OS – Unit 2
Processes, Scheduling, Deadlocks, Sync'n, and CSM

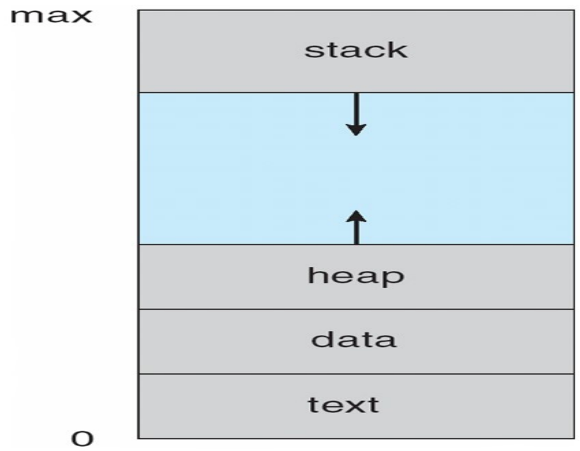
Process Concept

- An OS executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

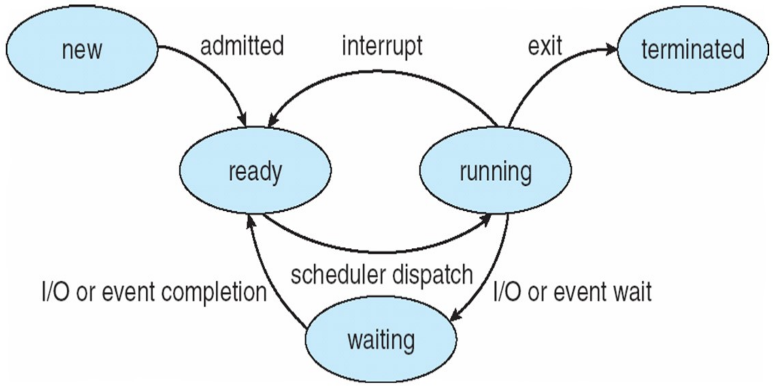
Process in Memory



Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

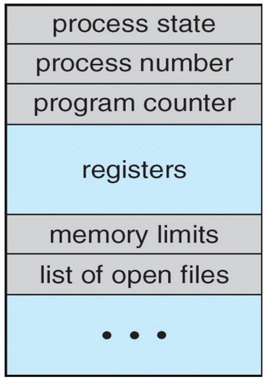


Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



PCB

- While creating a process the OS performs several operations. To identify the processes, it assigns a process identification number (PID) to each process. As the OS supports multi-programming, it needs to keep track of all the processes. For this task, the 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 this information is required and must be saved when the process is switched from one state to another. When the process makes a transition from one state to another, the OS must update information in the process's PCB. A PCB contains information about the process, i.e. registers, quantum, priority, etc. The process table is an array of PCBs, that means logically contains a PCB for all of the current processes in the system.

PCB parts

- **Pointer –** It is a stack pointer which is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process state –** It stores the respective state of the process.
- **Process number –** Every process is assigned with a unique id known as process ID or PID which stores the process identifier.
- **Program counter –** It stores the counter which contains the address of the next instruction that is to be executed for the process.
- **Register –** These are the CPU registers which includes: accumulator, base, registers and general purpose registers.
- **Memory limits –** This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.
- **Open files list –** This information includes the list of files opened for a process.

Adv's of PCB

- **Efficient process management:** The PT and PCB provide an efficient way to manage processes in an OS. The PT contains all the information about each process, while the PCB contains the current state of the process, such as the program counter and CPU registers.
- **Resource management:** The PT and PCB allow the OS to manage system resources, such as memory and CPU time, efficiently. By keeping track of each process's resource usage, the OS can ensure that all processes have access to the resources they need.
- **Process synchronization:** The PT and PCB can be used to synchronize processes in an OS. The PCB contains information about each process's synchronization state, such as its waiting status and the resources it is waiting for.
- **Process scheduling:** The PT and PCB can be used to schedule processes for execution. By keeping track of each process's state and resource usage, the OS can determine which processes should be executed next.

Disadv's of PCB

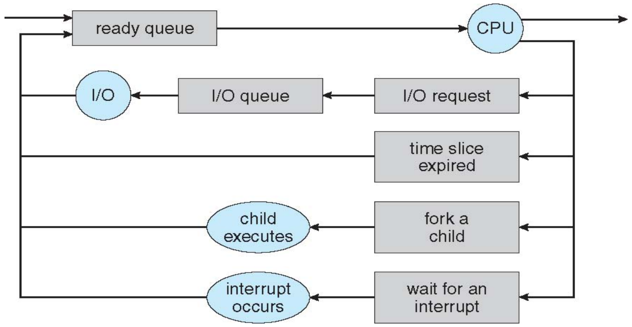
- **Overhead:** PT and PCB can introduce overhead and reduce system performance. The OS must maintain the PT and PCB for each process, which can consume system resources.
- **Complexity:** PT and PCB can increase system complexity and make it more challenging to develop and maintain OSs. The need to manage and synchronize multiple processes can make it more difficult to design and implement system features and ensure system stability.
- **Scalability:** PT and PCB may not scale well for large-scale systems with many processes. As the number of processes increases, PT and PCB can become larger and more difficult to manage efficiently.
- **Security:** PT and PCB can introduce security risks if they are not implemented correctly. Malicious programs can potentially access or modify the PT and PCB to gain unauthorized access to system resources or cause system instability.
- **Miscellaneous accounting and status data –** This field includes information about the amount of CPU used, time constraints, jobs or process number, etc. PCB stores the register content or execution content of the CPU when it was blocked from running. This execution content architecture enables OS to restore a process's execution context when the process returns to the running state. When the process makes a transition from one state to another, OS updates its information in the process's PCB. OS maintains pointers to each process's PCB in a process table so that it can access the PCB quickly.

Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

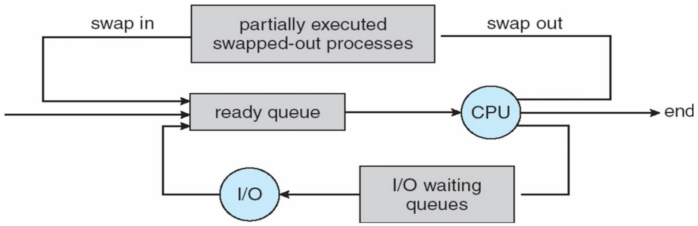


Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

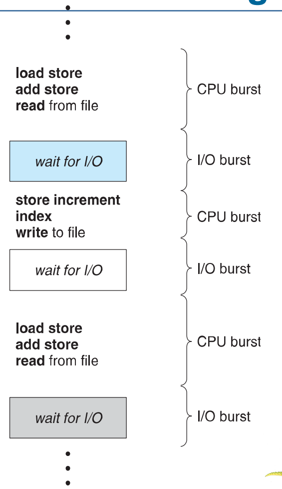


Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

Basic Concepts of CPU Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling are **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per unit time
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

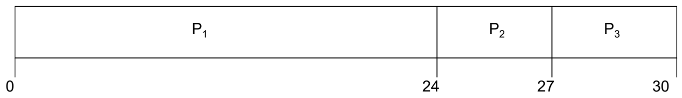
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



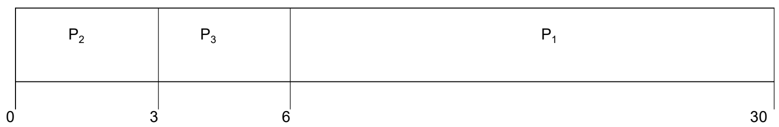
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU burst
  - Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

P_4	P_1	P_3	P_2

0 3 9 16 24

- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

P_2	P_5	P_1	P_3	P_4

0 1 6 16 18 19

- Average waiting time = 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets $1/n$ of the CPU time in chunks of at most *q* time units at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Comparison of Preemptive and Non-preemptive Scheduling

- In preemptive scheduling, CPU is assigned to the processes for a particular time period. In contrast, CPU is assigned to the process until it removes and switches to the waiting state.
- When a process with a high priority appears in the ready queue frequently in preemptive scheduling, the process with a low priority must wait for a long period and can starve. In contrast, when CPU is assigned to the process with the high burst time, the processes with the shorter burst time can starve in non-preemptive scheduling.
- When a higher priority process comes in CPU, the running process in preemptive is halted in the middle of its execution. On the other hand, the running process in non-preemptive doesn't interrupt in the middle of its execution and waits until it is completed.
- Preemptive is flexible in processing. On the other side, non-preemptive is strict.
- Preemptive is quite flexible because critical processes are allowed to access CPU because they come in the ready queue and no matter which process is currently running. Non-preemptive is tough because if an essential process is assigned to the ready queue, CPU process is not be interrupted.
- In preemptive, CPU utilization is more effective than non-preemptive scheduling. On the other side, in non-preemptive, CPU utilization is not effective as preemptive.
- Preemptive is very cost-effective because it ensures the integrity of shared data. In contrast, it is not in the situation of non-preemptive.

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions **hold simultaneously**.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

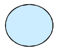
Resource-Allocation Graph

A set of vertices V and a set of edges E.

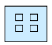
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the **processes** in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all **resource types** in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

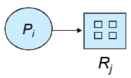
- Process



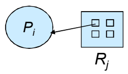
- Resource Type with 4 instances



- P_i requests instance of R_j



- P_i is holding an instance of R_j



System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions **hold simultaneously**.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

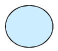
Resource-Allocation Graph

A set of vertices V and a set of edges E.

