Advance Operating System

Sudhan Kandel
Sudhankandel03@gmail.com

Unit 1 Table of Content

1. Process Management and Synchronization 1

1.1 Process Management 1

1.1.1 Process States 1

1.1.2 Process Control Block (PCB) 2

1.1.3 Thread 4

1.1.4 Inter-process Communication 8

1.1.5 Race Condition: 9

1.1.6 Implementation Mutual Exclusion 11

1.2 Process Scheduling 16

1.2.1 Scheduler 17

1.2.2 Scheduling Criteria 18

1.2.3 Process Scheduling Queues 18

1.2.4 Types of Scheduling 19

1.2.5 Round-Robin Scheduling Algorithms 20

1.3 Deadlock 22

1.3.1 Condition for Deadlock 23

1.3.2 Deadlock Detection 25

1.3.3 Deadlock Prevention 28

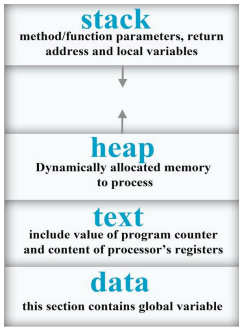
1.3.4 Banker's Algorithm 30

CHAPTER 1

1. Process Management and Synchronization

1.1 Process Management

- A process is an instance of a program running in a computer.
- It is close in meaning to task, a term used in some OS.
- In UNIX and some other OS, a process is started when a program is initiated (either by a user entering a shell command or by another program).
- A program by itself is not a process; a program is a ***passive entity,*** such as a file containing a list of instructions stored on disks. (often called an executable file)
- A program becomes a process when an executable file is loaded into memory and executed.
- When a program is loaded into a memory and it became a process , it can be divided into four sections :



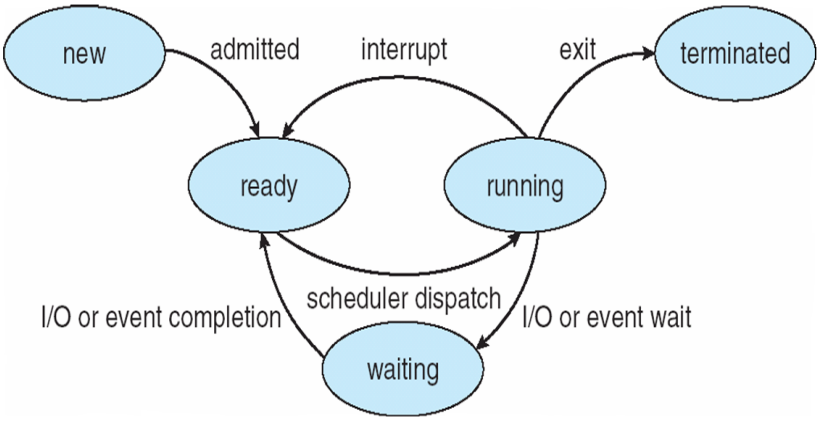
a. **Stack:** it contain temporary data such as method/ function parameters return address and local variables.
b. **Heap:** this is dynamically allocated memory to a process during its run time.
c. **Text:** this includes the current activity represented by the value of program counter and the contents of the processor's registers.
d. **Data**: This section contain the global and static variable

1.1.1 Process States

Processes in the operating system can be in any of the following states:

- **NEW-** The process is being created.
- **READY-** The process is waiting to be assigned to a processor.

- **RUNNING-** Instructions are being executed.
- **WAITING-** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **TERMINATED-** The process has finished execution.



1.1.2 Process Control Block (PCB)

Process Control Block (PCB, also called Task Controlling Block, Entry of the Process Table, Task Struck, or Switch frame) is a data structure in the operating system kernel containing the information needed to manage the scheduling of a particular process. The PCB is "the manifestation (expression) of a process in an operating system."

While creating a process the operating system performs several operations. To identify these process, it must identify each process, hence it assigns a process identification number (PID) to each process. As the operating system supports multi-programming, it needs to keep track of all the processes. For this task, the process control block (PCB) is used to track the process's execution status. Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another. When the process made transitions from one state to another, the operating system must update information in the process's PCB.

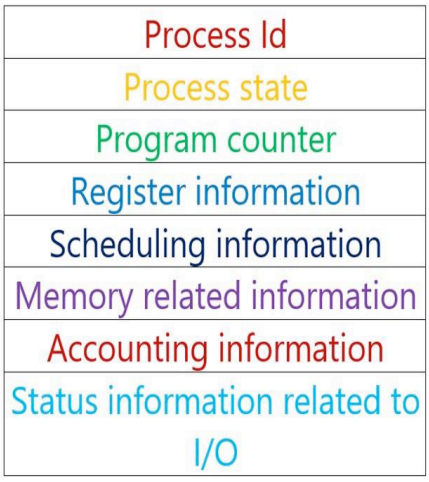
Role of PCB

The role or work of **process control block** (PCB) in process management is that it can access or modified by most OS utilities including those are involved with memory, scheduling, and input / output resource access. It can be said that the set of the process control blocks give the information of the current state of the operating system. Data structuring for processes is often done in terms of process control blocks. For example, pointers to other process control blocks inside any process control block allows the creation of those queues of processes in various scheduling states. The following are the various information that is contained by process control block:

a. Naming the process
b. State of the process
c. Resources allocated to the process
d. Memory allocated to the process
e. Scheduling information
f. Input / output devices associated with process

Components of PCB

The following are the various components that are associated with the process control block PCB:

