Week 1:

OS is an interface between user and hardware. Its job is to: execute and make solving user programs easier, make systems more convenient to use, use hardware efficiently. Its job is to control and co-ordinate the use of hardware among application programs. Using OS allows the user to code in high-level language as it converts the language to instruct the hardware.

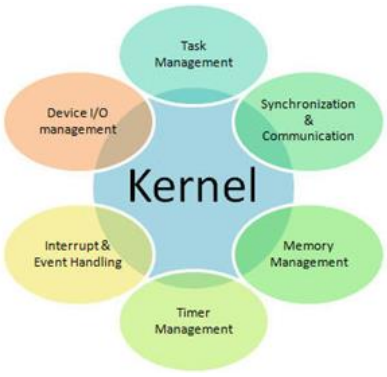
User view of OS: OS should offer ease of use and performance.

System view of OS: OS act as resource manager. OS should be able to manage all hardware resources in system. Hardware used in most efficient manner, process kept waiting for minimum amount of time if requires resources.

Resource allocator: Allocates resources (software & hardware) of computer system and manage them efficiently.

Control Program: Controls execution of user programs and operations of I/O devices.

Kernel: Program that executes forever (everything is an application compared to kernel), The internal part of the OS is often called the kernel.



- It's a program that represents the core of computer.
- Controls everything occurring in system.
- Interacts with hardware on demand from applications.
- Doesn't interact directly with user, interacts with shell, other programs and hardware devices on system (including CPU, memory, disk drives).
- Kernel code is loaded into protected area of memory so it isn't overwritten by less frequently used parts of OS or application programs.

Shell: is the outermost part of an OS and a program that interacts with user commands. It's a program that provides text-only user interface for OS. Term shell is from the fact it is outer layer of OS. Program gives user ability to communicate with OS. Shell is a program layer which understands and executes command user enters. Shell is also called command interpreter.

OS File Manager: Maintains information about files on system. E.g.(located in mass storage, size, type, protection, space available). Files allowed to group E.g. (directories/folders – hierarchical organisation).

OS Device Drivers: Software to communicate with peripheral devices or controllers. Each driver is unique. Translates general requests into specific steps for devices.

OS Memory Manager: Responsible for coordinating use of machine's main memory. Decides what area of memory to allocate to program and its data. Allocates and deallocates memory for programs and knows what areas are free.

OS Schedular: Maintains record for present & new process, removes completed. Memory area(s) assigned, priority, state readiness to execute (ready/wait).

OS Dispatcher: Ensures process ready to run are executed. Time divided into small segments called time slice. When time slice over, dispatcher allows schedular to update process state for each process, select next process to run.

Week 2:

1950 Serial OS: entire program instructions in punched card, translated into card reader, submitted to the OS. Drawbacks: No user & computer system interactions, less memory, lot of time for execution, 1 program executed at a time.

1960 Simple Batch System: High level language magnetic tapes, Jobs batched together based on language. **Operations:** place batched jobs on input. Program monitor manages execution of each program. Monitor utilities loaded when needed. 'Resident monitor' in main memory & available for execution.

Idea of SBS: Reduce setup time by batching similar jobs, Alternate execution between user & monitor program, Rely on available hardware to effectively alternate execution from various parts of memory. Uses automatic Job sequencing (moves to next job when one is done). **Job Control Language (JCL):** provides instructions to monitor: What complier to user, what data to use. **Resident Monitor/** Job Sequencer**:** First basic OS, – initial control in monitor, Loads next program transfers control, job complete - control passed back to monitor. **Desirable Hardware Features:** Memory Protection memory area not altered by user program. Privileged Instructions – executed by resident monitor, traps occur if instructors tried. Interrupts –prevent job monopolising system.

Uniprogramming (batch processing): Jobs are submitted /executed 1 by 1. Entire system used by 1 process at a time. Disadvantage: Wastage of CPU time, No User Interaction, No mechanism to prioritise processes.

1970 Multiprogrammed batch systems: Memory Layout: several jobs in main memory at once, CPU is multiplexed among them. When process goes to I/O, CPU allocated another process. **Uses Interrupts to run multiple programs simultaneously:** When program performs I/O, execute another program until interrupt is received. Requires secure memory, I/O for each program, intervention if program loops indefinitely, CPU scheduling chooses next job. Advantages: CPU utilised all times, Processes finished in less time. Disadvantage: No user interactions.

Types of Multiprogramming OS: **Multitasking OS:** Ability to execute many programs at once. OS swaps every program I/o memory. Program fetched from memory then kept store in secondary memory until required. **Multiuser OS**: OS allows many users to connect with single system running the same OS. **Difference between Multiprogramming and Multitasking OS:** (Multiprogramming) context switching implemented. Allows enhancing CPU utilisation by using jobs to decrease the CPU idle time, using single CPU. (Multitasking) context switching, time sharing implemented.

1970 Time-Sharing: Programs queued in FIFO order. Timer device interrupts after quantum (time slice), interrupted program returned to end of queue , next program is head of FIFO. Interactive – OS finishes execution of command, seeks next statement from user. Online filesystem. Several processes loaded into main memory simultaneously and several users share system. Aim - reduce overall process response time. CPU could execute several processes by providing equal time to each. Benefits: Multiple processes and user requests responded to simultaneously. Better response time. CPU not idle due to regular switching.