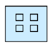
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the **processes** in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all **resource types** in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

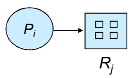
- Process



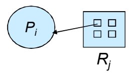
- Resource Type with 4 instances



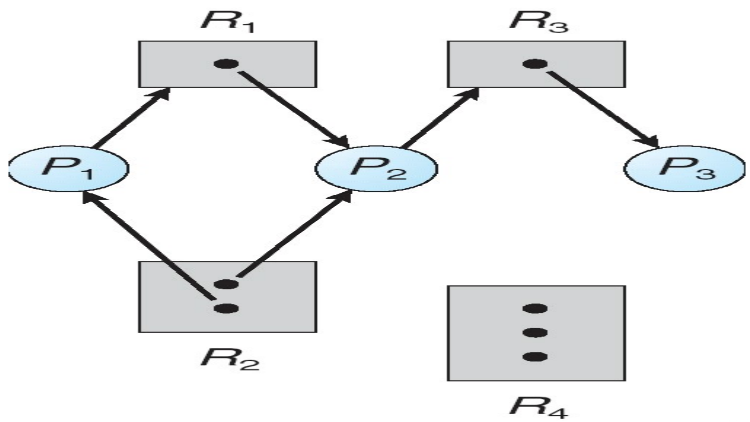
- P_i requests instance of R_j



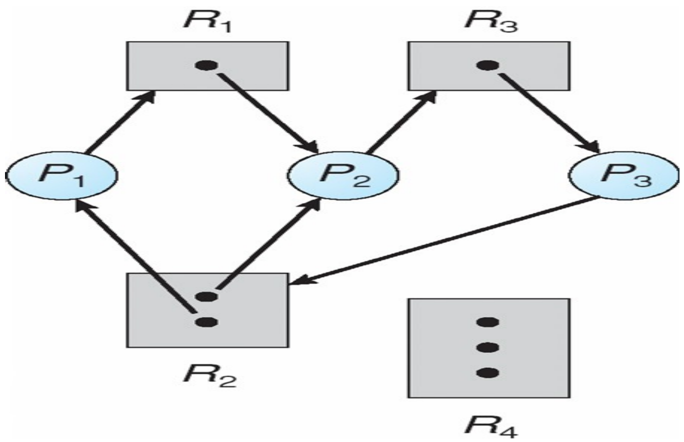
- P_i is holding an instance of R_j



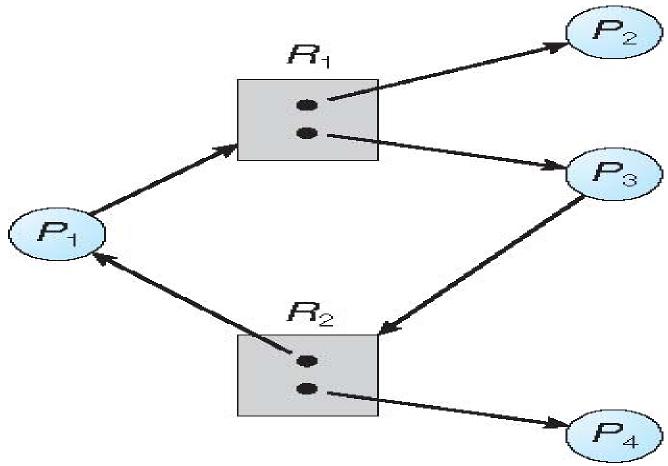
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains **no cycles** ⇒ **no deadlock**
- If graph contains **a cycle** ⇒
  - if only **one instance** per resource type, then **deadlock**
  - if **several instances** per resource type, **possibility of deadlock**

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  1. Deadlock **prevention**
  2. Deadlock **avoidance**
- **OR** allow the system to enter a deadlock state and then **detect it and recover from it.**
- **OR Ignore the problem** and pretend that deadlocks never occur in the system; **used by most operating systems, including UNIX**

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable/read-only resources (e.g., read-only files)
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated **all its resources before it begins execution**, or
  - allow process to request resources **only when the process has none** allocated to it.

Low resource utilization and possible starvation

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that **cannot be immediately allocated to it, then all resources currently being held are released**
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can **regain its old resources, as well as the new ones** that it is requesting
- **Circular Wait** –
  - impose a **total ordering** of all resource types,
  - and require that each process requests resources in an **increasing order** of enumeration

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- Resource-allocation ***state*** is defined
  - the number of **available** and **allocated** resources, and the **maximum** demands of the processes
- The deadlock-avoidance algorithm **dynamically examines the resource-allocation state** to ensure that there can never be a circular-wait condition

Process Synchronization

- Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other, and to prevent the possibility of inconsistent data due to concurrent access. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

- In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.
- On the basis of synchronization, processes are categorized as one of the following two types:
- **Independent Process**: The execution of one process does not affect the execution of other processes.
- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system.

- Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.
- **Race Condition:**
- When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition. Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Critical Section Problem:

- A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.
- In the entry section, the process requests for entry in the **Critical Section.**
- Any solution to the critical section problem must satisfy three requirements:
- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.
- **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
- Solutions are Peterson's method and Semaphores

- **Advantages of Process Synchronization:**
  - Ensures data consistency and integrity
  - Avoids race conditions
  - Prevents inconsistent data due to concurrent access
  - Supports efficient and effective use of shared resources
- **Disadvantages of Process Synchronization:**
  - Adds overhead to the system
  - Can lead to performance degradation
  - Increases the complexity of the system
  - Can cause deadlocks if not implemented properly.