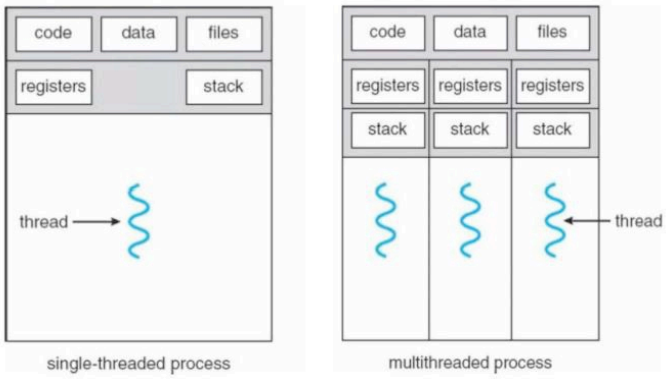


1. **Process ID:** In computer system there are various process running simultaneously and each process has its unique ID. This Id helps system in scheduling the processes. This Id is provided by the process control block. In other words, it is an identification number that uniquely identifies the processes of computer system.

2. **Process state:** As we know that the process state of any process can be New, running, waiting, executing, blocked, suspended, terminated. For more details regarding process states you can refer process management of an Operating System. Process control block is used to define the process state of any process. In other words, process control block refers the states of the processes.
3. **Program counter:** Program counter is used to point to the address of the next instruction to be executed in any process. This is also managed by the process control block.
4. **Register Information:** This information is comprising with the various registers, such as index and stack that are associated with the process. This information is also managed by the process control block.
5. **Scheduling information:** Scheduling information is used to set the priority of different processes. This is very useful information which is set by the process control block. In computer system there were many processes running simultaneously and each process have its priority? The priority of primary feature of RAM is higher than other secondary features. Scheduling information is very useful in managing any computer system.
6. **Memory related information:** This section of the process control block comprises of page and segment tables. It also stores the data contained in base and limit registers.
7. **Accounting information:** This section of process control block stores the details relate to central processing unit (CPU) utilization and execution time of a process.
8. **Status information related to input / output:** This section of process control block stores the details pertaining to resource utilization and file opened during the process execution.

1.1.3 Thread

A thread is the smallest unit of processing that can be performed in an OS. In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads. A thread is a basic unit of CPU utilization, it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files and signals. A traditional (or heavy weight) process has a single thread of control. If a process has multiple thread of control, it can perform more than one task at a time. Fig below illustrate the difference between single threaded process and a multithreaded process.



Properties of a Thread:

- Threads share data and information.
- Only one system call can create more than one thread (Lightweight process).
- Threads shares instruction, global and heap regions but has its own individual stack and registers.
- Thread management consumes no or fewer system calls as the communication between threads can be achieved using shared memory.
- The isolation property of the process increases its overhead in terms of resource consumption.

Types of Thread

There are two types of threads:

1. **User Level thread (ULT):** Is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause interrupt to Kernel. Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.

   Advantages of ULT:

   - Can be implemented on an OS that doesn't support multithreading.
   - Simple representation since thread has only program counter, register set, stack space.
   - Simple to create since no intervention of kernel.
   - Thread switching is fast since no OS calls need to be made.

Disadvantages of ULT:

- No or less co-ordination among the threads and Kernel.
- If one thread causes a page fault, the entire process blocks.

2. **Kernel Level Thread (KLT):** Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.

   Advantages of KLT:

   - Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
   - Good for applications that frequently block.

   Disadvantages of KLT:

   - Slow and inefficient.
   - It requires thread control block so it is an overhead.

Multithreading

Many software package that run on modern desktop pcs are multithreaded. An application is implemented as a separate process with several threads of control. A web browser might have one thread to display images or text while other thread retrieves data from the network. A word-processor may have a thread for displaying graphics, another thread for reading the character entered by user through the keyboard, and a third thread for performing spelling and grammar checking in the background.

Why Multithreading

In certain situations, a single application may be required to perform several similar task such as a web server accepts client requests for web pages, images, sound, graphics etc. A busy web server may have several clients concurrently accessing it. So if the web server runs on traditional single threaded process, it would be able to service only one client at a time. The amount of time that the

client might have to wait for its request to be serviced is enormous. One solution of this problem can be thought by creation of new process

When the server receives a new request, it creates a separate process to service that request. But this method is heavy weight. In fact this process creation method was common before threads become popular. Process creation is time consuming and resource intensive. It is generally more efficient for one process that contains multiple threads to serve the same purpose. This approach would multithread the web server process. The server would create a separate thread that would listen for client's requests. When a request is made by client, rather than creating another process, server will create a separate thread to service the request.

Benefits of Multi-threading:

a. **Responsiveness:** Multithreaded interactive application continues to run even if part of it is blocked or performing a lengthy operation, thereby increasing the responsiveness to the user.
b. **Resource Sharing:** By default, threads share the memory and the resources of the process to which they belong. It allows an application to have several different threads of activity within the same address space. These threads running in the same address space do not need a context switch.
c. **Economy:** Allocating memory and resources for each process creation is costly. Since thread shares the resources of the process to which they belong, it is more economical to create and context switch threads. Shorter context switching time. Less overhead than running several processes doing the same task.
d. **Utilization of multiprocessor architecture:** The benefits of multi-threading can be greatly increased in multiprocessor architecture, where threads may be running in parallel on different processors. Multithreading on a multi-CPU increases concurrency.

