empted and M runs then also it can be said that medium priority process has blocked high priority process. Again priority inversion happened.

This can solved by many techniques. One of which is priority inheritance.

Priority Inheritance: If a low priority process has been running through its critical section which is shared by a high priority process then for the time being the lower priority process inherits the priority and becomes a higher priority process.

Lec 31: How are programs compiled and run



Source code is given to compiler.

Compiler converts the source code to machine dependent object code.

Linker links the functions called inside of main function. It also combines the other library functions and produces an executable.

Loader runs the executable file. It copies this into main memory so that it can run step by step.

While a program is running it can be dynamically linked with other system libraries.

Dynamic linking is something which requires a process to go through memory and search if the linked resource is loaded or not into the main memory. And ideally no process should be allowed to go through memory and search. So dynamically memory require support of OS. Processes are not allowed to go through main memory and on behalf of them the OS does the searching.

Lec 32: Memory Management in OS

Memory Hierarchy

Computers have two main parts:

1. CPU

2. Memory: Memory has several forms.

Memory is desired to have:

1. Fast accessability
2. High Capacity
3. Low cost

All these three things do not come together.

The above diagram shows closeness of memory to CPU.

Cache is closest and has minimum access time and Secondary Memory is farthest and has highest access time.

Lec 33: Address binding

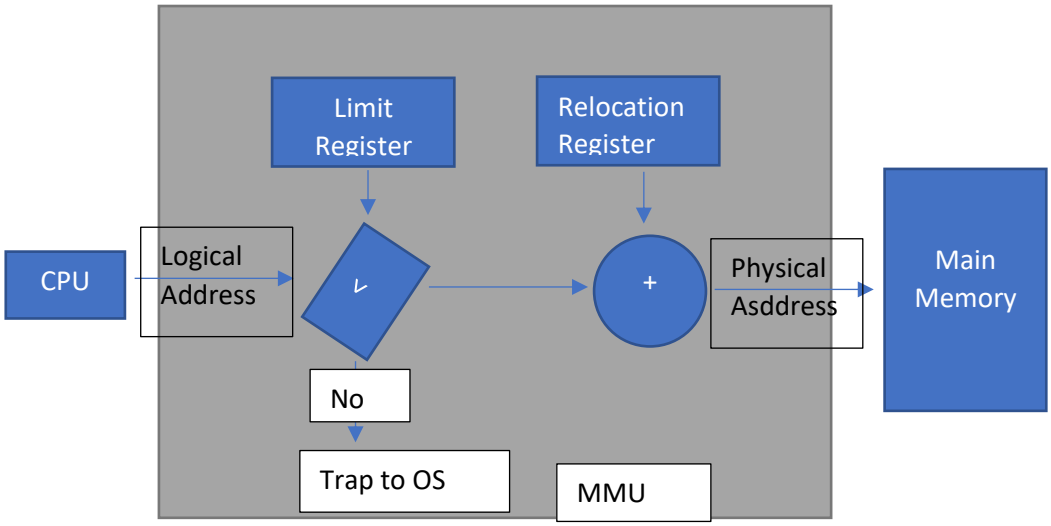
When we produce binary of a program, the binary contains relative or relocatable addresses. The addresses are like 000, 001 and so on (they begin with 0). To run the binary file it must be brought into main memory. The main memory has many processes running. The addresses in main memory are called physical addresses. When the binary is brought into some slot, let us say that slot starts from 5000 then the addresses in binary will be as 5000, 5001 and so on. Address binding is mapping of relocatable addresses and physical addresses.

Address binding may happen at different times. It may happen at:

1. Compile time: Compiler knows where is the file going to be loaded in main memory. In this type of binding, there is nothing like relocatable addresses. They have physical addresses only.
2. Load time: Loader has the executable file and it puts the process in main memory and all that mapping thing happens here. The problem here is that when a program is loaded in main memory its physical addresses can't be changed.
   Why do we need to change the physical addresses?
   The program might be doing some I/O. Meanwhile the process should be moved to hard disk so that another process can be executed. In shifting the process from main to hard disk and back to main the physical addresses may get changed.
3. Run time: Happens in all modern OS. Processes are allowed to move. First logical addresses are generated and these addresses are converted to physical addresses. Memory management unit does the logical to physical address conversion. They need hardware support. Load time and compile time binding do not require hardware support.