1970 Real-Time Systems: Correct system function depends on timeliness. Feedback/control loops. Sensors and actuators. Hard real-time systems - Failure if response time too long, Secondary storage is limited. Soft real-time systems - Less accurate if response time is too long. Useful for application like VR.

1980 Multiprocessor Systems: Multiple CPU in close communication. Improved Throughput, economical, increased reliability. **Kinds:** Vector, pipelined, Symmetric, asymmetric multiprocessing, Distributed memory/shared memory. Allows multiple processors connected with physical memory, computer buses, clocks, peripheral devices. Aim - consume high computing power, increase speed of system. Advantages: **Great Reliability, Improve Throughput, Cost Effective System, Parallel Processing.** Disadvantages: Expensive, Large memory, Speed degrades, Time delay - processor receives message takes appropriate action. Context switching.

Distributed/Loosely coupled systems : Distribute among many processors. Loosely coupled - no shared memory, various communication lines. Client/server architectures. Advantages: Resource sharing. Computation speed-up. Reliability. Communication - e.g. email. Allows distributing system on processors, serves on multiple real time products, multiple users. Processors connected by communication medium, every processor contains own local memory along with other local processor. OS involves multiple computers, nodes, and sites, linked together with LAN/WAN lines. Capable for sharing computational capacity and I/O files, allowing virtual machine abstraction to users. **Done within 3 areas:** Client-Server System: "Tightly Coupled OS". For multiprocessors and same kind multicomputer. Works as centralized server as it provides approval to all requests generated by client systems. Server systems divided into two segments: Computer Server System: Allows interface, client sends own all requests for executing. It sends to back response after executing action, transfer result to client. **File Server System:** Allows file system interface for clients to perform various tasks e.g. creating, updating, deleting files, etc. **Objective** – Hide, manage hardware resources. Peer-to-Peer System: "Loosely Couple System". Contains bunch of processors no shareable memories/clocks. Every processors consist own local memory, make communication with each other through communication medium. **Objective** – provides local services to remote clients. Middleware: allows the interoperability in-between applications running on other OS. Using these services those applications are capable for transferring data to each other. **Objective** – Allows distribution transparency. Advantages: can share all resources between sites - increases data availability. enhances speed of data exchange. reduces probability of data corruption - data replicated on all sites. provides excellent services. decrease load of jobs on one host. scaled easily. more reliable to single system. excellent performance. Better portability. Better re-usability of existing hardware. decrease duration time in data processing. high fault tolerance system. Better flexibility. openness system because system can be accessed from local, remote sites. works independently. Well protective system due to Unique ID. Limitations: centre hub fails, entire network will halt. designed with language not defined till now. more costly. security issues arise while sharing data on networks. data packet corrupted due to following in large networks. maintenance costly. site gets overload, creates big challenges. same time multiple users try to access same data from database then its performance can degrade. Administration is difficult task. only supports few software's.

Week 3:

The Process Concept

An operating system executes a variety of programs: **Batch system** – jobs. **Time-shared systems** – user programs or tasks. **Process** – a program in execution; Process is dynamic/active entity that actions the purpose of the application. When we write and compile the code, we get an executable file. File created is known as a program. File created has all the instructions or the code within it. File is inactive unless executed so program is passive/static entity. A program is system activity that has a set of instructions, performs a specific task. **Batch processing systems** - executing jobs. **Real-time operating system** - program. User can run many programs simultaneously. Process is executing program. The process executes all code line in program. A process can create, delete, schedule another process. After you write a program in any language, two steps follow: (1.Compiling. 2. Running/Executing - makes that program a process.)Process is an instance of the program, has a shorter and minimal lifespan, many resources, considerable overhead. **Program can exist independently but a process cannot exist without a program.**

A program's lifespan is longer - stored in secondary memory until erased, process' lifespan is shorter and limited - terminated after completed.

Process structure in RAM: (Text – program code. **Data** – contains global variables. Both fixed size because neither code/variables is going to change. **Stack** – contains temporary data, function parameters, return addresses, Local variables. Dynamic - difficult to determine the number of function calls required. **Heap** – memory allocated dynamically during run time - cannot determine the memory required.)

Process states: Process change state during lifetime (5 states):((**New** –process is created. **Ready** –process in RAM, waiting for processor/CPU allocation. **Running** – process gets CPU and is executing. **Waiting** –process waiting for some event to occur or an I/O device. **Terminated** – the process finishes its execution normally/forcefully.) **Step 1 –** new process created, admitted into ready state. **Step 2 –** no other process present at running state, is dispatched to running based on scheduler dispatcher. **Step 3 –** higher priority process ready, uncompleted process sent to waiting state from running state. **Step 4 –** I/O or event is completed, process send back to ready state based on the interrupt signal given by running state. **Step 5** – execution of process is completed in running state, will exit to terminate state, completion of process. Waiting, the process occupies main memory, constraint with limited memory, I/O/event might take time to complete.

Process in wait state shifted to secondary memory by defining 2 new states: (Suspend Wait State: If higher priority process to be executed while main memory is full, lower process goes from wait state to suspend wait state to free space in ready state for higher process. Lower process moved to suspend-ready state once resource accessible. Process shifted to ready state once main memory is available. Preferable to move process from wait state to suspend wait state.

