UNIT III: PROCESSOR MANAGEMENT

Overview-About Multi-Core Technologies-Job Scheduling Versus Process Scheduling- Process Scheduler-Process Scheduling Policies-Process Scheduling Algorithms –A Word about Interrupts
Deadlock-Seven Cases of Deadlock -Conditions for Deadlock- Modeling Deadlock-Strategies for Handling Deadlocks –Starvation
Concurrent Processes: What Is Parallel Processing-Evolution of Multiprocessors- Introduction to Multi-Core Processors-Typical Multiprocessing Configurations-Process Synchronization Software

JOB SCHEDULING VERSUS PROCESS SCHEDULING

Generally scheduling of jobs is actually handled on two levels by most operating systems. The Processor Manager is a composite of two sub managers. They are

1. **Job Scheduler** - one in charge of job scheduling
2. **Process Scheduler** - the other in charge of process scheduling.

There exists a hierarchy exists between the Job Scheduler and the Process Scheduler.

JOB SCHEDULER

Generally A user views a job either as a series of global job steps—compilation, loading and execution. Therefore, each job (or program) passes through a hierarchy of managers.

- First one it encounters is the Job Scheduler; this is also called the **high-level scheduler**.
- It is only concerned with selecting jobs from a queue of incoming jobs and placing them in the process queue, whether batch or interactive, based on each job's characteristics.
- The Job Scheduler's goal is to put the jobs in a sequence that will use all of the system's resources as fully as possible.
- The Job Scheduler strives for a balanced mix of jobs that require large amounts of I/O interaction and jobs that require large amounts of computation.
- Its goal is to keep most components of the computer system busy most of the time.
- If the Job Scheduler selected several jobs to run and they had a lot of I/O (**I/O bound jobs**), then the I/O devices would be kept very busy. The CPU might be busy handling the I/O, so little computation might get done.

- On the other hand, if the Job Scheduler selected several consecutive jobs with a great deal of computation **(CPU-bound Jobs)**, then the CPU would be very busy doing that. The I/O devices would be idle waiting for I/O requests.

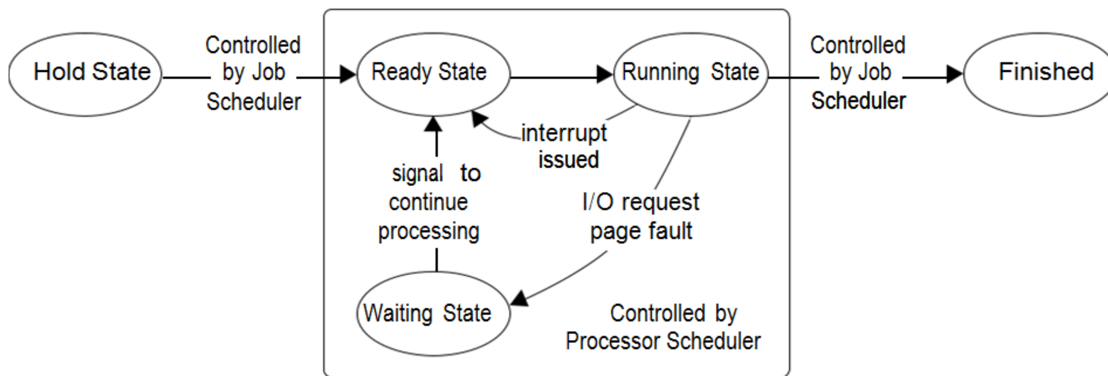
PROCESS SCHEDULER

- After a job has been placed on the READY queue by the Job Scheduler, the Process Scheduler takes over it.
- The Process Scheduler is the **low-level scheduler** that assigns the CPU to execute the processes of those jobs placed on the READY queue by the Job Scheduler.
- It determines which jobs will get the CPU, when, and for how long. It also decides when processing should be interrupted, determines which queues the job should be moved to during its execution, and recognizes when a job has concluded and should be terminated.
- To schedule the CPU, the Process Scheduler alternate between CPU cycles and I/O cycles.

Note:

- In a highly interactive environment, there’s also a third layer of the Processor Manager called the **middle-level scheduler**.
- When the system is over-loaded, the middle-level scheduler finds it is advantageous to remove active jobs from memory to reduce the degree of multiprogramming, which allows jobs to be completed faster.
- The jobs that are swapped out and eventually swapped back in are managed by the middle-level scheduler.
- In a single-user environment, there’s no distinction made between job and process scheduling because only one job is active in the system at any given time.

JOB AND PROCESS STATUS



As a job moves through the system, it's always in one of five states (or at least three). These are called the **job status** or the **process status**.

1. HOLD
2. READY
3. RUNNING
4. WAITING
5. FINISHED

HOLD: When the job is accepted by the system, it's put on HOLD and placed in a queue.
In some systems, the job spooler (or disk controller) creates a table with the characteristics of each job in the queue and notes the important features(CPU time, priority, special I/O devices required, and maximum memory) of the job. This table is used by the Job Scheduler to decide which job is to be run next.
READY: From HOLD, the job moves to READY when it's ready to run but is waiting for the CPU. In some systems, the job (or process) might be placed on the READY list directly.

RUNNING: The job is being processed. In a single processor system, this is one "job" or process.

WAITING: The job is blocked; it can't continue until a specific resource is allocated or an I/O operation has finished.

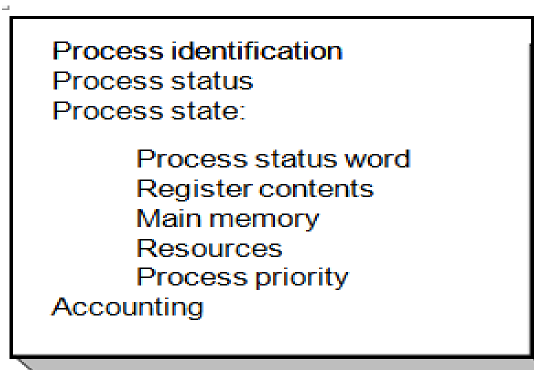
FINISHED: The job is FINISHED and returned to the user.

The transition from one job or process status to another is initiated by either the Job Scheduler or the Process Scheduler:

Transition	Handled by	Description
HOLD to READY	Job Scheduler	-
READY to RUNNING	Process Scheduler	According to some predefined algorithm (i.e., FCFS, SJN, priority scheduling, SRT, or round robin).
RUNNING to WAITING		Initiated by an instruction in the job such as a command to READ, WRITE, or other I/O request, or one that requires a page fetch.
WAITING to READY		Is initiated by a signal from the I/O device manager that the I/O request has been satisfied. when page fault handler signal that the page is now in memory
RUNNING to READY		According to some predefined time limit or other criterion, for example a priority interrupts
RUNNING to FINISHED	Job scheduler / Process Scheduler	when (1) the job is successfully completed or (2) the operating system indicates that an error has occurred and the job is being terminated prematurely

PROCESS CONTROL BLOCK

Each process in the system is represented by a data structure called a **Process Control Block (PCB).** The PCB contains the basic information about the job, including what it is, where it's going, how much of its processing has been completed, where it's stored, and how much it has spent in using resources.