Lec 34: Runtime Binding

In runtime binding, the addresses generated by CPU are called logical addresses. And these are converted to physical at runtime.



When process is loaded into memory, context switching happens. These registers are loaded by OS. When logical addresses are generated, the registers are used to generate physical addresses. The limit register checks the limit of the process. If the addresses generated are beyond the limit of memory than it is said that it is trap to OS and process does not continue.

Achieving the same in software is quite time consuming.

Lec 35: Evolution of memory management

Memory Allocation in multi-tasking:

1. Static
   Types of Static Partition:
   a. Equal Size: All the partitions have equal size.
      Let the partitions have 10 mb size. If ith process has ni size then waste of memory is 10-ni. Because of the wastage, this method is not preferred. External and Internal Fragmentation happens.
   b. Unequal Size: In a slot is allocated for the process such that memory wastage is minimum.
      Slots size: 4, 5, 6 and Process Size: 3. Then slot of size 4 will be allocated.
      Here also, internal and external fragmentation happens.
2. Dynamic
   Defragmentation:
   Types of Dynamic Partition:
   a. Contiguous
      Memory is left empty initially. As the processes come, memory is allocated to them. No wastage happens.
      Let us say total memory is 10 mb and three processes come one by one: P1 (3mb), P2 (2mb) and P3 (3mb). Memory is allocated to them and 2 mb of memory is left unallocated. P2 finishes and goes out of memory. Now if P4 (3mb) wants to get into

memory, but it cannot because the free memory (2 mb by P2 and 2 mb more) might not be contiguous. Here external fragmentation happens.
To effectively use memory and avoid external fragmentation, there are two methods: Compaction and Defragmentation.
Compaction: We move processes together so that allocated space is together is and unallocated is together. This compaction is quite costly.

b. Non-contiguous
    1. Paging
    2. Segmentation
    3. Paging with segmentation

External Fragmentation:

Internal Fragmentation:

Lec 36: Dynamic Partitioning

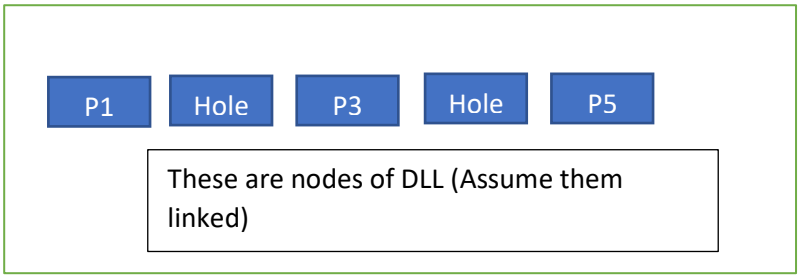
In dynamic partitioning the memory is used more effectively than static but it is tougher than static.

OS
P1
Hole
P3
Hole
Hole
P5

Holes are the empty spaces, where some processes were running which are now finished. Detection of holes is not easy so we have two ways to maintain holes:

1. Bitmap: Each memory unit is associated with a bitmap. If the value in bitmap for memory unit is 1 then this unit is occupied else if the value is 0 then unoccupied. Bitmap approach requires a lot of space.
2. Linked lists: The memory slots are managed as doubly linked lists.

Processes And Holes
P1
Hole
P3
Hole
Hole
P5



Whenever a process is finished, the
holes may combine to form a large hole. Every hole holds its size also. Whenever a new process arrives one of the holes needs to be allocated to that process. The allocation of hole to process is done in many ways, they are:

1. First Fit
2. Next Fit

3. Best Fit
4. Worst Fit
   And More

Lec 37: First Fit, Best Fit, Next Fit and Worst Fit

P1
//Hole 1: 10 mb
P2
//Hole 2: 4 mb
P3
//Hole 3: 12 mb

Let us say P4 (x mb) comes and we have to allocate memory for this.