Suspend Ready State: If higher priority process to be executed while main memory is full, lower process goes from ready to suspend ready state to free space in ready state for a higher process. Until main memory becomes available, lower process stays in suspend-ready state. Process brought to ready state when main memory becomes accessible.) Operations on process: One can perform two operations on a process in OS: 1.Process creation - Create a new process. Running process can create a new process/ processes. Original process is PARENT process, new process is CHILD process. CHILD can create new child process. Child and parent can perform at the same time. Process in system identified by process identifier, unique process ID called the PID (Process Identifier)

Parent:
- Continues to execute concurrently with its children
- *Waits* until some or all the children terminate

Child :
- Is a duplicate of the parent process
- Has a new program loaded into it

- fork() – unix
- CreateProcess () – Windows

2.Process Termination - There are two methods a process can terminate: Normal termination – process finishes executing final statement. Resources freed by OS. using the **exit() system call**. Forced Termination – parent can terminate child by invoking system call. parent can terminate child due to the following reasons: (Child exceeds its usage of resources. Task assigned to the child is no longer required. Parent exits and OS does not allow child to run if parent terminates. **Using the abort() system call**.) **Cascading termination** - Some OS don't allow CHILD to exist without PARENT, CHILD killed along with its CHILD. **Zombie Process -** process terminated but status not read by parent. In Linux, if parent terminates then all associated child forced to exit. In Windows, child continue running if parent terminated. **Process control block(PCB):** Process identification number (PID) assigned to process. OS needs to keep track of all processes. PCB is used to track the process's execution status. Each block of memory contains information about the process state. Information required, must be saved when process switches states - OS must update information in the process's PCB.Is data structure that stores information of process. Information associated with each process, called task control block: (Process state. Program counter – location of next instruction. CPU registers – contents of all process 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.) PCBs stored in kernel space, unique for every process. OS uses Process Table to find the PCB present in memory. Process table contains Process ID, reference to the corresponding PCB in memory. When context switch occurs between processes, OS refers to Process table to find reference to PCB with help of corresponding Process ID

Week4:

$$\text{CPU utilization} = \frac{\text{total CPU time}}{\text{total real time}}$$

Uniprogramming and utilisation - CPU instructions were very fast electrical signals. I/O operations slow. If program required many I/O operations, creates wasted time as CPU is idle measured as CPU utilisation. utilisation - actual usage of a resource divided by the potential usage. **Multiprogramming and Concurrency:: multiprogramming** (and **multitasking**), several processes loaded into memory and available to run. Process initiates an I/O operation, the kernel selects different process to run on the CPU. Allows kernel to keep CPU active, reducing the amount of wasted time. **Implementing Multiprogramming::** There are two forms of multiprogramming that have been implemented: **1) Pre-emptive multitasking:** A multiprogramming strategy in which processes are granted a time-limited access (time is quantum) to the CPU and interrupted when that time limit expires**. 2)Cooperative multitasking:** process runs until it voluntarily relinquishes control of CPU or initiates an I/O request. Advantages: its simplicity of design and implementation. If processes are very interactive, it can have low overhead costs. Disadvantage: vulnerable to rogue processes that dominate the CPU time. **Context Switching::** storing context/state of process to be reloaded when required and execution resumed from the same point as earlier. Critical for multiprogramming, they introduce complexity and overhead costs in terms of wasted time. **(1)** kernel needs to perform a context switch, must decide which process to make active; choice performed by **scheduling** routine – needs time to run. The steps involved in context switching are as follows – Save the context of process running on CPU; Update PCB and other important fields. Move PCB process into relevant queue e.g. ready queue. Select new process for execution. Update PCB of the new process; Includes updating the process state to running. Update memory management data structures as required. Restore context of the previous when loaded again on processor by loading previous values of PCB and registers. **(2)** introduce delays related to the memory hierarchy. **3 major triggers for context switching. These are given as follows –Multitasking:** process switched out of CPU so another process can run. state of old process saved, and new process state is loaded. pre-emptive system - processes switched out by scheduler. **Interrupt Handling:** hardware automatically switches part of context when interrupt occurs.. some context changed to minimize the time to handle interrupt. **User and Kernel Mode Switching:** context switch takes place when transition between user mode and kernel mode is required in OS. gives impression to user that system has multiple CPUs by executing multiple processes. **Process Scheduling** activity of process manager - handles removal of running process from CPU and selection of another process on basis of a particular strategy. **Goal:** keep CPU busy all time, maximise CPU use. **Time-sharing system goal** - switch between user processes so each user can interact with process. To meet these goals, a Process scheduler selects among available processes for next execution on CPU. **There are two categories of scheduling: (1) Non-pre-emptive:** resource can't be taken from process until process completes execution. The switching of resources occurs when running process terminates and moves to a waiting state. (**voluntary**). **(2) Pre-emptive:** OS allocates resources to process for fixed time. the process switches running state to ready state/waiting state to ready state. switching occurs as CPU give priority to other processes and replace higher process with the running process. (**forced**). Process Scheduling Queues helps to maintain a distinct queue for each process states and PCBs. Process of same execution state placed in same queue. state of a process is modified, its PCB needs to be unlinked from its existing queue, which moves back to new state queue. **Three types of operating system queues are: 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. **Queueing diagram** represents queues, resources, flows (**Rectangle represents a queue. Circle denotes the resource. Arrow indicates the flow of the process**). Process scheduling: (1) Every new process put in Ready queue . Waits in ready queue until finally processed for execution. (2) One of the processes is allocated the CPU and it is executing. (3) The process issues an I/O request. (4) Then placed in the I/O queue. (5) process should create a new subprocess. (6) process waiting for termination. (7) removed forcefully from CPU, as a result interrupt. interrupt completed; process sent back to ready queue. Scheduling Criteria: Max CPU utilization. Max throughput. Min turnaround time. Min waiting time. Min response time.