Process Identification

Each job is uniquely identified by the user's identification number.

Process Status

This indicates the current status of the job— HOLD, READY, RUNNING, or WAITING and the resources responsible for that status.

Process State

This contains all of the information needed to indicate the current state of the job such as:

Process Status Word—the current instruction counter and register contents when the job isn't running but is either on HOLD or is READY or WAITING. If the job is RUNNING, this information is left undefined.

Register Contents—the contents of the register if the job has been interrupted and is waiting to resume processing.

Main Memory—pertinent information, including the address where the job is stored and, in the case of virtual memory, the mapping between virtual and physical memory locations.

Resources—information about all resources allocated to this job. These resources can be hardware units (disk drives or printers, for example) or files.

Process Priority—used by systems using a priority scheduling algorithm to select which job will be run next.

Accounting

This contains information used mainly for billing purposes and performance measurement. It indicates what kind of resources the job used and for how long. It include

- Amount of CPU time used from beginning to end of its execution.
- Total time the job was in the system until it exited.
- Main storage occupancy—how long the job stayed in memory until it finished execution. This is usually a combination of time and space used; for example, in a paging system it may be recorded in units of page-seconds.
- Secondary storage used during execution. This, too, is recorded as a combination of time and space used
- System programs used, such as compilers, editors, or utilities.
- Number and type of I/O operations, including I/O transmission time, that includes utilization of channels, control units, and devices.

- Time spent waiting for I/O completion.
- Number of input records read (specifically, those entered online or coming from optical scanners, card readers, or other input devices), and number of output records written.

PROCESS SCHEDULING POLICIES

In a multiprogramming environment, there are usually more jobs to be executed than could possibly be run at one time. Before the operating system can schedule them, it needs to resolve three limitations of the system:

1. There are a finite number of resources (such as disk drives, printers, and tape drives)
2. Some resources, once they're allocated, can't be shared with another job (e.g., printers)
3. Some resources require operator intervention i.e., they can't be reassigned automatically from job to job (such as tape drives).

Several criteria to be considered while scheduling processes are

1. **Maximize throughput**: Run as many jobs as possible in a given amount of time.
2. **Minimize response time**. Quickly turn around interactive requests. This could be done by running only interactive jobs and letting the batch jobs wait until the inter-active load ceases.
3. **Minimize turnaround time**. Move entire jobs in and out of the system quickly.
4. **Minimize waiting time**. Move jobs out of the READY queue as quickly as possible. This could only be done by reducing the number of users allowed on the system so the CPU would be available immediately whenever a job entered the READY queue.
5. **Maximize CPU efficiency.** Keep the CPU busy 100 percent of the time. This could be done by running only CPU-bound jobs (and not I/O-bound jobs).
6. **Ensure fairness for all jobs.** Give everyone an equal amount of CPU and I/O time.

VARIOUS TIMINGS RELATED WITH PROCESS

1. Arrival time
2. Waiting time
3. Response time
4. Burst time
5. Completion time
6. Turn Around Time

Arrival Time- Arrival time is the point of time at which a process enters the ready queue.
Waiting Time- Waiting time is the amount of time spent by a process waiting in the ready queue for getting the CPU.

Waiting time = Completion time (or) Turn Around Time – Burst time

Response Time - Response time is the amount of time after which a process gets the CPU for the first time after entering the ready queue.

Response Time = Time at which process first gets the CPU – Arrival time

Burst Time

- Burst time is the amount of time required by a process for executing on CPU.
- It is also called as **execution time** or **running time**.
- Burst time of a process can not be known in advance before executing the process.
- It can be known only after the process has executed.

Completion Time

- Completion time is the point of time at which a process completes its execution on the CPU and takes exit from the system.
- It is also called as **exit time**.

Turn Around Time

- Turn Around time is the total amount of time spent by a process in the system.
- When present in the system, a process is either waiting in the ready queue for getting the CPU or it is executing on the CPU.

  Turn Around time = Burst time + Waiting time (or)

  Turn Around time = Completion time – Arrival time

PROCESS SCHEDULING ALGORITHMS

The Process Scheduler relies on a **process scheduling algorithm**, based on a specific policy, to allocate the CPU and move jobs through the system. Generally scheduling algorithms are classified as **Preemptive and Non-Preemptive algorithms**.

- A scheduling strategy that interrupts the processing of a job and transfers the CPU to another job is called a **preemptive scheduling policy**. It is widely used in time-sharing environments.
- **In non-preemptive scheduling policy** once a job captures the processor and begins execution, it remains in the RUNNING state uninterrupted until it issues an I/O request (natural wait) or until it is finished.

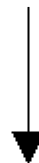
Early operating systems used non-preemptive policies designed to move batch jobs through the system as efficiently as possible and most current systems emphasis on interactive use and **response time.**

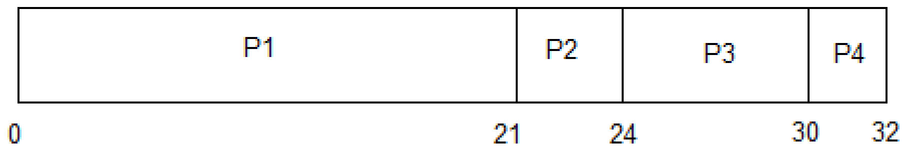
FIRST-COME, FIRST-SERVED

- First-come, first-served (FCFS) is a non-preemptive scheduling algorithm
- It handles jobs according to their arrival time: the earlier they arrive, the sooner they're served.
- It's a very simple algorithm to implement because it uses a FIFO queue.
- This algorithm is fine for most batch systems, but it is unacceptable for interactive systems because interactive users expect quick response times.
- With FCFS, as a new job enters the system its PCB is linked to the end of the READY queue and it is removed from the front of the queue when the processor becomes available

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with **Arrival Time** 0, and given **Burst Time**, let's find the average waiting time using the FCFS scheduling algorithm.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2





This is the GANTT chart for the above processes

Calculating Average Waiting Time

AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution. Lower the Average Waiting Time, better the scheduling algorithm.

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms

Problems with FCFS Scheduling

1. It is **Non Pre-emptive** algorithm, which means the **process priority** doesn't matter.
2. Not optimal Average Waiting Time.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource (CPU, I/O etc) utilization.

Convoy Effect:

Convoy Effect is a situation where many processes, who need to use a resource for short time are blocked by one process holding that resource for a long time.
This essentially leads to poort utilization of resources and hence poor performance.

SHORTEST JOB FIRST (SJF) SCHEDULING

Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first.