1. First Fit: Let x=5. Traverse through the list for holes from the beginning. The first hole which has size greater than or equal to 5 mb will be allocated to P4 and remaining 5 mb will become hole now.
2. Best Fit: Let x=3. Traverse through the whole list for holes. Find the hole which has size equal to or greater than the required and difference between sizes of hole and process is minimum. Hole 2 will be allocated. And remaining 1 mb will become hole now.
   Disadvantages:
   a. Traverse through whole list
   b. It would create a many small holes.
3. Next Fit: Let us say x = 8. Traverse through the list and apply first fit. Hole 1 is allocated and hole of 2 mb created. Now let us say a process with size 3 mb comes. Now don't start from beginning. Start from the next memory slot, that is 2 mb hole and apply first fit. Do first fit in circular manner until a suitable slot is found or one complete circular tour is completed.
4. Worst Fit: Find the largest whole and allocated that to the process. The remaining part to would become a new hole now.

First Fit algo is considered the best among these algos.

Lec 38: Paging in Memory Management

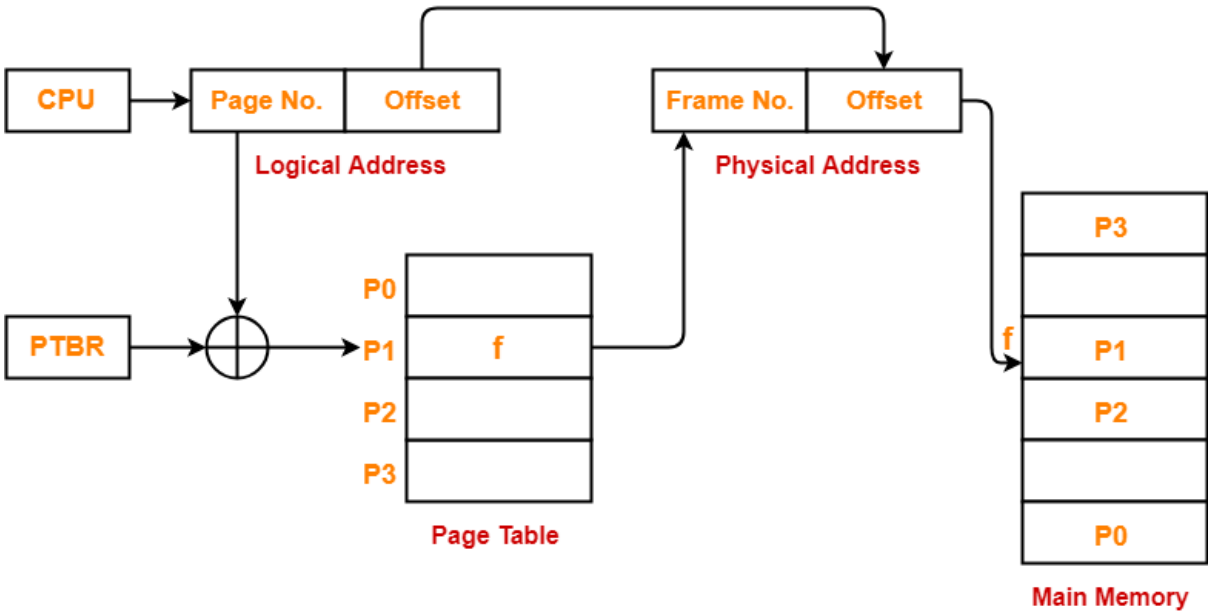
Most popular idea in Memory Management is Paging.

Let us understand paging.

In paging, main memory is divided into slots of equal sizes and each slot is called frame. Similarly, process is divided into parts of equal sizes and part is called page. Size of a page must be equal to size of frame. When a process is loaded into memory, different pages of a process are stored in different frames. The different frames may or may not be at contiguous locations.

We use runtime binding for managing addresses of pages of processes.

To manage the addresses, we use page table. It stores mapping of logical addresses of pages and memory frames. They need hardware support. Basically, page table takes the logical address of a page and tells the physical address of a page in main memory. Page table has to be in main memory while the process is running.



Translating Logical Address into Physical Address

Besides PTBR (Page table base register) there is one more register for security purpose. Most of the processors have minimum page size as 4 kb.

Logical addresses are continuous.

Offset can be seen as items within a page. Page 4 and Offset 3 means 3rd item in 4th page.