Multithreading Model

The user threads must be mapped to kernel threads, by one of the following strategies:

- Many to One Model
- One to One Model
- Many to Many Model

1.1.4 Inter-process Communication

IPC is a mechanism that allows the exchange of data between processes. Processes frequently needs to communicate with each other. For example, the output of the first process must be passed to the second process and so on. Thus there is a need for communication between the processes, preferably in a well-structured way not using the interrupts. IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. Inter-process communication (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. Processes executing concurrently in the operating system may be either independent process or co-operating process**.**

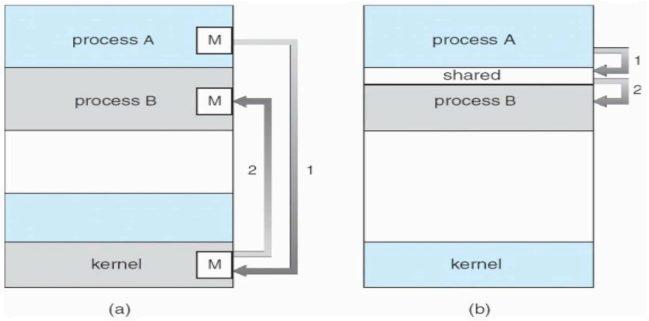
- **Independent process:** A process is independent if it can't affect or be affected by another process.
- **Co-operating Process:** A process is co-operating if it can affects other or be affected by the other process. Any process that shares data with other process is called co-operating process.

Reasons for providing an environment for process co-operation:

1. **Information sharing:** Several users may be interested to access the same piece of information (for instance a shared file). We must allow concurrent access to such information.
2. **Computation Speedup:** To run the task faster we must breakup tasks into sub-tasks. Such that each of them will be executing in parallel to other, this can be achieved if there are multiple processing elements.
3. **Modularity:** Construct a system in a modular fashion which makes easier to deal with individual.
4. **Convenience:** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

There are two fundamental ways of IPC.

a. Message Passing
b. Shared Memory



a. **Message Passing:** Communication takes place by means of messages exchanged between the co-operating process Message passing is useful for exchanging the smaller amount of data. Easier to implement than shared memory. Slower than that of Shared memory as message passing system are typically implemented using system call Which requires more time consuming task of Kernel intervention.
b. **Shared Memory:** Here a region of memory that is shared by co-operating process is established. Process can exchange the information by reading and writing data to the shared region**.** Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer. System calls are required only to establish shared memory regions. Once shared memory is established no assistance from the kernel is required, all access are treated as routine memory access.

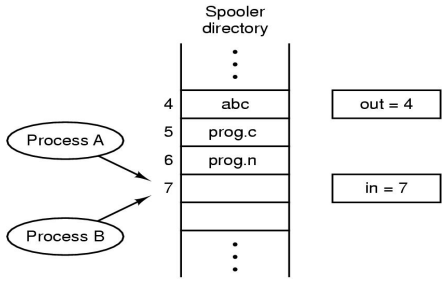
1.1.5 Race Condition:

The situation where 2 or more processes are reading or writing some shared data, but not in proper sequence is called race Condition**.** The final results depends on who runs precisely (accurately) when**.**

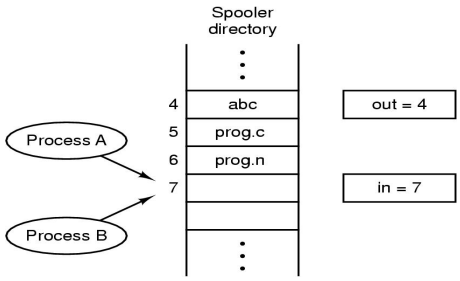
Example, a print spooler.

- When any process wants to print a file, it enters the file name in a special **spooler directory.**

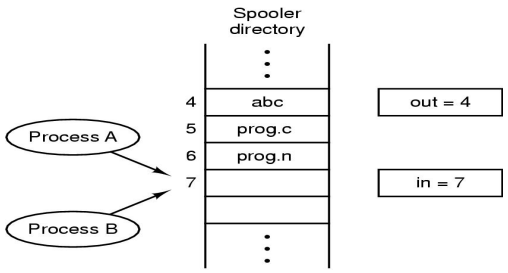
- Another process, the **printer daemon, periodically checks to see if** there are any files to be printed, and if there are, it prints them and removes their names from the directory.
- Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables,
- Out: which points to the next file to be printed
- In: which points to the next free slot in the directory.



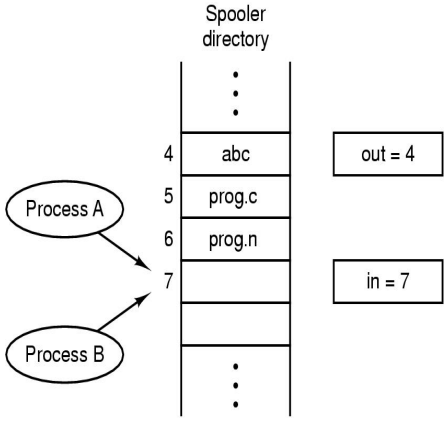
- At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed).
- More or less simultaneously, processes A and B decide they want to queue a file for printing as shown in the fig.
- Process A reads in and stores the value, 7, in a local variable called **next_free_slot.**



- Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.
- Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things.



- Eventually, process a runs again, starting from the place it left off last time. It looks at **next_free_slot, finds a 7 there, and writes its file** name in slot 7, erasing the name that process B just put there.
- Then it computes **next_free_slot + 1, which is 8, and sets in to 8.**
- The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.