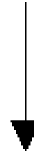
- This is the best approach to minimize waiting time.
- This is used in Batch Systems
- It is of two types:
  1. Non Pre-emptive
  2. Pre-emptive
- To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time.
- This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all)

Problem with Non Pre-emptive SJF

If the **arrival time**s for processes are different, it leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed, but this can be solved using the concept of **aging**.

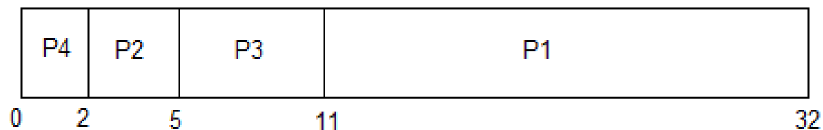
Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with **Arrival Time** 0, and given **Burst Time**, let's find the average waiting time using the SJF scheduling algorithm.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

PRIORITY SCHEDULING

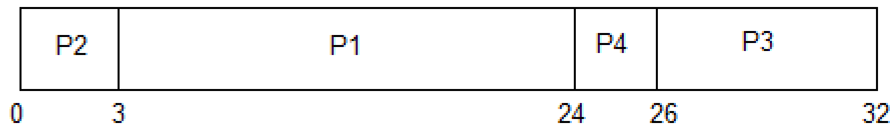
- **Priority scheduling** is a non-preemptive algorithm
- Most common scheduling algorithms in batch systems, even though it may give slower turnaround to some users
- It allows the programs with the highest priority to be processed first, and they aren't interrupted until their CPU cycles (run times) are completed or a natural wait occurs.
- If two or more jobs with equal priority are present in the READY queue, the processor is allocated to the one that arrived first.
- With a priority algorithm, jobs are usually linked to one of several READY queues by the Job Scheduler based on their priority so the Process Scheduler manages multiple READY queues instead of just one.
- Priorities can also be determined by the Processor Manager based on characteristics intrinsic to the jobs such as **Memory requirements, Number and type of peripheral devices, Total CPU time and Amount of time already spent in the system.**

Some systems increase the priority of jobs that have been in the system for an unusually long time to expedite their exit. This is known as **aging**.

Example

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Problem with Priority Scheduling Algorithm

The chances of **indefinite blocking** or **starvation exist.** If new higher priority processes keeps coming in the ready queue then the processes waiting in the ready queue with lower priority may have to wait for long durations before getting the CPU for execution.

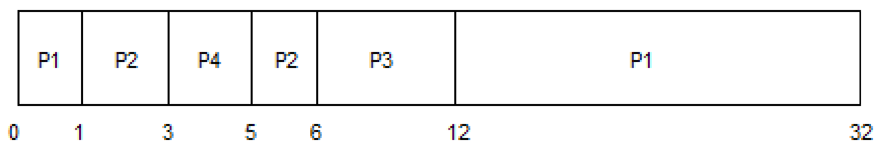
SHORTEST REMAINING TIME NEXT

- It is Pre-emptive Version of Shortest Job First Scheduling
- In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with **short burst time** arrives, the existing process is preempted or removed from execution, and the shorter job is executed first.
- This algorithm can't be implemented in an interactive system because it requires advance knowledge of the CPU time required to finish each job.

- It is often used in batch environments when it is desirable to give preference to short jobs, even though SRT involves more overhead than SJN because the operating system has to frequently monitor the CPU time for all the jobs in the READY queue and must perform context switching for the jobs being swapped (switched) at preemption time

PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



The average waiting time will be, ((5-3) + (6-2) + (12-1))/4 = 4.25 ms

The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

Way of Execution of Processes:

P1 arrives first, hence it's execution starts immediately, but just after 1 ms, process **P2** arrives with a **burst time** of 3 ms which is less than the burst time of **P1**, hence the process **P1**(1 ms done, 20 ms left) is pre-emptied and process **P2** is executed.

As **P2** is getting executed, after 1 ms, **P3** arrives, but it has a burst time greater than that of **P2**, hence execution of **P2** continues. But after another millisecond, **P4** arrives with a burst time of 2 ms, as a result **P2** (2 ms done, 1 ms left) is pre-empted and **P4** is executed.

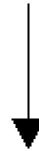
After the completion of **P4**, process **P2** is picked up and finishes, then **P2** will get executed and at last **P1**.

ROUND ROBIN SCHEDULING

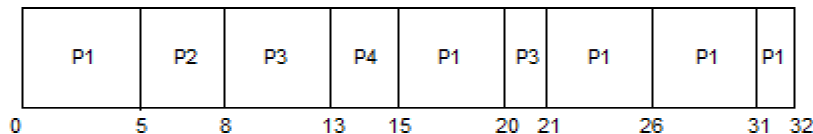
- Round robin is a preemptive process scheduling algorithm
- It is used extensively in interactive systems.
- It's easy to implement
- It is not based on job characteristics but on a predetermined slice of time that's given to each job to ensure that the CPU is equally shared among all active processes
- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.

Consider a set of Processes arrive in same time and provided with time quantum of 5 ms.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2



The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Way of Execution of Processes:

1. Jobs are placed in the READY queue using a first-come, first-served scheme
2. Process Scheduler selects the first job from the front of the queue, sets the timer to the time quantum, and allocates the CPU to this job.

3. If processing isn't finished when time expires, the job is preempted and put at the end of the READY queue and its information is saved in its PCB.
4. In the event that the job's CPU cycle is shorter than the time quantum

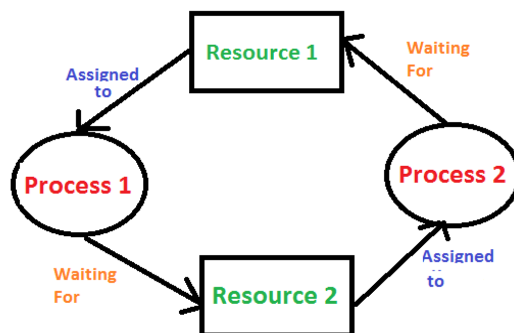
- If this is the job's last CPU cycle and the job is finished, then all resources allocated to it are released and the completed job is returned to the user.
- If the CPU cycle has been interrupted by an I/O request, then information about the job is saved in its PCB and it is linked at the end of the appropriate I/O queue. Later, when the I/O request has been satisfied, it is returned to the end of the READY queue to await allocation of the CPU.

Scheduling Algorithms: Summary

Algorithm	Policy Type	Best for	Disadvantages	Advantages
FCFS	Nonpreemptive	Batch	Unpredictable turnaround times	Easy to implement
SJN	Nonpreemptive	Batch	Indefinite postponement of some jobs	Minimizes average waiting time
Priority scheduling	Nonpreemptive	Batch	Indefinite postponement of some jobs	Ensures fast completion of important jobs
SRT	Preemptive	Batch	Overhead incurred by context switching	Ensures fast completion of short jobs
Round robin	Preemptive	Interactive	Requires selection of good time quantum	Provides reasonable response times to interactive users; provides fair CPU allocation

DEADLOCK

Deadlock is a situation where a set of process(es) are blocked because each process(es) is holding a resource and waiting for another resource acquired by some other process(es) which require a set of resources from requested process(es).



