Operating Systems Ka Khulasa

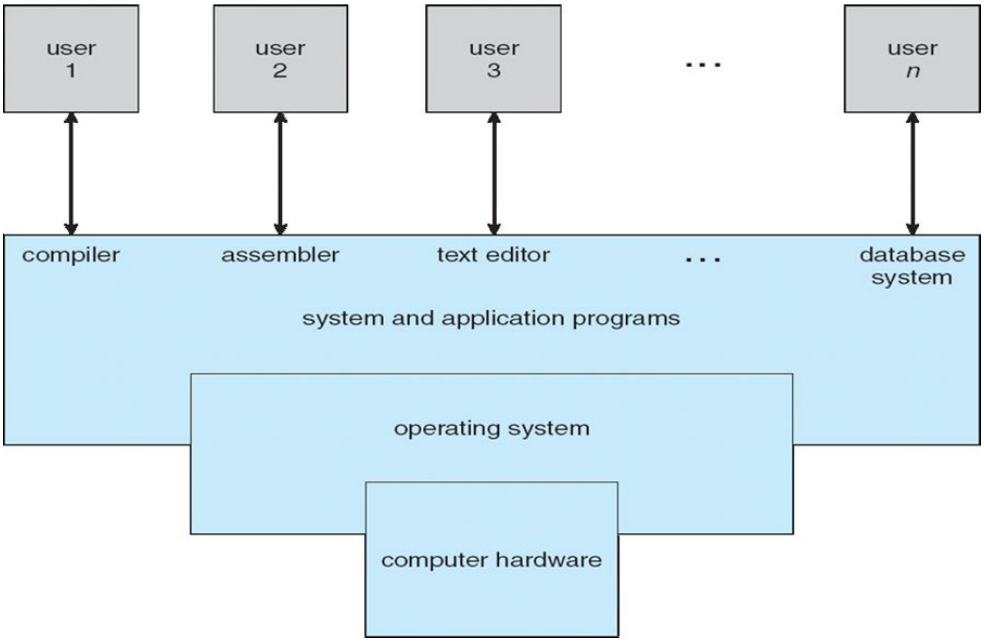
Syllabus

Chapter 1: Introduction
Chapter 2: System Structures
Chapter 3: Process Concept
Chapter 6: CPU Scheduling (Topics from Multilevel Queues onwards)
Chapter 4: Threads (All topics)
Chapter 8: Memory Management (All topics)
Chapter 9: Virtual Memory
Chapter 5: Process Synchronization
Chapter 7: Deadlocks
Chapter 10-13: Mass Storage Structure
Chapter 14-15: Protection & Security

This page is intentionally left blank

Introduction

- Definition
  - Program that manages computer hardware
  - Intermediary between user and hardware
  - Provides an environment for application programs
- Design
  - Optimize utilization of hardware
  - Modular with a layered architecture
- Purpose
  - Control and coordinate resources among applications, optimally
- Computer Components
  - Hardware
    - CPU, Memory, I/O etc
  - OS
    - You already know
  - Applications
    - Word Processor, Compiler, Web Browser etc
  - User
    - You are special
- Highly Complicated Visual



From the user's point of view, the OS is designed for the following situations:

- PC
  - Ease of use
  - Performance

    - Single user interface
- MainFrame or MiniComputer
    - Maximize resource utilization for each user
- Network or Server
    - Compromise between individual usability and maximum resource utilization
- Mobile/Smartphones
    - Individual usability
    - Battery life is also taken importance
- Embedded
    - No/Little user interface
    - Run without user intervention