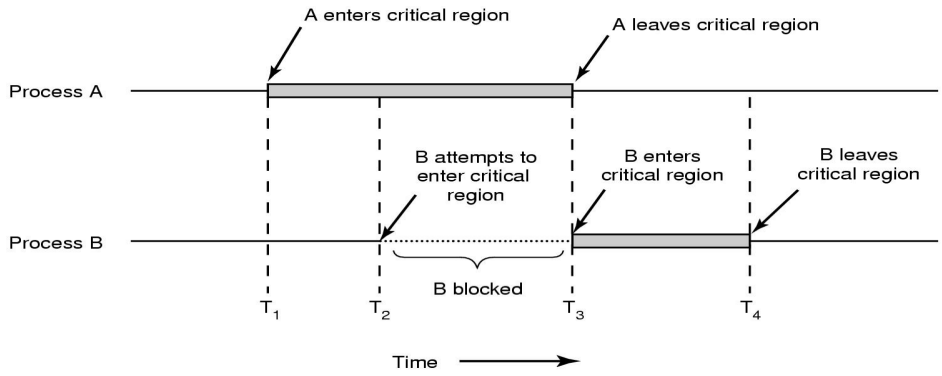
1.1.6 Implementation Mutual Exclusion

IPC: Critical Regions and solution

The part of the program where the shared memory is accessed that must not be concurrently accessed by more than one processes is called the critical region

Four conditions to provide mutual exclusion Or (Rules for avoiding Race Condition) Solution to Critical section problem:

- No two processes simultaneously in critical region
- No assumptions made about speeds or numbers of CPUs
- No process running outside its critical region may block another process
- No process must wait forever to enter its critical region



Mutual exclusion using critical regions

Busy waiting

- Continuously testing a variable until some value appears is called busy waiting.
- If the entry is allowed it execute else it sits in tight loop and waits.
- Busy-waiting: consumption of CPU cycles while a thread is waiting for a lock
  - Very inefficient
  - Can be avoided with a waiting queue

How to Manage Race Condition

a. Mutual Exclusion with Busy Waiting
   - Lock Variable

b. Mutual Exclusion without Busy Waiting
   - Sleep and wakeup
   - Semaphore

I. Lock Variable

- Integer variable turn is initially 0.
- It keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initially, process 0 inspects turn, finds it to be 0, and enters its critical region.
- Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
- Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided, since it wastes CPU time**.**

- A lock that uses busy waiting is called a spin lock. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region.
- This way no two process can enters critical region simultaneously i.e mutual exclusion is fulfilled.

Drawbacks:

- Taking turn is not a good idea when one of the process is much slower than other.
- This situation requires that two processes strictly alternate in entering their critical region.

Example:

- Process 0 finishes the critical region it sets turn to 1 to allow process 1 to enter critical region.
- Suppose that process 1 finishes its critical region quickly so both process are in their non-critical region with turn sets to 0.
- Process 0 executes its whole loop quickly, exiting its critical region & setting turn to 1.
- At this point turn is 1 and both processes are executing in their noncritical regions.
- Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop.
- Unfortunately, it is not permitted to enter its critical region now since turn is 1 and process 1 is busy with its noncritical region.
- This situation violates the condition 3 set above: No process running outside the critical region may block other process.
- The concept of mutual exclusion is fulfilled here, but the concept of progress is not fulfilled.
- The process who is not interested to enter the critical section is blocking the next process to enter the critical section.

II. Sleep and Wakeup

Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened e.g. Wakeup (consumer) or wakeup (producer).

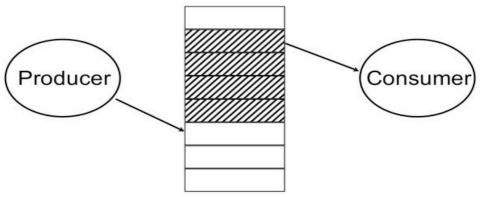
When a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not allowed, the process just sits in a tight loop waiting until it is allowed. Beside of wasting CPU time, this approach can also have unexpected effects. Consider a computer with two processes, *H*, with high priority and *L*, with low priority. The scheduling rules are such that *H* runs whenever it is in ready state. At a certain moment, with *L* is in its critical region, *H* becomes ready to run. *H* now begins busy waiting. Before *H* is completed, *L* cannot be scheduled. So *L* never gets the chance to leave its critical region, so *H* loops forever.

Now let me look at some interposes communication primitives that block processes when they are not allowed to enter their critical regions, instead of wasting CPU time in an empty loop. One of the simplest primitives is the pair **sleep** and **wakeup**. Sleep is a system call that causes the caller to block, the caller is suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

Examples:

Producer-consumer problem (Bounded Buffer):

- Two processes share a common, fixed-size buffer.
- One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out



Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.

Solution

- Producer goes to sleep and to be awakened when the consumer has removed data.

2. The consumer wants to remove data from the buffer but buffer is already empty.

Solution

- Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

III. Semaphores

- Semaphore is an integer variable to count the number of wakeup processes saved for future use.
- A semaphore could have the value 0, indicating that no wakeup processes were saved, or some positive value if one or more wakeups were pending.
- There are two operations, down (wait) and up (signal) (generalizations of sleep and wakeup, respectively).
- The down operation on a semaphore checks if the value is greater than 0. If so, it decrements the value and just continues.
- If the value is 0, the process is put to sleep without completing the down for the moment.
- The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down.
- Checking the value, changing it and possibly going to sleep is as a single indivisible **atomic action**. It is guaranteed that when a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

Atomic operations:

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in case of the P(S) operation the testing of the integer value of S (S<=0) and its possible modification (S=S-1), must also be executed without interruption.
- Modification to the integer value of the semaphore in the wait {p(s)} and signal{V(s)} operation must be executed indivisibly(only one process can modify the same semaphore value at a time)

Semaphore operations:

- P or Down, or Wait: P stands for *proberen ("to test")*
- V or Up or Signal: Dutch words. V stands for *verhogen ("increase")*