While a process is running, it is not necessary that all the pages of the process are in main memory. If machine looks for a page in main memory and could not find it then it is called page fault. That page has to be brought from Hard Disk to main memory.

Along with mapping of addresses, the page table stores more data about pages such as if the page was modified in RAM or not. If the page is not modified OS does not need to write back the page into Hard Disk. One more thing page table stores, is access rights regarding the pages.

Processes are stored in Hard Disk and Frames are stored in Main Memory. Processes while doing I/O or something like that are stored at special places in Hard Disk in form of pages and brought back to main memory when I/O finishes.

Lec 39: Virtual Memory and Related Concepts

The idea of paging was to keep a part of process in main memory in form of pages instead of whole process. Virtual memory is very similar concept to this.

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard disk that's set up to emulate the computer's RAM.

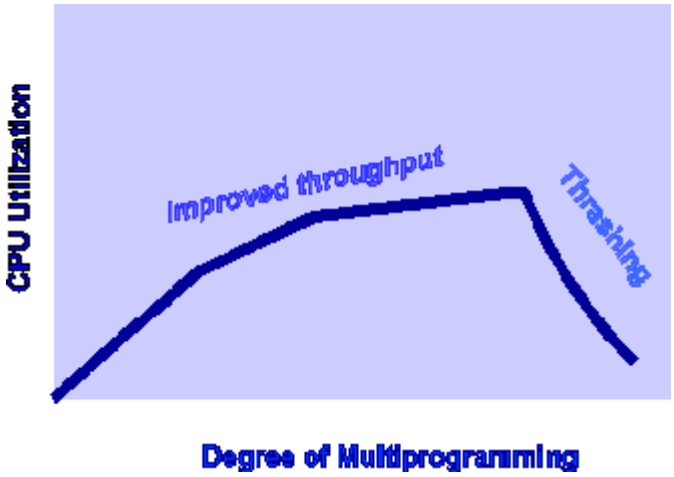
Hit Ratio: (Hits)/(Hits+Misses) OR (Hits)/(Total Requests)

Performance is badly affected by page faults.

Lec 40: TLB, Demand Paging, Thrashing, Page Replacement Algorithms

Terms related to Virtual Memory:

1. TLB:
   Translational Lookaside Buffer.
   Every page of a process may have multiple logical addresses.
   When logical address is to be converted to physical address, there is a need to lookup in the page table. This lookup might be very frequently for same page number. TLB optimizes this lookup.
2. Demand Paging: Not keeping all the pages of a process in main memory.
   In a purely demand paging environment, minimal number of pages from a process are kept into main memory.
3. Thrashing: By keeping lesser the number of pages from processes into main memory, we can have many processes running and hence increase the degree of multiprogramming. Keeping too less pages in main memory will make too many page faults. Page faults are expensive.



   CPU utilization increases with increase of multiprogramming up to a level only. After that with increase in degree of multiprogramming the CPU utilization falls. This fall is called thrashing.
4. Page Replacement:
   When a page fault occurs, there is a need to bring a page from HD to RAM. RAM may not have free frames. Then we need to replace the required page with some page in RAM. This is called page replacement. There are various algos for page replacement.
   Page replacement can be:
   a. Local: When a process needs a page which is not in the memory, it can bring in the new page and allocate it a frame from its own set of allocated frames only.
      It is most popular.
   b. Global: When a process needs a page which is not in the memory, it can bring in the new page and allocate it a frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
      Gives overall better performance but process performance is compromised.

Page Replacement Algos:

1. First In First Out:
   Suffers from Belady's Anomaly.
   This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the

queue. When a page needs to be replaced page in the front of the queue is selected for removal.
Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10-page faults.

2. Optimal:
   Ideal but not feasible.
   In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.
3. Least Recently Used:
   Most popular.
   In this algorithm, page will be replaced which is least recently used.

Lec 41: Segmentation in Memory Management

Segmentation is alternative of paging.

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments. (In paging the divisions must of same size.)

In segmentation the process is divided according to user's view. Dividing according to user's view means related items together.