Necessary Conditions for Deadlock

A deadlock situation can arise if and only if the following four conditions hold simultaneously in a system-

Mutual Exclusion: At least one resource is held in a non-sharable mode that is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and Wait: There exists a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

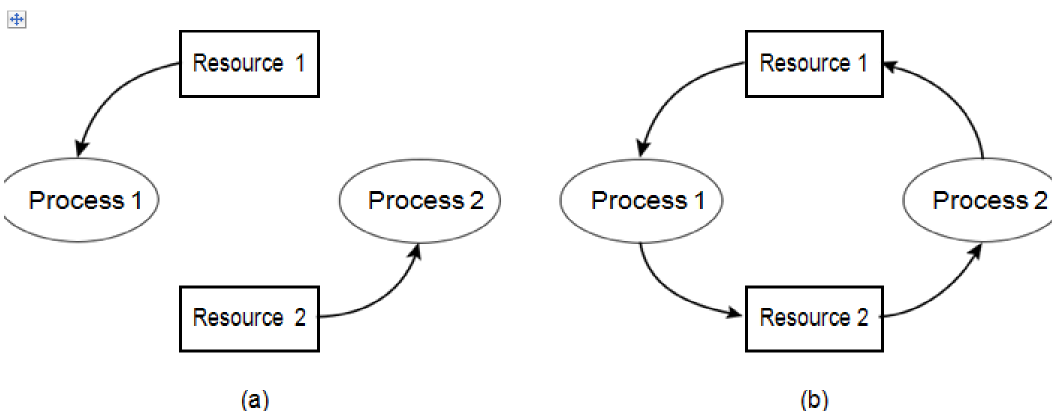
No Preemption: Resources cannot be preempted; that is, a resource can only be released voluntarily by the process holding it, after the process has completed its task.

Circular Wait: There must exist a set $\{p_0, p_1, ...p_n\}$ of waiting processes such that p_0 is waiting for a resource which is held by p_1, p_1 is waiting for a resource which is held by p_2,..., p_{n-1} is waiting for a resource which is held by p_n and p_n is waiting for a resource which is held by p_0.

MODELING DEADLOCKS

Holt showed how the four conditions can be modeled using **directed graphs.** These graphs uses set of symbols:

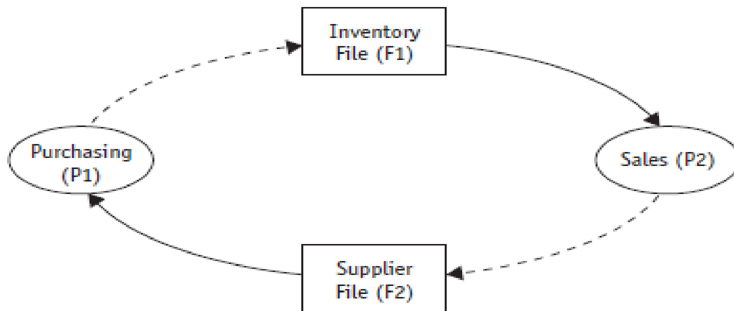
- Processes represented by circles
- Resources represented by squares.
- A solid arrow from a resource to a process means that the process is holding that resource.
- A dashed line with an arrow from a process to a resource means that the process is waiting for that resource. The direction of the arrow indicates the flow. If there's a cycle in the graph then there's a deadlock involving the processes and the resources in the cycle.



SEVEN CASES OF DEADLOCK

Case 1: Deadlocks on File Requests

If jobs are allowed to request and hold files for the duration of their execution, a deadlock can occur as the simplified directed graph graphically illustrates.



Consider two application programs, purchasing (P1) and sales (P2), which are active at the same time. Both need to access two files, inventory (F1) and suppliers (F2), to read and write transactions. One day the system deadlocks when the following sequence of events takes place:

1. Purchasing (P1) accesses the supplier file (F2) to place an order for more lumber.
2. Sales (P2) accesses the inventory file (F1) to reserve the parts that will be required to build the home ordered that day.
3. Purchasing (P1) doesn't release the supplier file (F2) but requests the inventory file (F1) to verify the quantity of lumber on hand before placing its order for more, but P1 is blocked because F1 is being held by P2.
4. Meanwhile, sales (P2) doesn't release the inventory file (F1) but requests the supplier file (F2) to check the schedule of a subcontractor. At this point, P2 is also blocked because F2 is being held by P1.

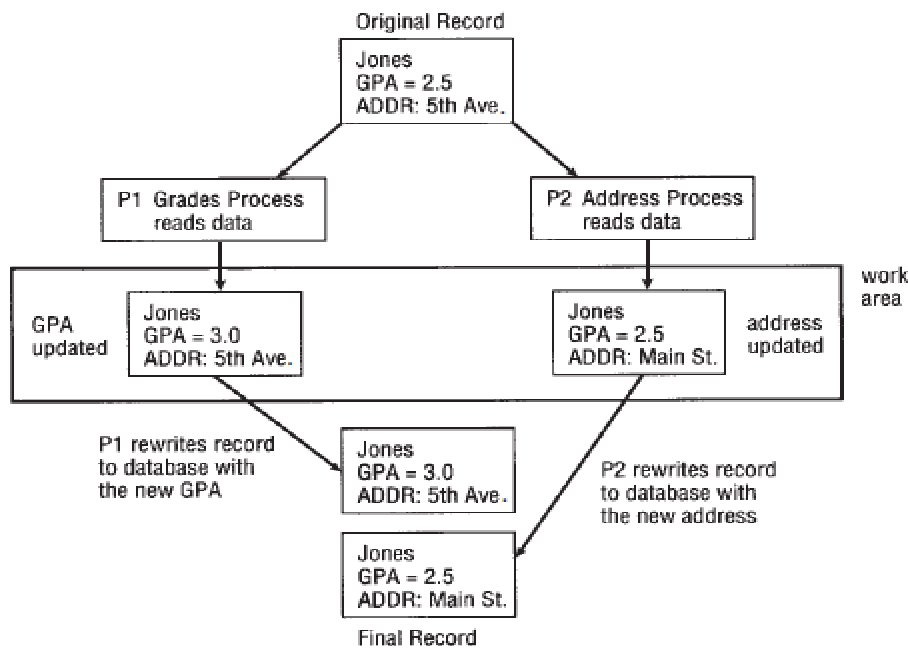
Case 2: Deadlocks in Databases

A deadlock can also occur if two processes access and lock records in a database. **Locking** is a technique used to guarantee the integrity of the data through which the user locks out all other users while working with the database. Locking can be done at three different levels:

- The entire database can be locked for the duration of the request;
- A subsection of the database can be locked;
- Only the individual record can be locked until the process is completed.