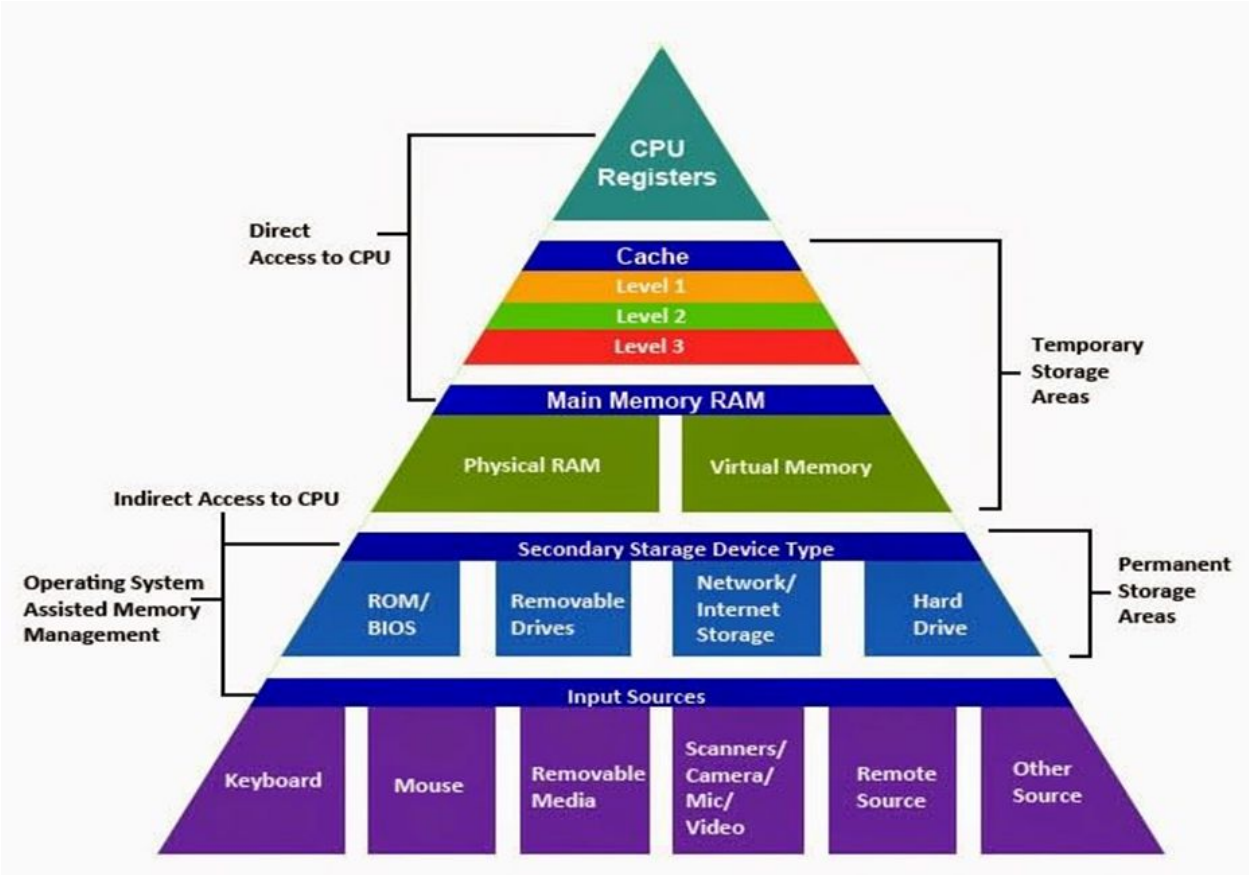
From the system's point of view, the OS does the following:

- Resource allocation
    - Memory Space, File Storage Space, I/O devices, CPU time etc
    - Handle conflicting requests
- Control programs
    - Execute and prevent errors
    - Control I/O devices

One program running at all times is called the Kernel. The other programs are system programs which are associated with the OS and application programs not associated with the OS.
For mobiles a set of frameworks called middleware, is also included to provide developers with additional services.
To understand operating part, we need to consider the computer system first. Generally a computer system consists of the following:

- One or more CPUs
- Device and Memory Controller
- Bootstrap Program
    - Typically stored in ROM
    - Load OS Kernel into memory
    - System services (daemons) also start and run with the kernel (First system process is init on UNIX)
- After loading, OS waits for an event (signalled by hardware/software interrupt)
    - Hardware interrupts can happen at any time but software ones trigger an interrupt by executing a system call
    - Interrupt causes a transfer of control to ISR
    - Procedure of transfer varies according to the system but something common is saving the address of interrupted instruction and state of processor then returns after completion of the interrupt handler
- The other thing that drives OS are traps
    - Traps are also called exceptions
    - Special request from user program
    - Most likely caused by an error

The Storage Structure is quite simple:



Higher to lower levels:

- Decrease in speed
- Increase in size
- Decrease in cost
- More persistence/permanence

The following jobs are handled by the OS for the memory:

- Memory management (where to place the processes)
- CPU scheduling (choose which process to run from memory)
- Swapping processes in and out of main memory
- Break processes into virtual memory which are too large for the physical memory
- File system and disk management
- Protection (from other processes/users), job synchronization + communication and prevent deadlocks

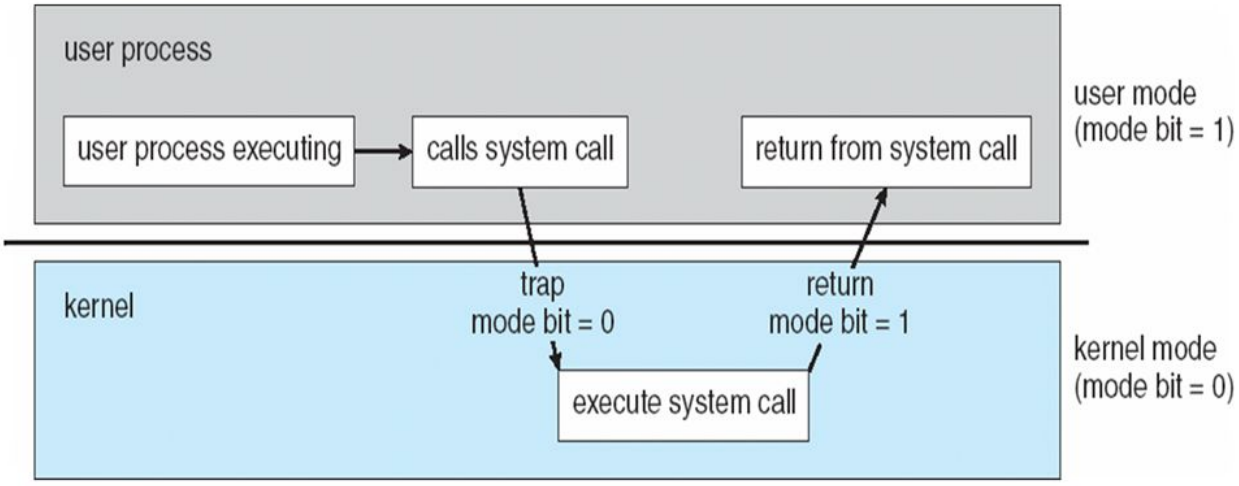
Dual Mode of Operation

Operating System Modes were created to distinguish between user and OS code. We need at least two modes:

- Kernel Mode
    - Also called supervisor/system/privileged mode

- User Mode

A switch bit (0=Kernel or 1=User) is usually used to switch between the two. When the system is executing an application, it is in user mode but to complete a system call by the application it must switch to kernel mode:



Helps achieve protection from one user to another and from user to OS. Concept can be extended by using more bits:

- Virtual Machine Manager
    - More privileges than user but less than kernel

A timer is used to prevent getting stuck in an infinite loop in the user program:

- The computer is interrupted after a small amount of time like 1/60th of a second.
- A variable timer can also be used along with a counter so keep interrupting after every millisecond 3000 times

Memory Management

A program (passive) in execution is called a process (active). It needs resources and it is managed in the following way:

- Scheduling processes and threads on the CPU
- Creation and deletion of user and system processes
- Suspension and resuming of process
- Process communication and synchronization

Similarly memory has to be managed:

- CPU can access main memory directly hence all processes are stored there
- Several programs are stored to save time while multitasking
- OS has to keep track of memory being used by process
- Deciding which process to keep and remove from memory; Allocate and deallocate as necessary

Storage Management

Storage Management is different than memory management. It consists of:

- Providing a logical view of information relating to storage
- Abstract the complex and define simple storage units like files and folders
- File is a collection of related information and a folder is a collection of files
- Hence OS manages creation and deletion of file and directories
- Also backing up files onto nonvolatile storage media

Mass Storage Management refers to the secondary storage containing backup of the main memory.

- Hence speed is important for this subsystem
- Tertiary storage can also be of use though they vary according to their use as WORM (Write Once Read Many) or RW (Read Write)
- OS have a choice to leave tertiary storage to application programs

The fastest way to serve memory is through caching.

- Any information needed is first checked there for a hit and if it is a miss then it is requested from the main memory.
- Very useful for local servers to avoid going to the global server.
- Hardware caching is not in OS control

Due to the small size of cache, cache management is necessary. OS must select size carefully and an efficient replacement policy. Movement of information between the levels may be implicit or explicit:

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

A problem arises known as the cache coherency problem:

- Different levels may have different updated informations of the same file
- Poses a problem for multiple process execution, hence change in one level must be propagated to the other levels as well

- Even more complicated for multiple cores as each cache must be updated with the new value

I/O Management

OS must do the following:

- Hide details of hardware characteristics from user and other applications
- Buffering caching and spooling (keeping a list of documents to be printed for instance and sending to the hardware one at a time)
- Creating a general device-driver interface
- Managing drivers for specific hardware devices

Protection & Security

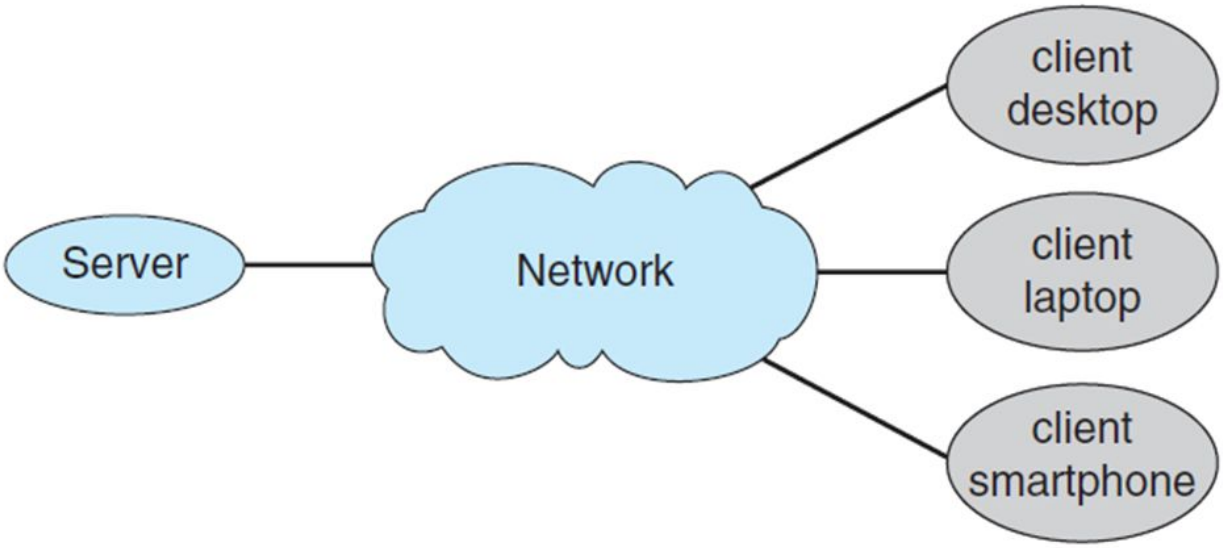
Protection is to protect resources allocated for one user from other users in a multiuser environment. Security is to defend the system against internal or external attacks such as viruses.