There three types of Process Schedulers: 1. Long-term scheduler(job scheduler) – selects which processes should be brought into the ready queuechooses processes from the pool (secondary memory) keeps them in ready queue in primary memory. Invoked infrequently (may be slow), controls **degree of multiprogramming.** Provides balanced combo of jobs, Processes can be described: **I/O-bound process** –more I/O, many short CPU bursts. **CPU-bound process** – more computations; few very long CPU bursts. Strives for good ***process mix.*** **2. Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU. **Main goal-** boost system performance to set criteria. Helps select from group of processes that are ready to execute and allocates CPU to one. Invoked frequently (must be fast). ***The dispatcher*** responsible for loading process selected on the CPU **Context switching is done by the dispatcher only.** Dispatch latency – time it takes for the dispatcher to stop one process and start another running. **Dispatcher does the following:** Switching context. Switching to user mode. Jumping to the proper location in the newly loaded program. **3. Medium-term scheduler** can be added if multiple programming needs to decrease. (1) Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping.** (2) responsible for suspending and resuming the process. **CPU utilization** – keep the CPU as busy as possible. **Throughput** – # of processes that complete their execution per time unit. **Turnaround time** – amount of time to execute a particular process. **Waiting time** – amount of time a process waiting in ready queue. **Response time** – amount of time it takes from request submitted to first response produced, not output (for time-sharing environment).

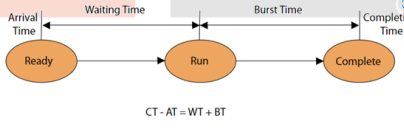
Week 5:

multi-programming goal is keeping CPU busy. Uni-processor - CPU is idle, its allocated to new process.

Short-term scheduler decides process to allocate CPU. CPU scheduling/ Process scheduling/ thread scheduling – select next process when CPU is idle based on scheduling algorithms. First-Come, First-Served (FCFS), Shortest-Job-Next (SJN), Priority, Shortest Remaining Time, Round Robin(RR), Multiple-Level Queues. CPU scheduling required when: (1)Process switches from running state to waiting state. (process needs I/O, releases CPU voluntarily – Non-pre-emptive scheduling). (2) Process switches from running state to ready state (time expired/ higher process so releases CPU forcefully – Pre-emptive scheduling). (3) Process switches from waiting state to ready state (time expired/ higher process so releases CPU forcefully – Pre-emptive scheduling). (4) A Process terminates. (releases CPU voluntarily – Non-pre-emptive scheduling). A Dispatcher is module of OS which actually allocates/ gives control of CPU to process. **involves three steps: (1)** performs Context switching. **(2)** Switching to user mode. **(3)** Jumping to proper location in user program to restart process. **Dispatcher** needs to be as fast as possible, as it is run on every context switch. The time consumed - **dispatch latency**. **1. Arrival Time: T**ime process enters into ready queue. **2. Burst Time:** total time required by CPU to execute whole process. Doesn't include waiting time. **3. Completion Time: T**ime process enters the completion state or time process completes execution. **4. Turnaround time:** total time spent by process from arrival to completion. **5. Waiting Time:** Total time process waits for CPU to be assigned. **6. Response Time:** difference between arrival time and time process first gets CPU.

Non-Preemptive scheduling
- Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
- Voluntarily releases the CPU

Preemptive Scheduling
- The CPU is allocated to the new process if it has higher priority as compared to current process or if the time slice expires.
- Forced to release the CPU



CT - AT = WT + BT

TAT = CT - AT

Waiting Time = TAT - BT

TAT → Turn around time
BT → Burst time
AT → Arrival time

process	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	3
P4	3	1
P5	4	8

P4	P1	P3	P2	P5
1	4	10	18	20	24

P1: 4 – 2 = 2
P2: 18 – 5 = 13
P3: 10 – 3 = 7
P4: 0 – 0 = 0
P5: 20 – 8 = 16

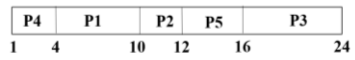
Waiting time = Start time – Arrival time
Av. Waiting time = 2+13+7+0+16/5 = 7.6

Week 5.1

First Come First Serve Algorithm: process arrives first is allocated CPU first. Queue maintained in RAM (ready queue) and processes allocated to CPU as per their position in queue i.e., first process in queue is allocated CPU first. **Key Points of FCFS** : Job that arrives first is scheduled first. Non-pre-emptive algorithm. Implemented using queue. Suffers from Convoy effect (phenomenon associated with the FCFS algorithm, in which the whole OS slows down due to few slow processes).

Week 6.1: Shortest Job first (SJF):
Job with smallest Burst Time is executed first. Helpful for types of processing of jobs in which completion time of a job can be calculated easily like Batch-type processing. Can improve CPU processing as it executes the shorter jobs first which leads to a short turnaround time. Suffers from Starvation if shorter processes keep coming. **SJF can be:** Non-pre-emptive (Simply SJF), Pre-emptive (SRTF). (shortest burst time chosen first, once that is completed the next shortest burst time is completed next.