Locking the entire database prevents a deadlock from occurring but it restricts access to the database to one user at a time and, in a multiuser environment, response times are significantly slowed; this is normally an unacceptable solution.
When the locking is performed on only one part of the database, access time is improved but the possibility of a deadlock is increased because different processes sometimes need to work with several parts of the database at the same time.

If locks are not used to preserve their integrity, the updated records in the database might include only some of the data—and their contents would depend on the order in which each process finishes its execution. This is known as a **race** between processes and is illustrated in the following example



1. The grades process (P1) is the first to access your record (R1), and it copies the record to its work area.
2. The address process (P2) accesses your record (R1) and copies it to its work area
3. P1 changes your student record (R1) by entering your grades for the fall term and calculating your new grade average.
4. P2 changes your record (R1) by updating the address field.
5. P1 finishes its work first and rewrites its version of your record back to the database. Your grades have been updated, but your address hasn't.
6. P2 finishes and rewrites its updated record back to the database. Your address has been changed, but your grades haven't.

The system can’t allow the integrity of the database to depend on a random sequence of events.

Case 3: Deadlocks in Dedicated Device Allocation

The use of a group of dedicated devices, such as a cluster of DVD read/write drives, can also deadlock the system.

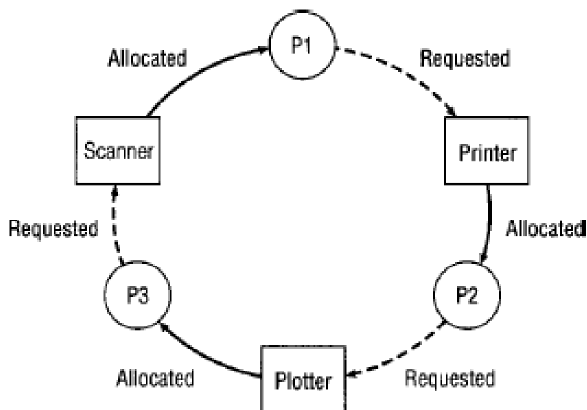
Let’s say two users running a program (P1 and P2), and both programs will eventually need two DVD drivers to copy files from one disc to another. The system is small, however, and when the two programs are begun, only two DVD-R drives are available and they’re allocated on an “as requested” basis.

1. P1 requests drive 1 and gets it.
2. P2 requests drive 2 and gets it.
3. P1 requests drive 2 but is blocked.
4. P2 requests drive 1 but is blocked.

Neither job can continue because each is waiting for the other to finish and release its drive - an event that will never occur. A similar series of events could deadlock any group of dedicated devices.

Case 4: Deadlocks in Multiple Device Allocation

Deadlocks aren’t restricted to processes contending for the same type of device; they can happen when several processes request, and hold on to, several dedicated devices while other processes act in a similar manner as shown below.



Case 5: Deadlocks in Spooling

- Printers are usually sharable devices, called virtual devices that use high-speed storage (Spooler) to transfer data between it and the CPU.

- The spooler accepts output from several users and acts as a temporary storage area for all output until the printer is ready to accept it. This process is called **spooling**.
- The spooler receives the pages one at a time from each of the document but the pages are received separately, several page ones, page twos, etc.
- The spooler is full of partially completed output so no other pages can be accepted, but none of the jobs can be printed out (which would release their disk space) because the printer only accepts completed output files.

Case 6: Deadlocks in a Network

A network that's congested or has filled a large percentage of its I/O buffer space can become deadlocked if it doesn't have protocols to control the flow of messages through the network

Case 7: Deadlocks in Disk Sharing

- Disks are designed to be shared, so it is common for two processes to be accessing different areas of the same disk.
- This ability to share creates an active type of deadlock, known as livelock.
- Processes use a form of busy-waiting that's different from a natural wait. In this case, it's waiting to share a resource but never actually gains control of it. Notice that neither process is blocked which would cause a deadlock. Instead, each is active but never reaches fulfillment.

STRATEGIES FOR HANDLING DEADLOCK

In general, operating systems use one of following strategies to deal with deadlocks:

- ✓ Prevent one of the four conditions from occurring (**prevention**).
- ✓ Avoid the deadlock if it becomes probable (**avoidance**).
- ✓ Detect the deadlock when it occurs and (**Deadlock Detection**)
- ✓ Recover from it gracefully (**Recovery**).

Deadlock prevention

One way to handle deadlocks is to ensure that at least one of the four necessary conditions causing deadlocks is prevented by design.

Mutual Exclusion

- Mutual exclusion is necessary in any computer system because some resources such as memory, CPU, and dedicated devices must be exclusively allocated to one user at a time.
- In the case of I/O devices, such as printers, the mutual exclusion may be bypassed by spooling, which allows the output from many jobs to be stored in separate temporary spool files at the same time, and each complete output file is then selected for printing when the device is ready.

Hold and Wait

To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:

1. Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
2. Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request.
   This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.

No Preemption

Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.

1. If a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request.
2. When a resource is requested and not available, then the system looks to see what other processes currently have those resources and are them blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.

Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

It is bypassed by “hierarchical ordering” of resources. The solution was proposed by Havender and it is based on numbering resources and to require that processes request resources only in strictly increasing order.

In other words, in order to request resource Rj, a process must first release all Ri such that i >= j. One big challenge in this scheme is determining the relative ordering of the different resources

Advantage:

1. Jobs do not state the resource need in advance
2. But it should anticipate the order in which resource can be requested

Disadvantage:

1. Discovering the best order so that needs of majority of users are satisfied
2. Assigning numbers to non-physical resources (files)

DEADLOCK AVOIDANCE

The basic idea of deadlock avoidance is to grant only those requests for available resources, which cannot possibly results in deadlock.

A decision is made dynamically whether the current resource if allocated lead to deadlock.

- If possibly cannot, the resource is granted to the requesting process.
- Otherwise, the requesting process is suspended till the time when its pending request can be safely granted.

The two approaches followed for deadlock avoidance are:-

- Do not start the process if its demand might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation might result in deadlock.