- To achieve protection, users must be assigned unique ids, security id (SID for windows)
- In UNIX a group id is also possible to set that a group of users can only read this file
- To escalate privileges, you can use a program with a different id (SetUID)

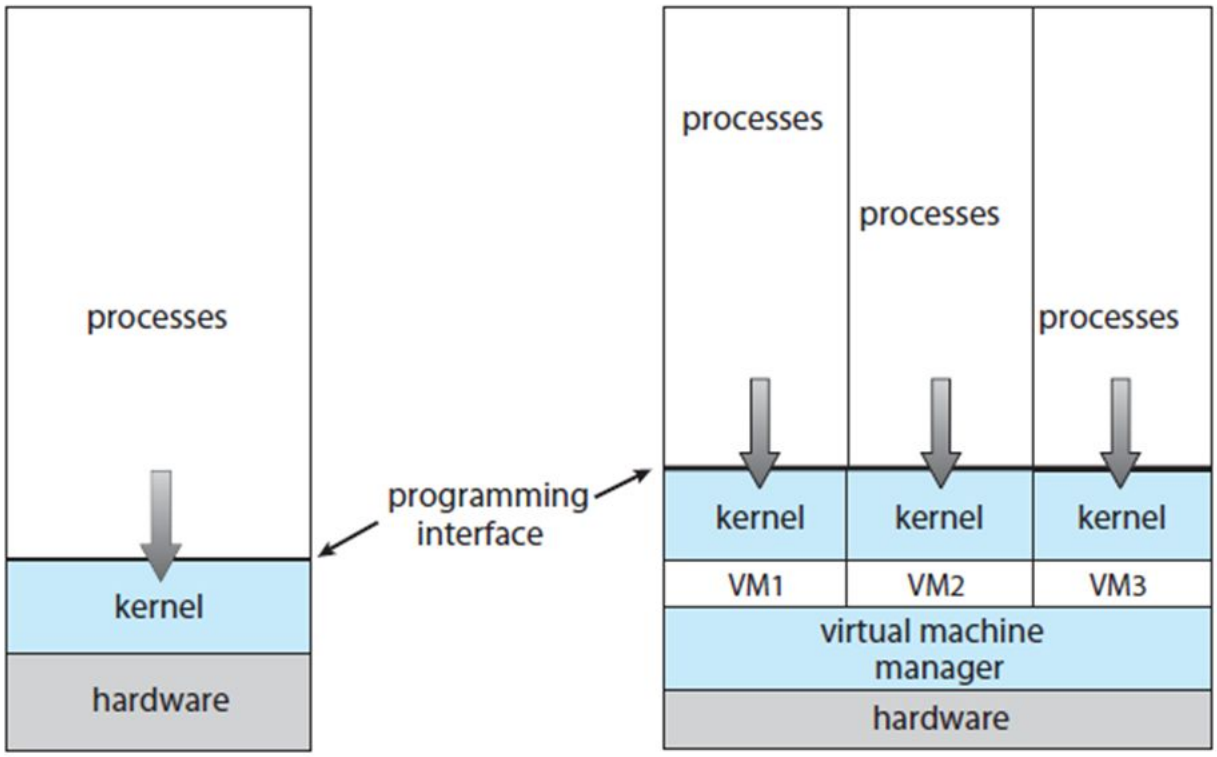
Computing Environments

There are several types of computing environments:

- Traditional Computing
    - No longer exists as it used to with the rise of networks and wireless connections
- Mobile Computing
    - Increasing in applications
    - Memory capacity and power consumption are limited
    - IOS & Android
- Distributed Systems
    - Physically separate, possibly heterogeneous, computer systems networked to provide access to resources
    - A network operating system provides the illusion of these systems acting as one
- Client-Server Computing
    - Only responds to requests by clients
    - Can serve file or perform an action
    - File servers provide a file system interface



- Peer To Peer Computing
    - Each node acts as a server and a client
    - Server bottleneck is resolved
    - A discovery protocol is setup to get the broadcasted request done by the most efficient peer. Discovery protocol may depend on other clients or a centralized system
    - Skype or WebRTC
- Virtualization
    - OS runs an application within another OS
    - Emulated mostly since source CPU type is different from target CPU type - much slower than native code running
    - Java Virtual Machine running on top of the CPU machine
    - Portability increases



- Cloud Computing
    - Extension of virtualization - serve storage, computing or applications across a network
    - Different types of clouds:
        - Public Cloud run for anyone willing to pay
        - Private Cloud run by companies
        - Hybrid includes both aspects
    - Services offered:
        - SAAS: Serving a word processor via the Internet
        - PAAS: Collection of software (software stack) such as a database server
        - IAAS: Storage available for backup
        - Different combinations of these three
    - Cloud Management Tools are also provided
- Real Time Embedded Systems
    - Most prevalent found everywhere. Run real time operating systems (RTOS)
    - Little or no user interface
    - System just fails if the time requirements are not met for execution

OS Structure

There are three main things that make the structure of an OS. The services it provides, the interface and the components it has along with their interconnection.