Process	Burst time	Arrival time
P1	6	2
P2	2	5
P3	8	3
P4	3	1
P5	4	8

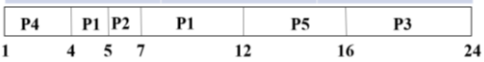


P1: 4 – 2 = 2
P2: 10 – 5 = 5
P3: 16 – 3 = 13
P4: 1 – 1 = 0
P5: 12 – 8 =4
Av waiting time = 2+5+13+0+4/5 = 4.8

Image for SJF

Week 6.2: Shortest Job First (Pre-emptive)/ Shortest Remaining Time First (SRTF)
While a process is running if a process with lower burst time arrives then it will pre-empt the currently running process. Whenever a new process arrives check whether its burst time is less than the remaining burst of currently running process Also known as Shortest Remaining Time First (SRTF)

Process	Burst time	Arrival time
P1	6/0	2
P2	2/0	5
P3	8	3
P4	3/0	1
P5	4	8



Waiting time = Start time – Arrival time + wait time for next burst

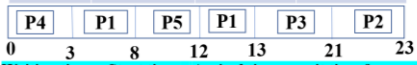
P1 = 4-2 +2 = 4
P2 = 5 – 5 = 0
P3 = 16 – 3 = 13
P4 = 1 – 1 = 0
P5 = 12 – 8 = 4

Av. Waiting Time = 4+0+13+0+4/5 = 4.2

Image for SRTF

Week 6.3: Priority Scheduling (Non-Pre-emptive)
Number is assigned to each process which indicates its priority level. Lower the number, higher is the priority. Follows method which priority is set to processes available for execution, and process is selected based on descending order into ready queue for execution by CPU. Several factors used to determine priority value of process. E.g. time taken to complete execution, memory spaces required by process, etc. Suffers from Starvation if shorter processes keep coming. This problem can be solved using the concept of ageing. In Non-pre-emptive algorithm, process with highest priority is allowed to execute in CPU, and if there is an arrival of another process with a priority higher than process under execution, then new process will have to wait until execution of current process is completed. **Non-Pre-emptive Priority Scheduling:** Each instruction has control of CPU until finished, then CPU is passed to the next high priority.

Process	Burst time	Arrival time	Priority
P1	6/1	0	2
P2	2	5	3
P3	8	3	2
P4	3/0	0	1
P5	4	8	1



Waiting time = Start time – Arrival time + wait time for next burst

P1: 3 – 0 + 4 = 7
P2: 21 – 5 = 16
P3: 13 – 3 = 10
P4: 0 – 0 = 0
P5: 8 – 8 = 0

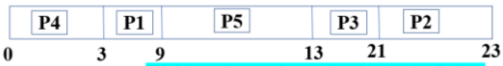
Av. Waiting time = (7+16+10+0+0)/5 = 33/5 = 6.6

Image for Pre-emptive priority

23

Week 6.4: Priority Scheduling (Pre-emptive)
If a new process arrives which is having higher priority than the currently running process, then the currently running process is pre-empted. (if higher priority arrives it takes over and is executed immediately and the current process has to wait.

Process	Burst time	Arrival time	Priority
P1	6	0	2
P2	2	5	3
P3	8	3	2
P4	3	0	1
P5	4	8	1



Waiting time = Start time – Arrival time

P1: 3 – 0 = 3
P2: 21 – 5 = 16
P3: 13 – 3 = 10
P4: 0 – 0 = 0
P5: 9 – 8 = 1

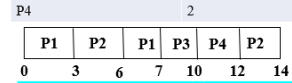
Av. Waiting Time = (3+16+10+0+1)/5=6

Image for Non-pre-emptive priority

Week 7.1: Round-Robin Scheduling Algorithm
pre-emptive. CPU shifted to next process after fixed interval of time called time quantum. Process that is pre-empted is added to the end of queue. Acts as FCFS with Preemption. Not good (Time quantum, Very less – Context switching, Huge – FCFS). 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.

Time quantum = 3

Process	Burst time	Arrival time
P1	4/1/0	0
P2	5/2/0	2
P3	3/0	4
P4	2	5



Waiting time = Start time – Arrival time + wait time for next burst

P1 = 0 – 0 + 3 = 3
P2 = 3 - 2 + 6 = 7
P3 = 7 – 4 = 3
P4 = 10 – 5 = 5

Av. Waiting Time =(3+7+3+5)/4 = 4.5

Image for Round Robin

14

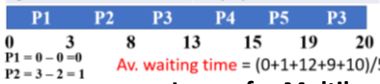
Week 7.2: Multilevel Queue Scheduling Algorithm
Ready queue is partitioned into separate queues, e.g.: foreground (interactive), background (batch). Process permanently in a given queue. Each queue has own scheduling algorithm: foreground – RR, background – FCFS. **Scheduling must be done between the queues:** Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation. Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Feedback Queue: process can move between various queues; aging can be implemented this way. Multilevel-feedback-queue scheduler defined by following parameters: number of queues, scheduling algorithms for each queue, method to determine when to upgrade process, method to determine when to demote process, method to determine which queue process will enter when process needs service.