Disadvantages

- It is not possible to know future resource requirement of process.
- Process can be blocked for long periods

BANKERS 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 need **'k'** instances of resource type $\mathbf{R_j}$ for its execution.
- 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 consists 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, 3, 4….n

2) Find an i such that both

a) Finish[i] = false

b) $Need_i$ <= Work

if no such i exists goto step (4)

3) Work = Work + Allocation[i]

Finish[i] = true

goto step (2)

4) if Finish [i] = true for all i

then the system is in a safe state

Example:

Considering a system with five processes P_0 through P_4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t_0 following snapshot of the system has been taken:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

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

So, the content of Need Matrix is:

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Applying the Safety algorithm on the given system,

m=3, n=5 — Step 1 of Safety Algo
Work = Available
Work = | 3 | 3 | 2 |
Finish = | false | false | false | false | false | (0 1 2 3 4)

For i = 0 — Step 2 ✗
$Need_0$ = 7, 4, 3
Finish [0] is false and $Need_0$ > Work (7,4,3 > 3,3,2)
So P_0 must wait. But Need ≤ Work

For i = 1 — Step 2 ✓
$Need_1$ = 1, 2, 2
Finish [1] is false and $Need_1$ < Work (1,2,2 < 3,3,2)
So P_1 must be kept in safe sequence

Step 3
Work = Work + $Allocation_1$ (3, 3, 2 + 2, 0, 0)
Work = | 5 | 3 | 2 | (A B C)
Finish = | false | true | false | false | false | (0 1 2 3 4)

For i = 2 — Step 2 ✗
$Need_2$ = 6, 0, 0
Finish [2] is false and $Need_2$ > Work (6, 0, 0 > 5, 3, 2)
So P_2 must wait

For i=3 — Step 2 ✓
$Need_3$ = 0, 1, 1
Finish [3] = false and $Need_3$ < Work (0, 1, 1 < 5, 3, 2)
So P_3 must be kept in safe sequence

Step 3
Work = Work + $Allocation_3$ (5, 3, 2 + 2, 1, 1)
Work = | 7 | 4 | 3 | (A B C)
Finish = | false | true | false | true | false | (0 1 2 3 4)

For i = 4 — Step 2 ✓
$Need_4$ = 4, 3, 1
Finish [4] = false and $Need_4$ < Work (4, 3, 1 < 7, 4, 3)
So P_4 must be kept in safe sequence

Step 3
Work = Work + $Allocation_4$ (7, 4, 3 + 0, 0, 2)
Work = | 7 | 4 | 5 | (A B C)
Finish = | false | true | false | true | true | (0 1 2 3 4)

For i = 0 — Step 2 ✓
$Need_0$ = 7, 4, 3
Finish [0] is false and Need < Work (7, 4, 3 < 7, 4, 5)
So P_0 must be kept in safe sequence

Step 3
Work = Work + $Allocation_0$ (7, 4, 5 + 0, 1, 0)
Work = | 7 | 5 | 5 | (A B C)
Finish = | true | true | false | true | true | (0 1 2 3 4)

For i = 2 — Step 2 ✓
$Need_2$ = 6, 0, 0
Finish [2] is false and $Need_2$ < Work (6, 0, 0 < 7, 5, 5)
So P_2 must be kept in safe sequence

Step 3
Work = Work + $Allocation_2$ (7, 5, 5 + 3, 0, 2)
Work = | 10 | 5 | 7 | (A B C)
Finish = | true | true | true | true | true | (0 1 2 3 4)

Step 4
Finish [i] = true for 0 ≤ i ≤ n
Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Disadvantage of Banker's algorithm

- Does not allow the process to change its Maximum need while processing
- It allows all requests to be granted in restricted time, but one year is a fixed period for that.
- All processes must know and state their maximum resource needs in advance.

DEADLOCK DETECTION

Deadlock can be detected by building directed resource graphs and looking for cycles.

Unlike the avoidance algorithm, which must be performed every time there is a request, the algorithm used to detect circularity can be executed whenever it is appropriate.

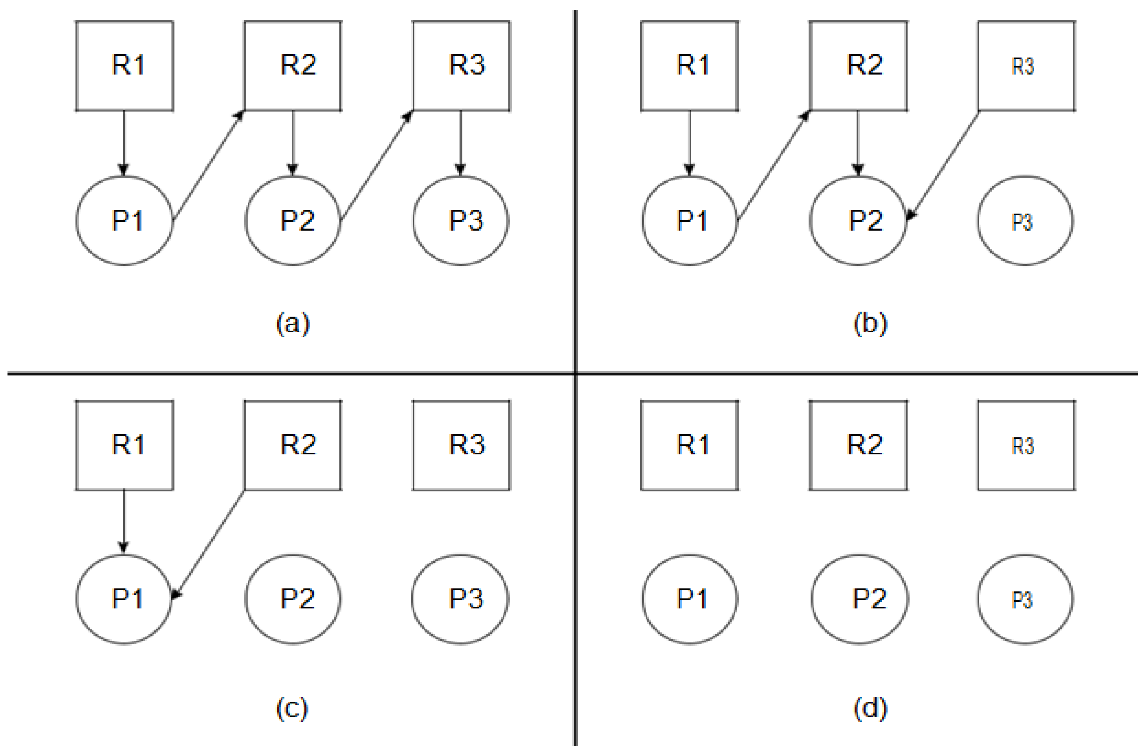
The detection algorithm can be explained by using directed resource graphs and "reducing" them. The steps to reduce a graph are these:

1. Find a process that is currently using a resource and *not waiting* for one. This process can be removed from the graph and the resource can be returned to the "available list."

2. Find a process that's waiting only for resource classes that aren't fully allocated. This process isn't contributing to deadlock since it would eventually get the resource it's waiting for, finish its work, and return the resource to the "available list".

3. Go back to step 1 and continue with steps 1 and 2 until all lines connecting resources to processes have been removed. If there are any lines left, this indicates that the request of the process in question can't be satisfied and that a deadlock exists.

Following figure illustrates a system in which three processes—P1, P2, and P3—and three resources—R1, R2, and R3—aren't deadlocked.