Wait (sem)

- Decrement the semaphore value. If negative, suspend the process and place in queue. (Also referred to as *P (), down in literature.)*

Signal (sem)

- Increment the semaphore value, allow the first process in the queue to continue. (Also referred to as *V (), up in literature.)*

Advantages of semaphores

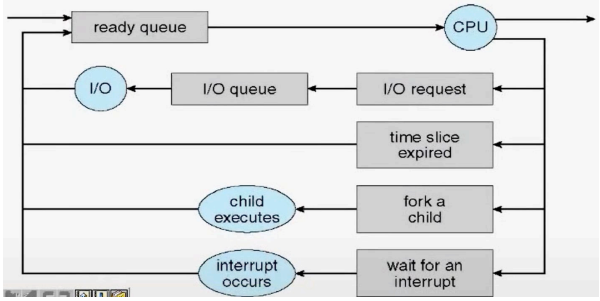
- Processes do not busy wait while waiting for resources.
- While waiting, they are in a "suspended" state, allowing the CPU to perform other work.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

Disadvantage of semaphores

- Can only be invoked by processes--not interrupt service routines because interrupt routines cannot block
- User controls synchronization--could mess up

1.2 Process Scheduling

In a multiprogramming system, frequently multiple process competes for the CPU at the same time. When two or more process are simultaneously in the ready state a choice has to be made which process is to run next. This part of the OS is called Scheduler and the algorithm is called scheduling algorithm. Process execution consists of cycles of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst that is followed by I/O burst, which is followed by another CPU burst then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.



1.2.1 Scheduler

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. There are three types of Scheduler:

1. Long term (job) scheduler
2. Medium term scheduler
3. Short term (CPU) scheduler

1. The long-term scheduler

Due to the smaller size of main memory initially all program are stored in secondary memory. When they are stored or loaded in the main memory they are called process. This is the decision of long term scheduler that how many processes will stay in the ready queue. Hence, in simple words, long term scheduler decides the degree of multi-programming of system.

2. Medium term scheduler

Most often, a running process needs I/O operation which doesn't requires CPU. Hence during the execution of a process when an I/O operation is required then the operating system sends that process from running queue to blocked queue. When a process completes its I/O operation then it should again be shifted to ready queue. ALL these decisions are taken by the medium-term scheduler. Medium-term scheduling is a part of swapping.

3. Short term (CPU) scheduler

When there are lots of processes in main memory initially all are present in the ready queue. Among all of the process, a single process is to be selected for execution. This decision is handled

by short term scheduler. Let's have a look at the figure given below. It may make a more clear view for you.

1.2.2 Scheduling Criteria

Many criteria have been suggested for comparison of CPU scheduling algorithms.

1. **CPU utilization**: we have to keep the CPU as busy as possible. It may range from 0 to 100%. In a real system it should range from 40 – 90 % for lightly and heavily loaded system.
2. **Throughput:** It is the measure of work in terms of number of process completed per unit time. E.g. For long process this rate may be 1 process per hour, for short transaction, throughput may be 10 process per second.
3. **Turnaround Time:** It is the sum of time periods spent in waiting to get into memory, waiting in ready queue, execution on the CPU and doing I/O. The interval form the time of submission of a process to the time of completion is the turnaround time. Waiting time plus the service time.
    - **Turnaround time**= Time of completion of job - Time of submission of job. (waiting time + service time or burst time)
4. **Waiting time:** It's the sum of periods waiting in the ready queue.
5. **Response time:** In interactive system the turnaround time is not the best criteria. Response time is the amount of time it takes to start responding, not the time taken to output that response.

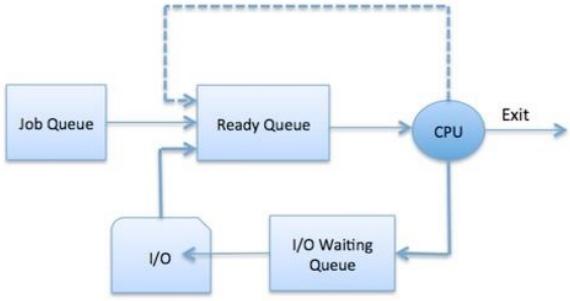
1.2.3 Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue:** This queue keeps all the processes in the system.
- **Ready queue:** This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.

- **Device queues:** The processes which are blocked due to unavailability of an I/O device constitute this queue.



Two-State Process Model

Two-state process model refers to running and non-running states which are described below

1. **Running:** When a new process is created, it enters into the system as in the running state.
2. **Not running:** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

1.2.4 Types of Scheduling

1. **Preemptive Scheduling**

- Preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time.
- If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available).
- Doing preemptive scheduling requires having a clock interrupt occur at the end of time interval to give control of the CPU back to the scheduler.
- Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state.

- The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.
- Algorithms based on preemptive scheduling are: Round Robin (RR), Shortest Job First (SJF basically non preemptive) and Priority (non preemptive version), etc.

2. Non preemptive Scheduling

- Non-preemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU.
- Even it runs for hours, it will not be forcibly suspended.
- Non-preemptive Scheduling is used when a process terminates, or a process switches from running to waiting state.
- In this scheduling, once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.
- Algorithms based on preemptive scheduling are: Shortest Remaining Time First (SRTF), Priority (preemptive version), etc.

1.2.5 Round-Robin Scheduling Algorithms

Round Robin (RR) scheduling is a preemptive algorithm that relates the process that has been waiting the longest. This is one of the oldest, simplest and widely used algorithms. The round robin scheduling algorithm is primarily used in time-sharing and a multi-user system environment where the primary requirement is to provide reasonably good response times and in general to share the system fairly among all system users. Basically the CPU time is divided into time slices.