With segmentation we can achieve both which we used to achieve by paging:

1. Loading parts of process at non-contiguous location
2. Using virtual memory

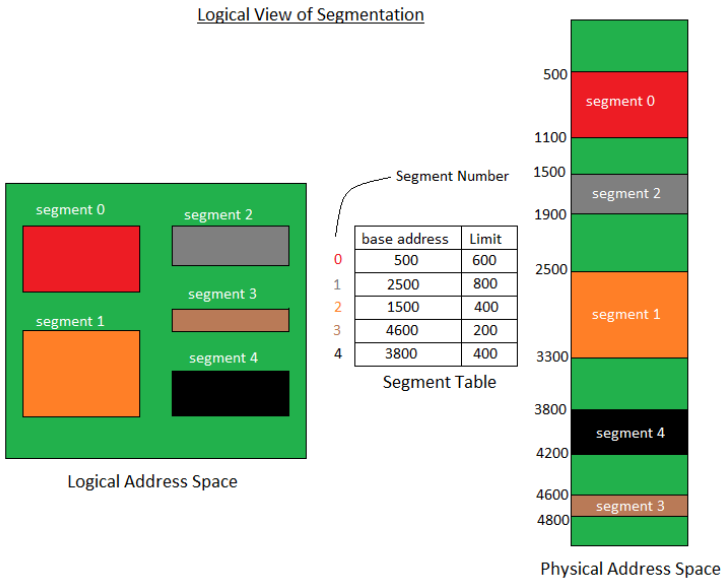
Types of pure segmentation:

1. Simple Segmentation: All segments need to be loaded into main memory at runtime (maybe contiguous or non-contiguous)
2. Virtual Segmentation: Not all segments need to be loaded into main memory at runtime.

Logical address to Physical address conversion is a bit more complex here. Segment table is used for managing addresses.

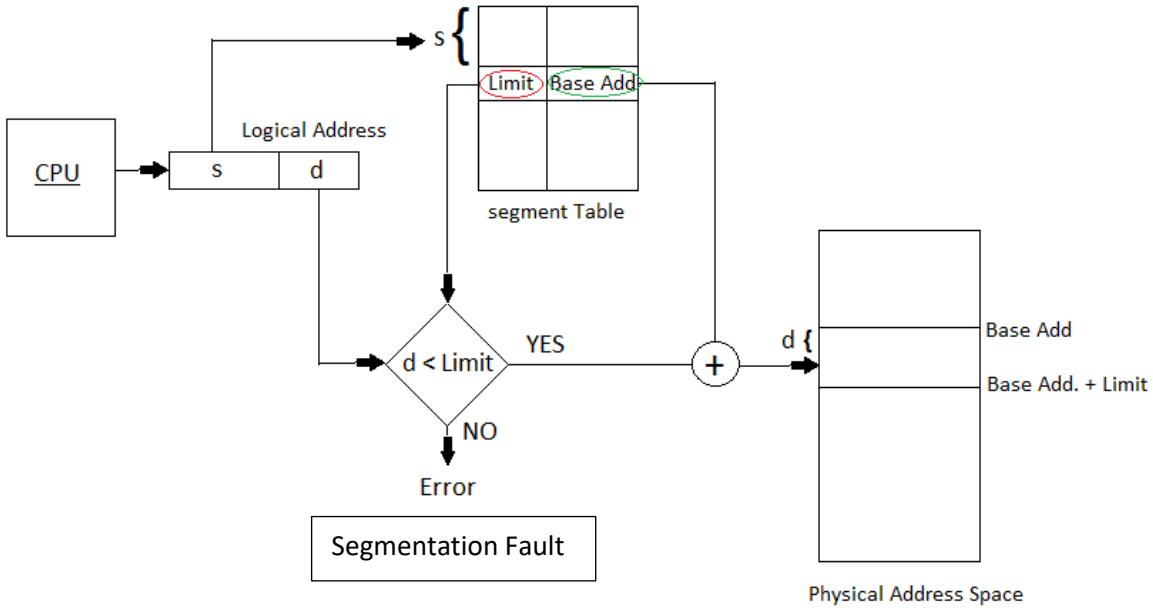
Segment Table – It maps two-dimensional Logical address into one-dimensional Physical address. It's each table entry has:

1. Base Address: It contains the starting physical address where the segments reside in memory.
2. Limit: It specifies the length of the segment.