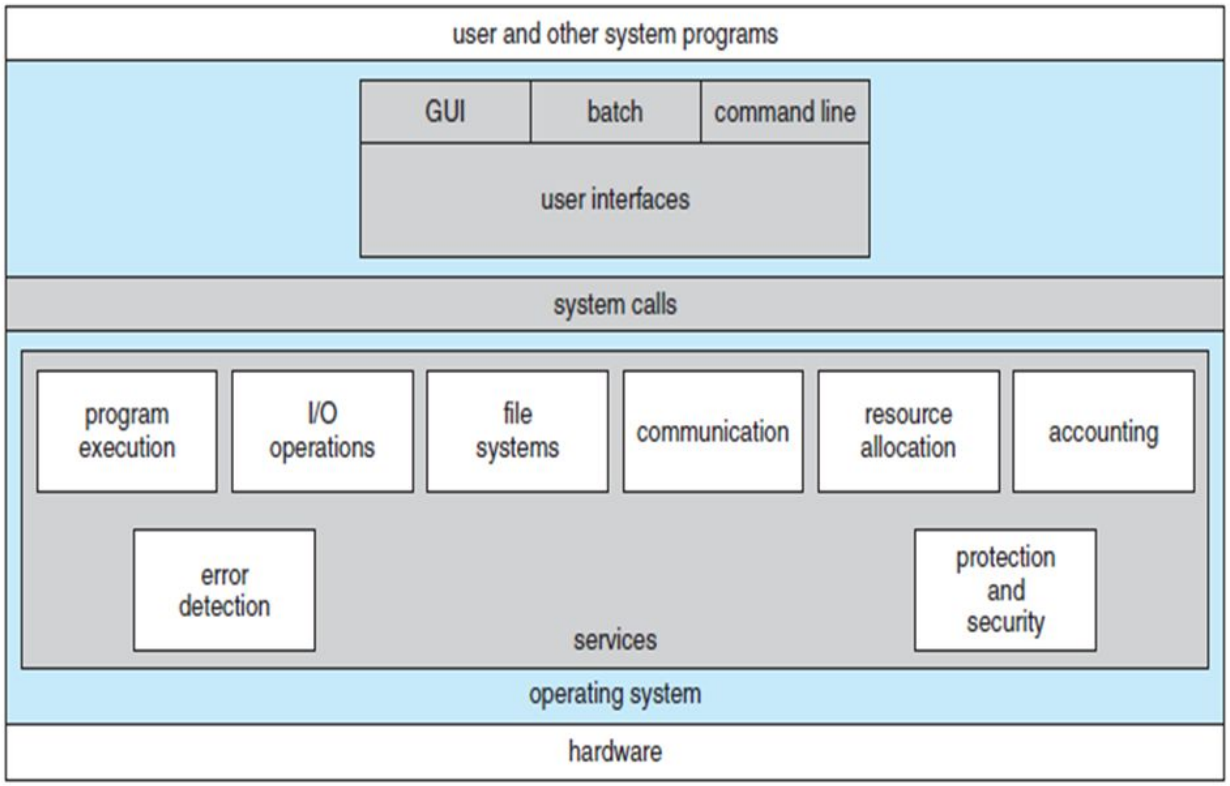
OS Services

Common services provided for users are:

- User Interface (CLI, Batch Interface & GUI)
- Program Execution (Run safely and check if ended abnormally)
- I/O Operations
- File System Manipulation
- Communication (between processes - message passing or shared memory)Error Detection

Common services for the system itself:

- Accounting (Record keeping for future planning)
- Resource allocation
- Protection and Security



OS Interface

The interface provided by the OS for the user can be in the form of:

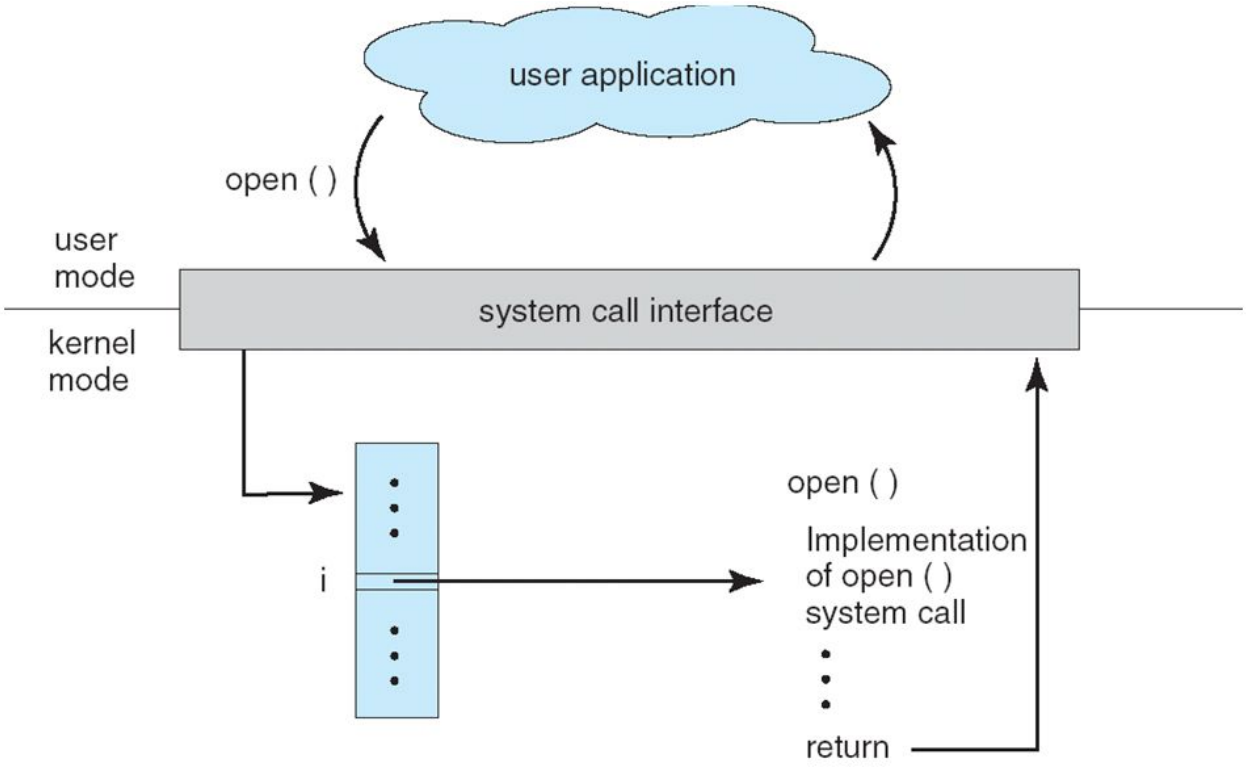
- Command Interpreter
    - Directly put in commands
    - Shell or terminal

    - Either the commands are executed by itself or implemented through system programs - interpreter only identifies which program to load
- GUI

OS Components

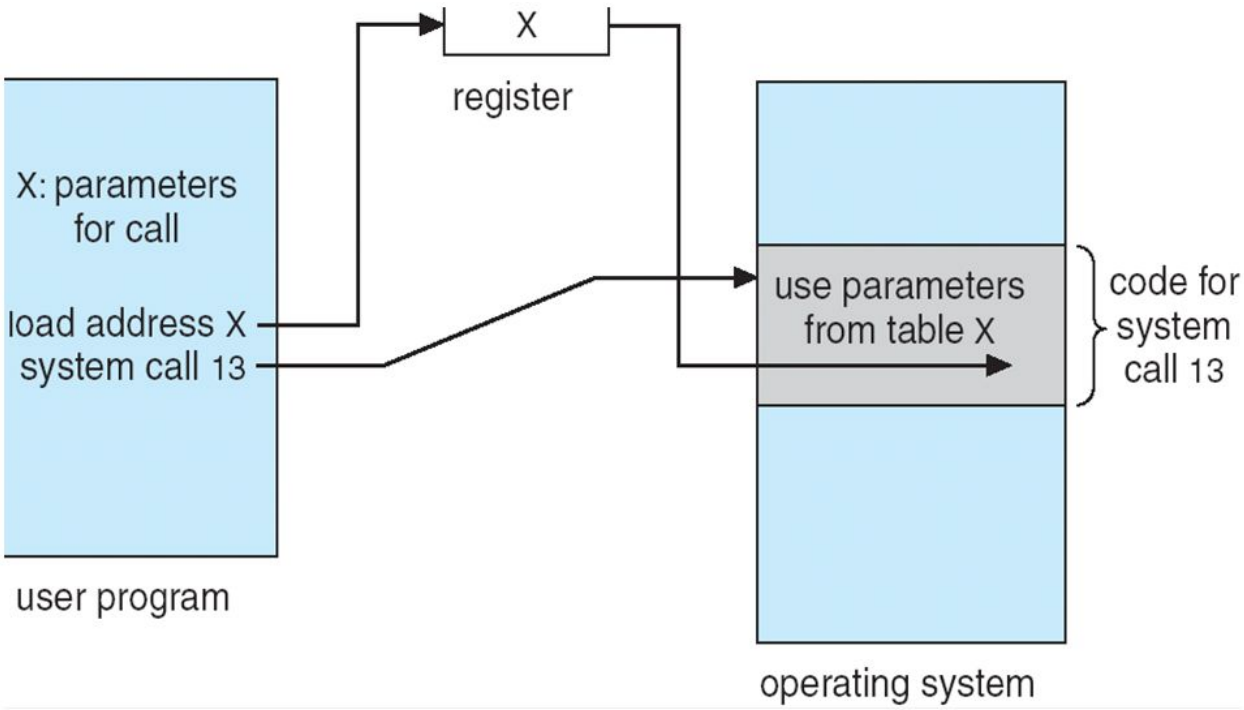
System Calls are not usually directly accessible hence OS does the following:

- Provide API for services made available by OS which provides portability across OS
- Examples are Windows API, Java API etc
- System Call Interface (SCI) has support for most programming languages



The procedure has different implementations:

- Parameters are passed in the registers
- Place in a block or table of memory and address of block is passed in register
- Push parameters on stack which are popped by OS routine



There are six types of system calls:

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

There are different types of system programs/utilities:

- File management - move copy rename

- Status information - date, time, disk space etc
- File modification
- Programming Language Support - Compilers/Interpreters, assemblers and debuggers
- Program Loading and Execution
- Communication related - network creators
- Background services (daemons) - process scheduler

Design & Implementation

First we have to define the design goals and specifications. These are affected by the choice of hardware and system (General Purpose or Specific):

- User Goals
    - Convenient and easy to use
    - Reliable, safe and fast
- System Goals
    - Easy to design and maintain
    - Flexible, reliable and error free

These are very vague and hence there is no unique solution to assess the requirements. We do however have policies and mechanisms:

- Mechanism
    - How to solve the problem
    - Use the timer construct for CPU protection
- Policy
    - What exactly to do
    - How long is the timer to be set
    - Changes according to the situation
- Microkernel based OS usually separate the two and come up with policy free mechanisms

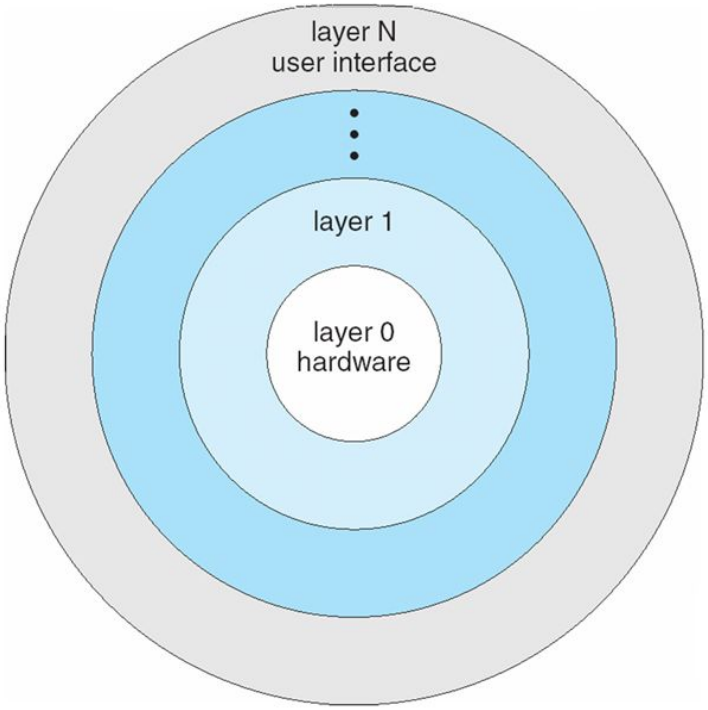
The implementation of all modern OS is done in HLL with the following results:

- Easier to code, debug and portable
- Can be written in multiple languages for different applications
- Reduced speed and increased storage requirements
- Bottleneck routines can be identified

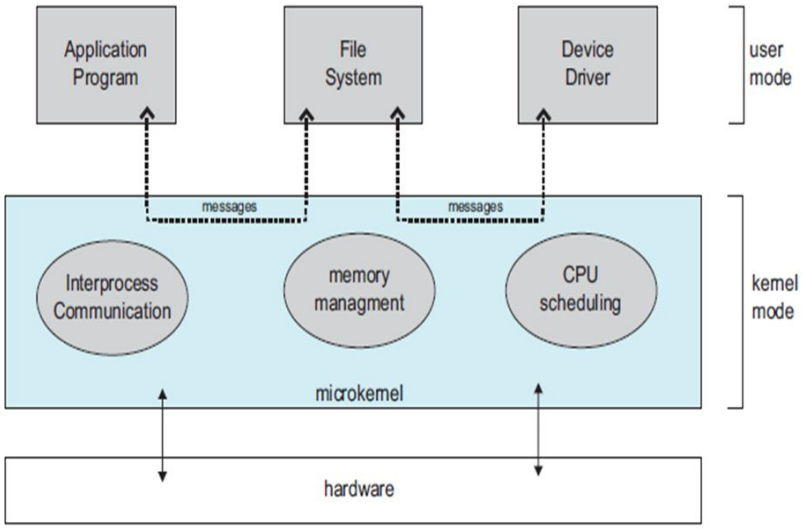
There are several types of structures followed. The most common of which are:

- Simple / Monolithic System
    - No separation between interface and different levels of functionality
    - Crashes often
    - MS-DOS/Original Unix
- Layered Approach
    - Bottom is hardware and top is user interface
    - Layer M can contain the api for HLL to invoke system calls
    - Simplicity of construction, debugging and simplifies system verification
    - Problems are defining the layers clearly, less efficiency in performance