Process	Queue	Burst Time	Arrival Time
P1	Q1	3	0
P2	Q2	5	2
P3	Q2	6/1/0	2
P4	Q1	2	4
P5	Q1	4	5

Queue	Scheduling
Q1	FCFS
Q2	SJF(NP)

Queues are scheduled using RR (TQ=5)



P1 = 0 – 0 =0
P2 = 3 – 2 = 1
P3 = 8 – 2 +6 = 12
P4 = 13 – 4 = 9
P5 = 15 – 5 =10

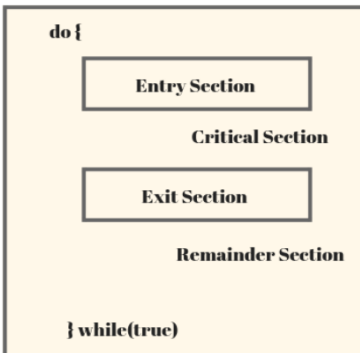
Av. waiting time = (0+1+12+9+10)/5 = 6.4

Image for Multilevel Queue

Multiple-Processor: CPU scheduling more complex when multiple CPUs available
Homogeneous processors within multiprocessor. A**symmetric multiprocessing** – only one processor accesses system data structures, alleviating need for data sharing. **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes (most common). **Processor affinity** – process has affinity for processor its currently running: soft **affinity, hard affinity**, Variations including **processor sets.**
SMP need to keep CPUs loaded for efficiency. **Load balancing** attempts to keep workload evenly distributed. **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs. **Pull migration** – idle processors pulls waiting task from busy processor.

Week 7.3: Threads, Process Communication
Thread is sequential flow of tasks within process. Can be of same or different types. Used to increase application's performance. Has its own program counter, stack, and set of registers. Threads of single process might share same code and data/file. Threads are also termed as lightweight processes as they share common resources. Modern applications are multithreaded. Threads run within application. Tasks with application can be implemented by separate threads: Update display, Fetch data, Spell checking, Answer a network request. Process creation is heavy-weight while thread creation is light-weight. Can simplify code, increase efficiency. Kernels are generally multithreaded.
Benefits: **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces. **Resource Sharing** – threads share resources of process, easier than shared memory or message passing. **Economy** – cheaper than process creation, thread switching lower overhead than context switching. **Scalability** – process can take advantage of multiprocessor architectures. User Level and Kernel level Threads: Strong relationship between user level threads and kernel level threads. Task accomplished on execution of program, results in process. Task incorporates sub task/s. Sub tasks carried out as functions within program by threads. OS (kernel) unaware of threads in user space.
Two types of threads, User level threads (ULT) and Kernel level threads (KLT). **User Level Threads :** Threads in user space designed by application developer using thread library to perform unique subtask. **Kernel Level Threads :** Threads in kernel space designed by OS developer to perform unique functions of OS. Similar to interrupt handler. Process Communication in OS: Interprocess Communication(IPC) provides mechanism to exchange data and information across multiple processes, on single or multiple computers connected by a network. Way for multiple processes or threads communicate among each other. IPC in OS obtains modularity, computational speedup and data sharing. Ways of IPC are pipe, message passing, message queue, shared memory, direct communication, indirect communication and FIFO. Multicore Programming: Multicore or multiprocessor systems putting pressure on programmers, challenges include: Dividing activities, Balance, Data splitting, Data dependency, Testing and debugging. **Concurrency** - (supports more than one task making progress) two or more tasks can start, run, and complete in overlapping time periods. Doesn't mean they'll ever both be running at same instant. E.g., two lines of customers ordering from a single cashier (lines take turns ordering). **Parallelism** - (implies system can perform more than one task simultaneously) tasks literally run at the same time. E.g., two lines of customers ordering from two cashiers (each line gets its own cashier). Multithreading: Multithreaded process, unique subtask allocated to every thread within process. Threads may use same/different data section. Threads within same process will share code section, data section, address space, open files etc. Subtasks are concurrently performed by sharing code section, may result in data inconsistency. Requires suitable synchronization techniques to maintain control flow to access shared data (critical section). Processes in OS: Processes that executing concurrently in OS may be either independent/cooperating processes. **Independent:** Process not affect/affected by other processes executing in system; Process don't share any data is independent. **Cooperating** Process can affect/affected by other processes executing in system; Process shares data is cooperating.

Week 8: Process Synchronization and Deadlocks
Process synchronisation - several threads/processes share data, running in parallel on different cores , then changes made by one process may override changes made by another process running parallel. Resulting in inconsistent data. Requires processes to be synchronized, handling system resources and processes to avoid situation is **Process Synchronization.** Process Synchronization prevents **race around condition,** in which several processes access and manipulate same data. Outcome of execution depends upon order of access. **Processes coordinate with each other.**
The main blocks of process are –
1.Entry Section – To enter critical section code, process must request permission. Entry Section code implements request. **2. Critical Section –** Segment of code where process changes common variables, updates table, writes to file etc. When 1 process is executing in its critical section, no other process is allowed to execute in its critical section.
3.Exit Section – After critical section is executed , this is followed by exit section code which marks end of critical section code. **4. Remainder Section –** remaining code of process is known as remaining section.



Critical Section - **A section of code, common to n cooperating processes, in which the processes may be accessing common variables.**
Critical Section Environment contains: **Entry Section:** Code requesting entry into the critical section. **Critical Section:** Code one process can execute at any one time. **Exit Section:** end of critical section, releasing or allowing others in. **Remainder Section:** Rest of code AFTER critical section.

The critical section must ENFORCE ALL THREE of the following rules: **Mutual Exclusion:** Only ONE process can execute in critical section at a time. **Progress:** If no one in critical section and someone wants in, then processes not in remainder section must decide in finite time who should go in. **Bounded Wait:** All requesters must eventually be let into critical section.

FLAG TO REQUEST ENTRY:
Each processes sets flag to request entry. Then each process toggles a bit to allow the other in first.
Code is executed for each process i.