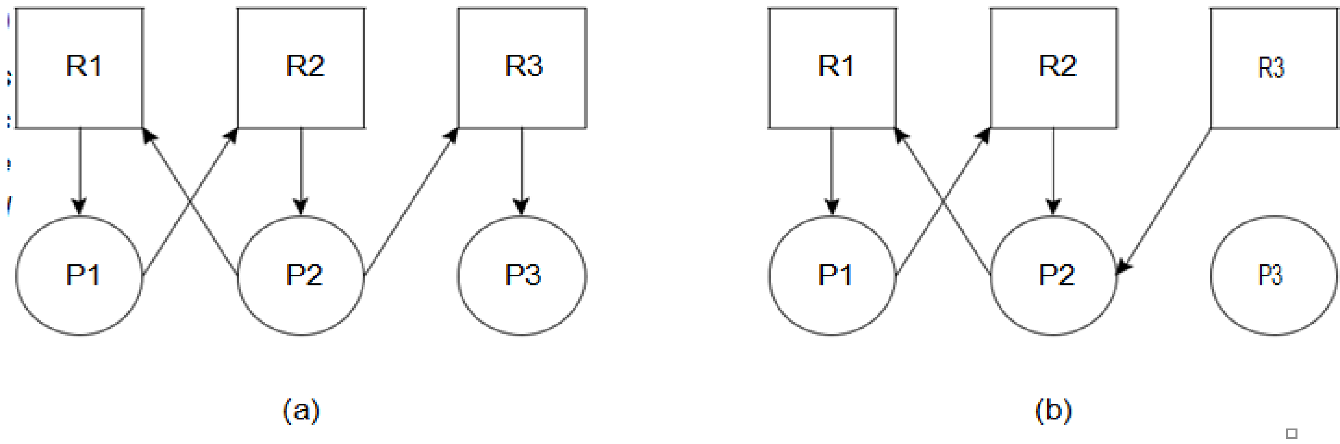
Above figure shows the stages of a graph reduction.

(b) the original state.
(c) In (b), the link between P3 and R3 can be removed because P3 isn't waiting for any other resources to finish, so R3 is released and allocated to P2 (step 1).

(d) In (c), the links between P2 and R3 and between P2 and R2 can be removed because P2 has all of its requested resources and can run to completion—and then R2 can be allocated to P1.

(e) Finally, in (d), the links between P1 and R2 and between P1 and R1 can be removed because P1 has all of its requested resources and can finish successfully. Therefore, the graph is completely resolved.

However, following figure shows a very similar situation that is deadlocked because of a key difference: P2 is linked to R1.



The deadlocked system in following Figure can't be reduced.

- In (a), the link between P3 and R3 can be removed because P3 isn't waiting for any other resource, so R3 is released and allocated to P2.
- But in (b), P2 has only two of the three resources it needs to finish and it is waiting for R1. But R1 can't be released by P1 because P1 is waiting for R2, which is held by P2; moreover, P1 can't finish because it is waiting for P2 to finish (and release R2), and P2 can't finish because it's waiting for R1. This is a circular wait.

RECOVERY FROM DEADLOCK

There are three basic approaches to recover from deadlock. They are

1. Terminate one or more processes involved in the deadlock
2. Preempt resources.

Process Termination

- Terminate every job that's active in the system and restart them from the beginning. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.

- Terminate only the jobs involved in the deadlock and ask their users to resubmit them. This is more conservative, but requires doing deadlock detection after each step.
- Identify which jobs are involved in the deadlock and terminate them one at a time, checking to see if the deadlock is eliminated after each removal, until the deadlock has been resolved. Once the system is freed, the remaining jobs are allowed to complete their processing and later the halted jobs are started again from the beginning.

Resource Preemption

These methods concentrate on the non-deadlocked jobs and the resources they hold.

- Selects a non-deadlocked job, preempts the resources it's holding, and allocates them to a deadlocked process so it can resume execution, thus breaking the deadlock.
- Stops new jobs from entering the system, which allows the non-deadlocked jobs to run to completion so they'll release their resources. Eventually, with fewer jobs in the system, competition for resources is curtailed so the deadlocked processes get the resources they need to run to completion.

When preempting resources to relieve deadlock, there are three important issues to be addressed:

1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the decision criteria
    - Process priorities.
    - How long the process has been running, and how close it is to finishing.
    - How many and what type of resources is the process holding.
    - How many more resources does the process need to complete.
    - How many processes will need to be terminated
    - Whether the process is interactive or batch.
    - Whether or not the process has made non-restorable changes to any resource.

2. **Rollback**
    - In this case of deadlock recovery through rollback, whenever a deadlock is detected, it is easy to see which resources are needed.
    - To do the recovery of deadlock, a process roll back to a safe state prior to the point at which that resource was originally allocated to the process.

Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning.

3. **Starvation**
   - In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.
   - To address this problem, an algorithm designed to detect starving jobs can be implemented, which tracks how long each job has been waiting for resources.
   - Once starvation has been detected, the system can block new jobs until the starving jobs have been satisfied. This algorithm must be monitored closely:
   - If monitoring is done too often, then new jobs will be blocked too frequently and throughput will be diminished. If it's not done often enough, then starving jobs will remain in the system for an unacceptably long period of time.

CONCURRENT PROCESSES

- Concurrent Processing is a form of Multiprocessing systems.
- Multiprocessing Systems have several processors working together in several distinctly different configurations as well as linked computing systems with only one processor each to share processing among them.
- In multiprocessing systems, the Processor Manager has to coordinate the activity of each processor, as well as synchronize cooperative interaction among the CPUs.

Parallel Processing

Parallel processing also a form of **Multiprocessing**, is a situation two or more CPUs are executing instructions simultaneously.

There are two primary benefits to parallel processing systems:

3. Increased reliability
4. Faster processing.

Increased Reliability

- Reliability is achieved because of availability of more than one CPU. If one processor fails, then the others can continue to operate and absorb the load.
- The failing processor can inform other processors to take over and the operating system can restructure its resource allocation strategies so the remaining processors don't become overloaded.
- Instructions can be processed in parallel in one of several ways.
    - Some systems allocate a CPU to each program or job.
    - Others allocate a CPU to each working set or parts of it.
    - Still others subdivide individual instructions so that each subdivision can be processed simultaneously (which is called concurrent programming).

Disadvantages:

- connect the processors into configurations
- co-ordinate their interaction

Evolution of Multiprocessors

Multiprocessing can take place at several different levels, each of which requires a different frequency of synchronization.

Parallelism Level	Process Assignments	Synchronization Required
Job Level	Each job has its own processor and all processes and threads are run by that same processor.	No explicit synchronization required.
Process Level	Unrelated processes, regardless of job, are assigned to any available processor.	Moderate amount of synchronization required to track processes.
Thread Level	Threads are assigned to available processors.	High degree of synchronization required, often requiring explicit instructions from the programmer.