Each process is allocated a small time-slice called quantum. No process can run for more than one quantum while others are waiting in the ready queue. If a process needs more CPU time to complete after exhausting one quantum, it goes to the end of ready queue to await the next allocation. To implement the RR scheduling, Queue data structure is used to maintain the Queue

of Ready processes. A new process is added at the tail of that Queue. The CPU scheduler picks the first process from the ready Queue, Allocate processor for a specified time Quantum. After that time the CPU scheduler will select the next process is the ready Queue.

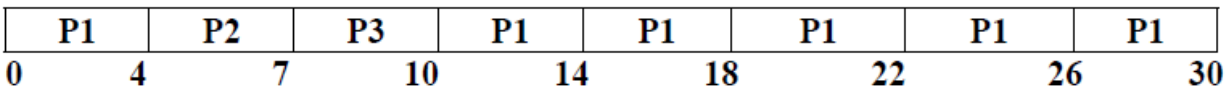
One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR). In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum. If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list. If the process has blocked or finished before the quantum has elapsed the CPU switching is done. Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in timesharing environments in which the system needs to guarantee reasonable response times for interactive users.

The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS. In any event, the average waiting time under round robin scheduling is on quite long.

Consider the following set of process with the processing time given in milliseconds.

Process	Processing Time
P1	24
P2	03
P3	03

If we use a time **Quantum of 4 milliseconds** then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time Quantum, and the CPU is given to the next process in the Queue, Process P2. Since process P2 does not need and milliseconds, it quits before its time Quantum expires. The CPU is then given to the next process, Process P3 one each process has received 1 time Quantum, the CPU is returned to process P1 for an additional time quantum. The Gantt chart will be:



Process	Processing Time	Turn around time = t(Process Completed) – t(Process Submitted)	Waiting Time = Turn around time – Processing time
P1	24	30 – 0 = 30	30 – 24 = 6
P2	03	7 – 0 = 7	7 – 3 = 4
P3	03	10 – 0 =10	10 – 3 = 7

Average turnaround time = (30+7+10)/3 = 47/3 = 15.66
Average waiting time = (6+4+7)/3 = 17/3 = 5.66
Throughput = 3/30 = 0.1
Processor utilization = (30/30) * 100 = 100%

1.3 Deadlock

In a computer system, we have a finite number of *resources* to be distributed among a number of competing processes. These system resources are classified in several types which may be either physical or logical. Examples of physical resources are Printers, Tape drivers, Memory space, and CPU cycles. Examples of logical resources are Files, Semaphores and Monitors. Each resource type can have some identical instances.

A process must request a resource before using it and release the resource after using it. It is clear that the number of resources requested cannot exceed the total number of resources available in the system.

In a normal operation, a process may utilize a resource only in the following sequence:

- ***Request***: if the request cannot be immediately granted, then the requesting process must wait until it can get the resource.
- ***Use****:* the requesting process can operate on the resource.
- ***Release****:* the process releases the resource after using it.

Examples for request and release of system resources are:

- Request and release the device,
- Opening and closing file,
- Allocating and freeing the memory.

A **deadlock** is a situation where a group of processes is permanently blocked as a result of each process having acquired a set of resources needed for its completion and having to wait for the release of the remaining resources held by others thus making it impossible for any of the deadlocked processes to proceed.

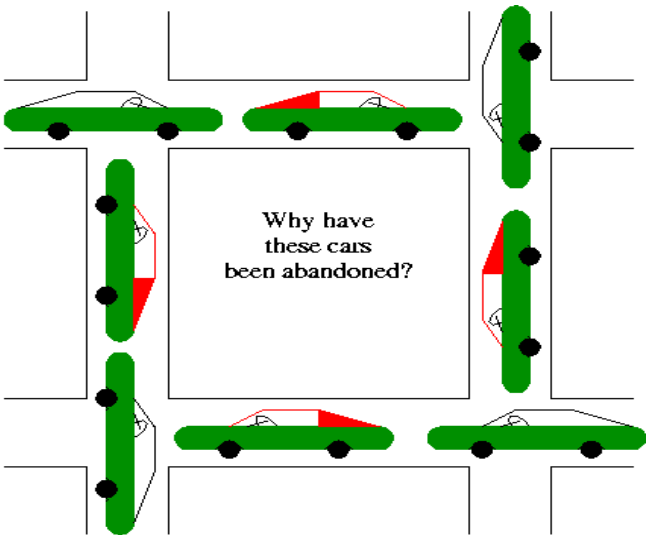
The operating system is responsible for making sure that the requesting process has been allocated the resource. A system table indicates if each resource is free or allocated, and if allocated, to which process. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

1.3.1 Condition for Deadlock

Coffman (1971) identified **four necessary conditions** that must hold simultaneously for a deadlock to occur.

1. **Mutual Exclusion Condition:** The resources involved are non-shareable. At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and Wait Condition:** In this condition, a requesting process already holds resources and waiting for the requested resources. A process, holding a resource allocated to it waits for an additional resource(s) that is/are currently being held by other processes.
3. **No-Preemptive Condition:** Resources already allocated to a process cannot be preempted. Resources cannot be removed forcibly from the processes. After completion, they will be released voluntarily by the process holding it.
4. **Circular Wait Condition:** The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

Let us understand this by a common example. Consider the traffic deadlock shown in the *Figure*



Consider each section of the street as a resource. In this situation:

- Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
- Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
- Non-preemptive condition applies, since a section of the street that is occupied by a vehicle cannot be taken away from it.
- Circular wait condition applies, since each vehicle is waiting for the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of the street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection. It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread, that is, each thread has access to the resources held by the process.