Shared variables
Boolean flag[2];
initially flag [0] = flag [1] = false.
flag [i] = true ⍰ Pi ready to enter its critical section

Peterson's Solution

```
do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn == j) ;
    critical section
    flag [i] = false;
    remainder section
} while (1);
```

The hardware required to support critical sections must have (minimally): Indivisible instructions Atomic load, store, test instruction. For instance, if store and test occur simultaneously, test gets EITHER old or new, but not some combination. Two atomic instructions, if executed simultaneously, behave as if executed sequentially. **Semaphore:** Peterson's algorithm, are complex, error prone, and inefficient. **Semaphores** refer to integer variables primarily used to solve critical section problem via combining two of atomic procedures, wait and signal, for process synchronization. Variable used to solve critical section problem and to achieve process synchronization in multiprocessing environment. **Wait:** The wait operation decrements value of argument S, if positive. If S is negative or zero, then no operation is performed. (**wait (mutex);** ←Mutual exclusion: mutex init to 1.**CRITICAL SECTION signal(mutex); REMAINDER). Signal:** Signal operation increments value of argument S. **Two types of semaphores:** Counting Semaphore used for mutual exclusion and conditional synchronization. Binary Semaphore designed for mutual exclusion. **Solution to Dining Philosopher:** Mutex is used so no two philosophers may access pickup or putdown at same time. Array is used to control behaviour of philosopher.
Deadlock: Danger of concurrent programming is deadlock. Race condition produces incorrect results, deadlock results in deadlocked processes never making any progress. Deadlocks occur via system calls, etc. Process waiting for a resource held by a second process that's waiting for a resource that first holds. Deadlock arises if four conditions **(Coffman Conditions)** hold simultaneously. **Mutual exclusion:** only one process can use resource. **Hold and wait:** process holding at least one resource waiting to acquire additional resources held by other processes. **No preemption:** resource released only voluntarily by process holding it, after process has completed task. **Circular wait:** exists set $\{P_0, P_1, P_n\}$ of waiting processes: P_0 waiting for resource held by P_1, P_1 waiting for resource held by P_n, and P_n waiting for resource held by P_0. graph contains no cycles ⇒ **no deadlock. G**raph contains a cycle ⇒ only one instance per resource type, deadlock. Several instances per resource type, possibility deadlock.

Week 9: Methods of deadlock handling
Ignore it: Ostrich algorithm Pretend there is no problem. Reasonable if deadlocks occur very rarely, cost of prevention is high. UNIX and Windows takes this approach. Is trade off between convenience & correctness. Prevent it: Stop Coffman Conditions **Mutual exclusion:** spool everything. **Hold & wait:** Request all resources initially. **No pre-emption:** Take resources away (no possible if resource is in use). **Circular wait:** Order resources numerically. Avoid it: Deadlock avoidance Requires system has reasoning information available. Requires each process declare maximum number of resources it may need. Deadlock-avoidance algorithm examines resource-allocation state to ensure can never be circular-wait condition. Resource-allocation state defined by number of available allocated resources, maximum demands of processes. **Safe State** process requests available resource, system decides if immediate allocation leaves system in safe state. System in safe state if exists sequence of ALL processes in systems. For each Pi, resources that Pi can still request can be satisfied by currently available resources. If system in safe state -> no deadlocks. If system in unsafe state -> possibility of deadlock. Avoidance -> ensure that system will never enter an unsafe state. Detect it: Deadlock Detection Allow system to enter deadlock state, Detection algorithm, Recovery scheme. **Single Instance of Each Resource Type**. Maintain wait-for graph. Nodes are processes. Pi -> Pj if Pi is waiting for Pj. Call algorithm that searches for cycle in graph. Cycle = exists deadlock. Algorithm to detect cycle in graph requires order of n2 operations, n is number of vertices in graph. **Recovery from Deadlock: Process Termination.** Abort all deadlocked processes. Abort one process at a time until deadlock cycle eliminated. **Orders to abort in:** Priority of process; How long process has computed, and how much longer to completion; Resources process used; Resources process needs to complete; How many processes will need to be terminated; Is process interactive/batch. **Recovery from Deadlock: Resource Pre-emption: Selecting a victim –** minimize cost. **Rollback –** return to some safe state, restart process for that state. **Starvation –** same process may always be picked as victim, include number of rollback in cost factor.