Besides base address and limit we have present/absent column in segment table then set value 1 if segment is present in main memory else 0.

Translation of two-dimensional Logical Address to one dimensional Physical Address.



Address generated by the CPU is divided into:

1. Segment number (s): Number of bits required to represent the segment.
2. Segment offset (d): Number of bits required to represent the size of the segment.

Advantages of Segmentation –

1. No Internal fragmentation.
2. Segment Table consumes less space in comparison to Page table in paging.

Disadvantage of Segmentation –

As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

Lec 42: Segmentation with Paging

Aim: To achieve advantages of paging and segmentation.

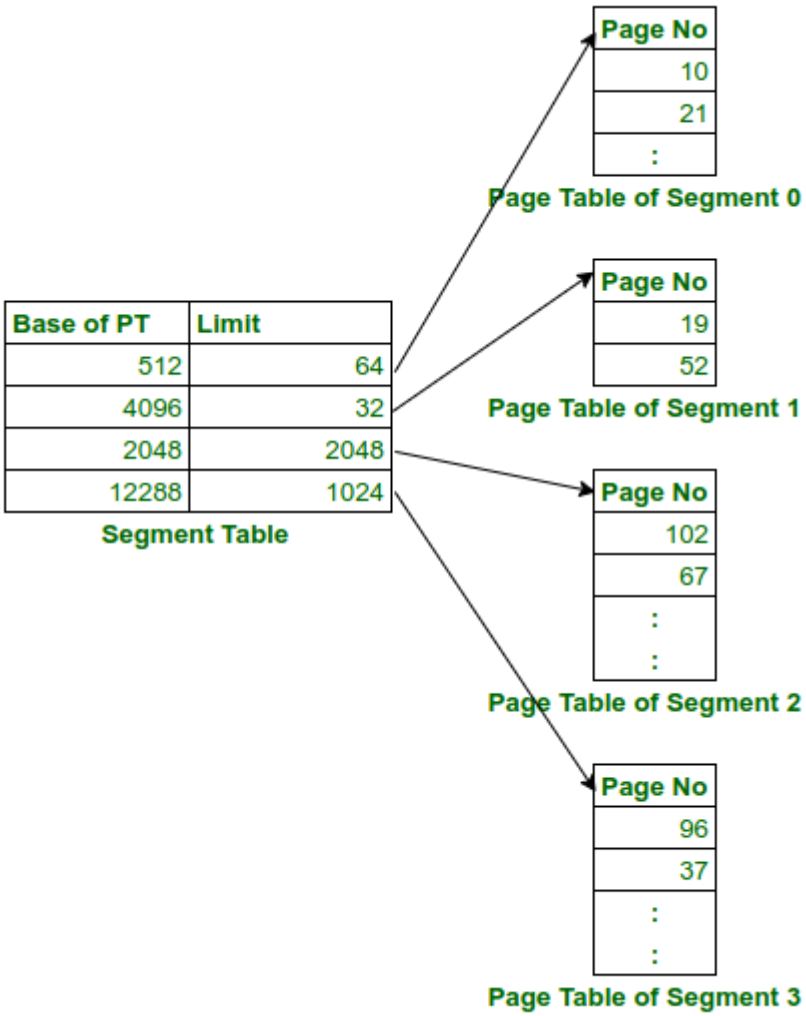
We do segmentation, then for each segment we do paging. The mapping between logical and physical addresses is complex now.

Since a process is divided into segment so each process has a segment table. Then each segment in individually divided into pages so each segment has its page table.

Segment table base register tells where is the segment table for the required process.

Segment table -> Page Table -> Frame

Once we find the frame, offset is added to go to desired place.



Advantages of Paged Segmentation:

1. No external fragmentation
2. Reduced memory requirements as no. of pages limited to segment size.

3. Page table size is smaller just like segmented paging,
4. Similar to segmented paging, the entire segment need not be swapped out.

Disadvantages of Paged Segmentation:

1. Internal fragmentation remains a problem.
2. Hardware is more complex than segmented paging.
3. Extra level of paging at first stage adds to the delay in memory access.

Pure segmentation is rarely used. Either pure paging or paged segmentation is used.