1.3.2 Deadlock Detection

There are two ways to detect the deadlock which is discussed below.

1. Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case: only one resource of each type exists. Such a system might have one scanner, one CD recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment. We will treat them later using a different method.

For such a system, we can construct a resource graph of the sort illustrated in Fig. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

As an example of a more complex system than the ones we have looked at so far, consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:

- Process *A* holds *R* and wants *S*.
- Process *B* holds nothing but wants *T*.
- Process *C* holds nothing but wants *S*.
- Process *D* holds *U* and wants *S* and *T*.
- Process *E* holds *T* and wants *V*.
- Process *F* holds *W* and wants *S*.
- Process *G* holds *V* and wants *U*.

The question is: "Is this system deadlocked, and if so, which processes are involved?"

To answer this question, we can construct the resource graph of Fig.(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig.(b). From this cycle, we can see that processes *D*, *E*, and *G* are all deadlocked. Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.

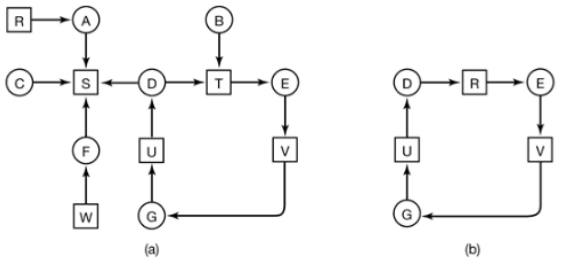


Fig: (a) A resource graph. (b) A cycle extracted from (a).

2. Deadlock Detection with Multiple Resources of Each Type

When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock among n Processes. P_1 through P_n. Let the number of resource classes be m, with E_1 resources of class 1, E_2 resources of class 2, and generally, E_i resources of class i ($1 < i < m$). E is the existing resource vector. It gives the total number of instances of each resource in existence. For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives. At any instant, some of the resources are assigned and are not available. Let A be the available resource vector, with A_i giving the number of instances of resource i that are currently available (i.e., unassigned). If both of our two tape drives are assigned, A_1 will be 0. Now we need two arrays, C, the current allocation matrix, and R, the request matrix. The i-th row of C tells how many instances of each resource class P_i currently holds. Thus C_{ij} is the number of instances of resource j that are held by process i. Similarly, R_{ij} is the number of instances of resource j that P_i wants. These four data structures are shown in Fig.

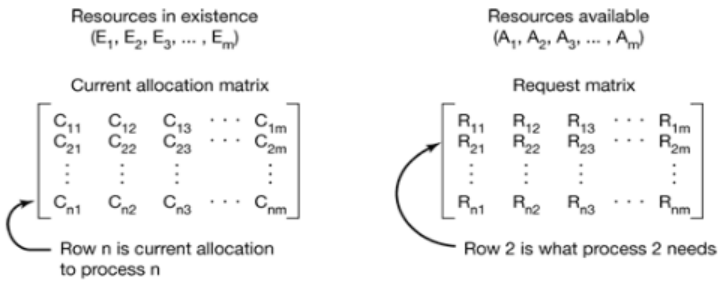


Fig: The four data structures needed by the deadlock detection algorithm

The deadlock detection algorithm is based on comparing vectors. Let us define the relation $A \leq B$ on two vectors A and B to mean that each element of A is less than or equal to the corresponding

element of B. Mathematically, $A \leq B$ holds if and only if $A_i \leq B_i$ for $1 < i < m$. Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked.

The deadlock detection algorithm can now be given, as follows.

1. Look for an unmarked process, P_i, for which the i-th row of R is less than or equal to A.
2. If such a process is found, add the i-th row of C to A, mark the process, and go back to step
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked. What the algorithm is doing in step 1 is looking for a process that can be run to completion. Such a process is characterized as having resource demands that can be met by the currently available resources. The selected process is then run until it finishes, at which time it returns the resources it is holding to the pool of available resources. It is then marked as completed. If all the processes are ultimately able to run, none of them are deadlocked. If some of them can never run, they are deadlocked. Although the algorithm is nondeterministic (because it may run the processes in any feasible order), the result is always the same.

As an example of how the deadlock detection algorithm works, consider Fig. Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanner, and CD-ROM drive. Process 1 has one scanner. Process 2 has two tape drives and a CD-ROM drive. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the R matrix.

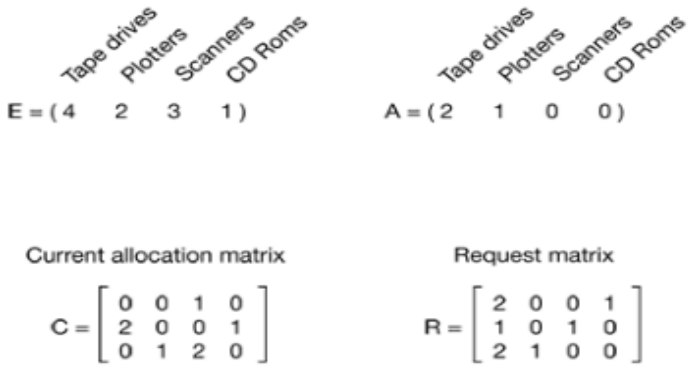


Fig: An example for the deadlock detection algorithm.

1.3.3 Deadlock Prevention

Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions. Let us study Havender's algorithm.