Week 10: Memory Management
Memory defined as collection of data in specific format. Used to store instructions and process data. Memory comprises large array/group of words/bytes, each with own location. Primary motive of computer system is executing programs. should be in the main memory during execution. CPU fetches instructions from memory according to value of the program counter. Main memory is central to operation of computer. Repository of rapidly available information shared by CPU and I/O. Where programs and information kept when processor is utilising them. Associated with processor = moving instructions/info is extremely fast. Known as RAM(Random Access Memory). Memory Management In multiprogramming, OS resides in memory, rest used by multiple processes. Task of subdividing memory among different processes is called memory management. method in OS to manage operations between main memory and disk during process execution. Aim is achieving efficient utilisation of memory. Memory Management required (de-)Allocate memory before and after process execution. Track of used memory space by processes. Minimise fragmentation issues. Proper utilization of main memory. Maintain data integrity while executing of process.
Logical and Physical Address **Logical Address space:** An address generated by CPU. Known as a Virtual address. Can be changed. **Physical Address space:** Address seen by memory unit. Known as a Real address. Computed by Memory Management Unit (MMU). Run-time mapping from virtual >physical addresses done by hardware device MMU. Remains constant. Memory-management unit (MMU) Hardware device at run time maps virtual to physical address. The user program deals with *logical* addresses; it never sees the *real* physical addresses. Execution-time binding occurs when reference is made to location in memory. Logical address bound to physical addresses. Uniprogramming Shares memory between program and OS. Memory allocation is trivial. No relocation needed, user programs always loaded into same memory location. linker produces same loading address for program. Multiprogramming method to accommodate several programs in memory at same time is partitioning. Achieve multiprogramming simply divide memory up into n partitions during system generation/startup
When job arrives, put into input queue for smallest partition large enough to hold it. Contiguous Allocation contiguous memory allocation technique, memory allocated continuously, and no spanning allowed. *process given continuous section of memory in main memory.* If process needs 2MB space, then process must be allocated to a partition which is 2MB or more. Two types in contiguous memory allocation fixed partition, dynamic partition. Memory Partitioning **Dynamic Partitioning -** main memory is not divided initially. Process arrives in main memory, given that much memory. Size is not fixed initially, size = size of process. Dynamic storage-allocation problem **First-fit**: Allocate ***first*** hole that is big enough. Scan list for first entry that fits. If greater in size, break it into allocated and free part. May have lots of unusable holes at beginning. External fragmentation. **Next-fit -** Begins its search from point in list where last request succeeded. Allocates block of memory at end of memory. Largest block of memory is broken up into smaller blocks. Compaction required to obtain large block at end of memory. Slightly slower. **Best-fit**: Allocate ***smallest*** hole, big enough; Chooses block closest in size to request. Poor performer. Searches complete list, unless ordered by size. Smallest amount of fragmentation left; memory compaction done more often. **Worst-fit:** Allocate ***largest*** hole; Chooses block largest in size (worst-fit). Leaves usable fragment left over. Poor performer. Searches entire list. Produces the largest leftover hole. Fragmentation **External Fragmentation** – total memory space exists to satisfy a request, but contiguous. **Internal Fragmentation** – allocated memory slightly larger than requested memory; size difference is memory internal to partition, but not being used. Reduce external fragmentation by **compaction.** Place all free memory together in one large block. Possible *only* if relocation is dynamic, done at execution time. I/O problem Latch job in memory while it is involved in I/O. Do I/O only into OS buffers. Non-contiguous memory Non-Contiguous memory management scheme, program divided into different blocks and loaded at different portions of memory. Can be classified depending upon size of blocks and whether blocks reside in the main memory. Reduces the wastage of memory. Two key schemes: Paging, Segmentation. Paging Physical address space of a process is non-contiguous. Implementation: (**Frames –** fixed size blocks of physical memory. **Pages** – fixed size slots of physical memory). When a process is to be executed, its pages are loaded into available memory grams from the backing store. **Page Table** – to translate logical address to physical address. Implementation of page table Page table kept in main memory. Page-table base register (PTBR) points to page table. Page-table length register (PTLR) indicates size of page table. Every data access requires two memory accesses. One for page table and one for data. Fast-lookup hardware cache called associative registers or translation look-aside buffers (TLBs). Advantages of Paging **Fast to allocate and free; Alloc()**: Keep free list of free pages and grab first page in list; **Free()**: Add page to free list. **Easy to swap-out memory to disk -** Page size matches disk block size. Can swap-out only necessary pages. Easy to swap-in pages back from disk. **Eliminates External Fragmentation. More efficient swapping:** Swapping is easy between equal-sized pages and page frames. Disadvantages of Paging - **Additional memory reference**: Inefficient. Page table too large to store. Page tables kept in main memory. MMU stores only base address of page table. Storage for page tables :Simple page table: Require entry for pages in address space. Partial solution: Base and limits for page table. **Internal fragmentation:** Page size does not match allocation size. Wasted memory grows with larger pages. Segmentation is a non-contiguous memory management technique which the program is divided into variable size parts called segments. Segmentation supports user's view of memory. A segment is a logical unit such as main program, procedures, function, stacks, heaps, common blocks, symbol table, arrays etc. The logical address space is a collection of segments. Each segment has Name, Length. Segmentation Architecture Logical address consists of a two tuple:,<segment-number, offset>. **Segment table** – maps two-dimensional physical addresses; each table entry has: **base** – contains starting physical address of segment. **limit** – specifies length of segment. **Segment-table base register (STBR)** points to the segment table's location in memory. **Segment-table length register (STLR)** indicates number of segments used by a program; segment number ***s*** is legal if ***s*** < **STLR. Advantages of Segmentation -** Offer protection within the segments. You can achieve sharing by segments referencing multiple processes. Not offers internal fragmentation. Segment tables use lesser memory than paging. **Disadvantages of Segmentation -** processes are loaded/ removed from main memory. free memory space separated into small pieces, may create problem of external fragmentation. Costly memory management algorithm. Demand Paging ***Virtual memory is implemented using DEMAND PAGING***
Like paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, use ***lazy swapper*** **or** ***pager***. Page Fault first reference to that page will trap to operating system: **page fault:** Operating system looks at another table to decide: Invalid reference ⇒ abort, Just not in memory. 1. Find free frame. 2.Swap page into frame via scheduled disk operation. Thrashing
If process does not have "enough" pages, the page-fault rate is very high. Page fault to get page. Replace existing frame. Need replaced frame back. This leads to: Low CPU utilisation. Operating system thinking that it needs to increase the degree of multiprogramming. Another process added to the system. **Thrashing** ≡ a process is busy swapping pages in and out.