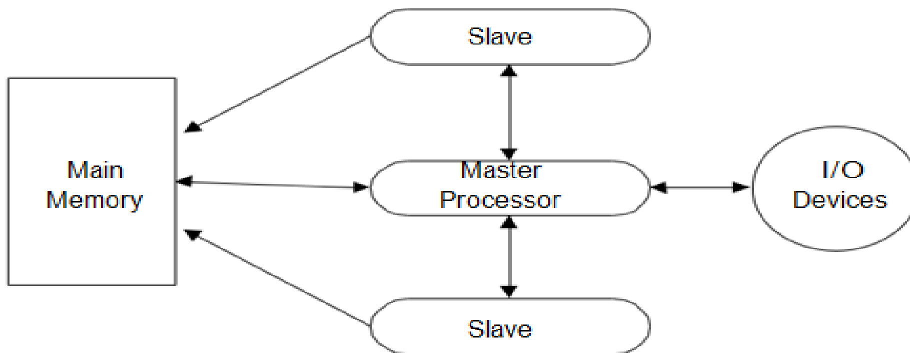
TYPICAL MULTIPROCESSING CONFIGURATIONS

Three typical configurations are: master/slave, loosely coupled, and symmetric.

Master/Slave Configuration

The **Master/Slave** configuration is an asymmetric multiprocessing system.

It is a single-processor system with additional slave processors, each of which is managed by the primary master processor as shown below



The master processor is responsible for managing the entire system: all files, devices, memory, and processors. It maintains the status of all processes in the system, performs storage management activities, schedules the work for the other processors, and executes all control programs.

This configuration is well suited for computing environments in which processing time is divided between front-end and back-end processors.

The front-end processor takes care of the interactive users and quick jobs, and the back-end processor takes care of those with long jobs using the batch mode.

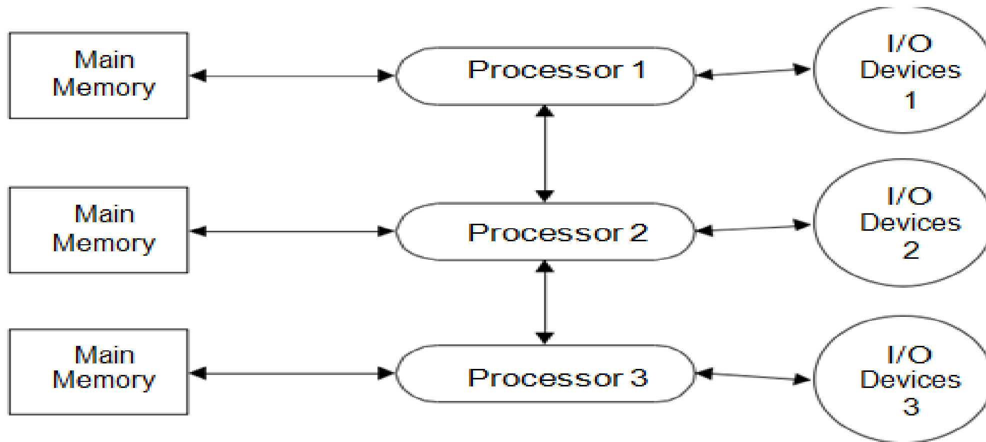
Advantage : Simplicity.

Disadvantages:

- If the master processor fails, the entire system fails. So it has less reliability
- Poor use of resources because slave processor is free but master processor is busy for most of time.
- The slave must wait until the master becomes free and can assign more work to it.

Loosely Coupled Configuration

The **loosely coupled configuration** features several complete computer systems, each with its own memory, I/O devices, CPU, and operating system. This configuration is called loosely coupled because each processor controls its own resources : its own files, access to memory, and its own I/O devices. Each processor maintains its own commands and I/O management tables.



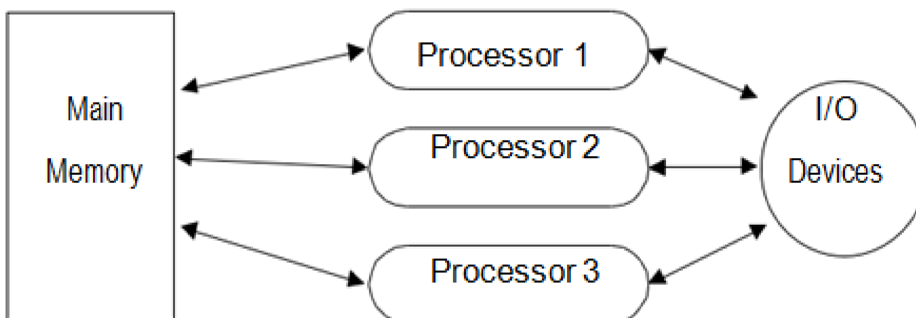
When a job arrives for the first time, it's assigned to one processor. Once allocated, the job remains with the same processor until it's finished. Each processor must have global tables that indicate to which processor each job has been allocated.

To keep the system well balanced and to ensure the best use of resources new jobs might be assigned to the processor with the lightest load or the best combination of output devices available.

This system isn't prone system failures because even when a single processor fails, the others can continue to work independently.

Symmetric configuration

The **symmetric configuration** (also called tightly coupled), processor scheduling is decentralized. A single copy of the operating system and a global table listing each process and its status is stored in a common area of memory so every processor has access to it. Each processor uses the same scheduling algorithm to select which process it will run next.



Advantages

- It's more reliable.
- It uses resources effectively.
- It can balance loads well.
- It can degrade gracefully in the event of a failure.

Disadvantages

- It is the most difficult configuration to implement because the processes must be well synchronized to avoid the problems of races and deadlocks.

Because each processor has access to all I/O devices and can reference any storage unit, there are more conflicts as several processors try to access the same resource at the same time.

This presents the need for algorithm to resolve conflicts between processors which is called **process synchronization**.

PROCESS SYNCHRONIZATION SOFTWARE

Several synchronization mechanisms are available to provide cooperation and communication among processes.

The common element in all synchronization schemes is to allow a process to finish work on a critical part of the program before other processes have access to it. This is applicable both to multiprocessors and to two or more processes in a single-processor (time-shared) processing system. It is called a **critical region** because it is a critical section and its execution must be handled as a unit.

Synchronization is sometimes implemented as a lock-and-key arrangement: Before a process can work on a critical region, it must get the key. And once it has the key, all other processes are locked out until it finishes, unlocks the entry to the critical region, and returns the key so that another process can get the key and begin work.

Several locking mechanisms have been developed, including

1. test-and-set
2. WAIT and SIGNAL
3. Semaphores.

Semaphores

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

Wait

The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

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

Signal

The signal operation increments the value of its argument S.

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

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows:

Counting Semaphores

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

Binary Semaphores

The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

Some of the advantages of semaphores are as follows:

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

Disadvantages of Semaphores

Some of the disadvantages of semaphores are as follows −

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