Havender's Algorithm

Elimination of "Mutual Exclusion" Condition

The mutual exclusion condition must hold for non-shareable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non- shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of "Hold and Wait" Condition

There are two possibilities for the elimination of the second condition. The first alternative is that a process request be granted all the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources.

For example, a program requiring ten tap drives must request and receive all ten drives before it begins executing. If the program needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours then substantial computer resources (9 tape drives) will sit idle for several hours. This strategy can cause indefinite postponement (starvation), since not all the required resources may become available at once.

Elimination of "No-preemption" Condition

The no preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated, to relinquish all its currently held resources, so that

other processes may use them to finish their needs. Suppose a system does allow processes to hold resources while requesting additional resources.

Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed, while the second process may hold the resources needed by the first process. This is a deadlock. This strategy requires that when a process that is holding some resources is denied a request for additional resources, the process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively.

Elimination of "Circular Wait" Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all the resource types and then forcing all processes to request the resources in order (increasing or decreasing). This strategy imposes a total ordering of all resource types and requires that each process requests resources in a numerical order of enumeration.

With this rule, the resource allocation graph can never have a cycle. For example, provide a global numbering of all the resources, as shown in the given Table

Number	Resource
1	Floppy drive
2	Printer
3	Plotter
4	Tape Drive
5	CD Drive

Now we will see the rule for this:

Rule: Processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may

not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

This strategy, if adopted, may result in low resource utilization and in some cases, starvation is possible too.

1.3.4 Banker's Algorithm

This approach to the deadlock problem anticipates a deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and act accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock. The most famous deadlock avoidance algorithm, from Dijkstra [1965], is the Banker's algorithm. It is named as Banker's algorithm because the process is analogous to that used by a banker in deciding if a loan can be safely made a not.

The Banker's Algorithm is based on the banking system, which never allocates its available cash in such a manner that it can no longer satisfy the needs of all its customers. Here we must have the advance knowledge of the maximum possible claims for each process, which is limited by the resource availability. During the run of the system we should keep monitoring the resource allocation status to ensure that no circular wait condition can exist.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. The following are the features that are to be considered for avoidance of the deadlock s per the Banker's Algorithms.

- Each process declares maximum number of resources of each type that it may need.
- Keep the system in a safe state in which we can allocate resources to each process in some order and avoid deadlock.
- Check for the safe state by finding a safe sequence: <P1, P2, ..., Pn> where resources that Pi needs can be satisfied by available resources plus resources held by Pj where j < i.
- Resource allocation graph algorithm uses claim edges to check for a safe state.

The resource allocation state is now defined by the number of available and allocated resources, and the maximum demands of the processes. Subsequently the system can be in either of the following states:

- Safe state: Such a state occurs when the system can allocate resources to each process (up to its maximum) in some order and avoid a deadlock. This state will be characterized by a safe sequence. It must be mentioned here that we should not falsely conclude that all unsafe states are deadlocked although it may eventually lead to a deadlock.
- Unsafe State: If the system did not follow the safe sequence of resource allocation from the beginning and it is now in a situation, which may lead to a deadlock, then it is in an unsafe state.
- Deadlock State: If the system has some circular wait condition existing for some processes, then it is in deadlock state.

Let us study this concept with the help of an example as shown below:

Consider an analogy in which 4 processes (P1, P2, P3 and P4) can be compared with the customers in a bank, resources such as printers etc. as cash available in the bank and the Operating system as the Banker.

Processes	Resources used	Maximum resources
P1	0	6
P2	0	5
P3	0	4
P4	0	7

Let us assume that total available resources = 10

In the above table, we see four processes, each of which has been granted a number of maximum resources that can be used. The Operating system reserved only 10 resources rather than 22 units to service them. At a certain moment, the situation becomes:

Processes	Resources used	Maximum resources
P1	1	6
P2	1	5
P3	2	4
P4	4	7

Available resources = 2

Safe State: The key to a state being safe is that there is at least one way for all users to finish. In other words the state of *Table* is safe because with 2 units left, the operating system can delay any request except P3, thus letting P3 finish and release all four resources. With four units in hand, the Operating system can let either P4 or P2 have the necessary units and so on.

Unsafe State: Consider what would happen if a request from P2 for one more unit was granted in *Table*. We would have following situation as shown in *Table*.

Processes	Resources used	Maximum resources
P1	0	6
P2	0	5
P3	0	4
P4	0	7

Available resource = 1

This is an unsafe state.

If all the processes request for their maximum resources respectively, then the operating system could not satisfy any of them and we would have a deadlock.

Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence of a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus used to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it is postponed until later. Haberman [1969] has shown that executing of the algorithm has a complexity proportional to *N2* where *N* is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant.

Limitations of the Banker's Algorithm

- There are some problems with the Banker's algorithm as given below: It is time consuming to execute on the operation of every resource.
- If the claim information is not accurate, system resources may be underutilised.
- Another difficulty can occur when a system is heavily loaded. Lauesen states that in this situation "so many resources are granted away that very few safe sequences remain, and as a consequence, the jobs will be executed sequentially". Therefore, the Banker's algorithm is referred to as the "Most Liberal" granting policy; that is, it gives away everything that it can without regard to the consequences.
- New processes arriving may cause a problem.
    - The process's claim must be less than the total number of units of the resource in the system. If not, the process is not accepted by the manager.
    - Since the state without the new process is safe, so is the state with the new process. Just use the order you had originally and put the new process at the end.
    - Ensuring fairness (starvation freedom) needs a little more work but isn't too hard either (once every hour stop taking new processes until all current processes finish).
- A resource becoming unavailable (e.g., a tape drive breaking), can result in an unsafe state.