    - Thus fewer layers with more functionality are better

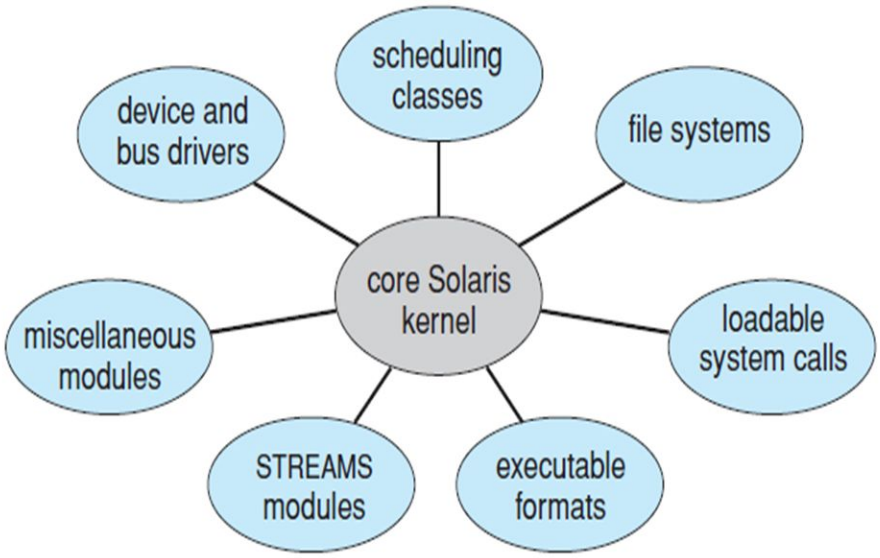


- Microkernels
    - Remove all nonessential components and reimplement as system and user level programs
    - Smaller kernel means minimal process/memory management
    - Communication is through message sending
    - Easier to extend the OS and port. More security and reliability
    - But less performance
    - Tru64 Unix and QNX



- Modules
    - Current methodology is to use loadable kernel modules

- Core components link with additional services during boot time or runtime
- Unix - Solaris, Mac OS X
- Solaris is a core kernel with seven types of loadable modules



- Hybrid
  - Most OS balance according to performance, security and usability
  - Linux. Solaris are monolithic but also modular
  - Android and IOS are also examples

Process Management

There used to be only one program executing at a time which was called a process, but now there can be several.

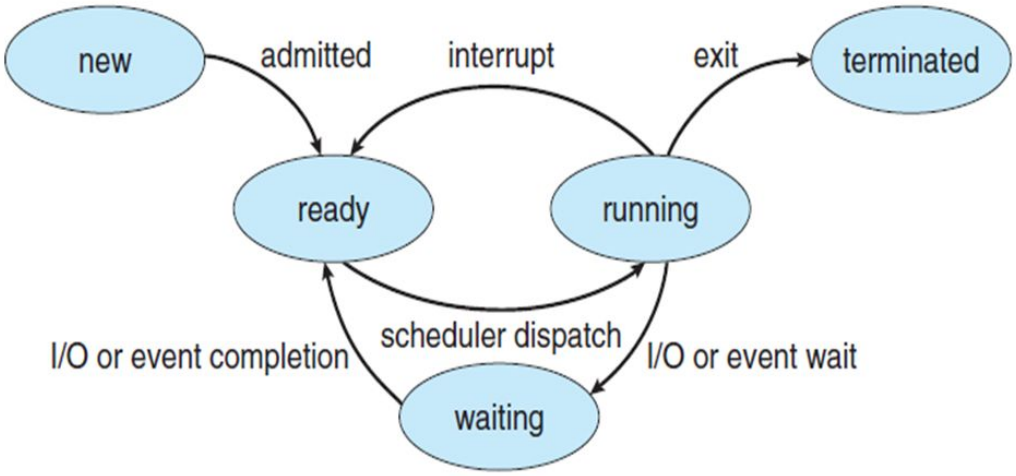
Process Concepts

The first thing to remember are the basics:

- OS processes and user processes are treated differently
- Job and process are used interchangeably
- Process is made of:
  - Program code or text section
  - Program counter and register contents
  - Process stack
  - Data section
  - Heap (optional)
- Two processes can be associated with the same program but will still be considered different processes

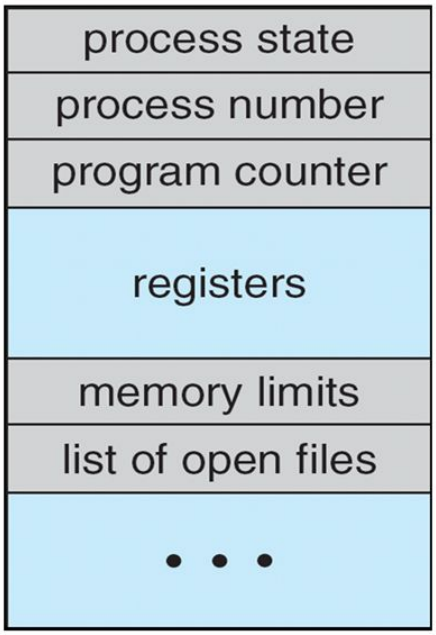
There are different states for processes:

- New: Being created
- Ready: Waiting to be assigned to processor
- Running: Instructions are being executed - ONLY ONE CAN RUN
- Waiting: For an event
- Terminate: Completed execution

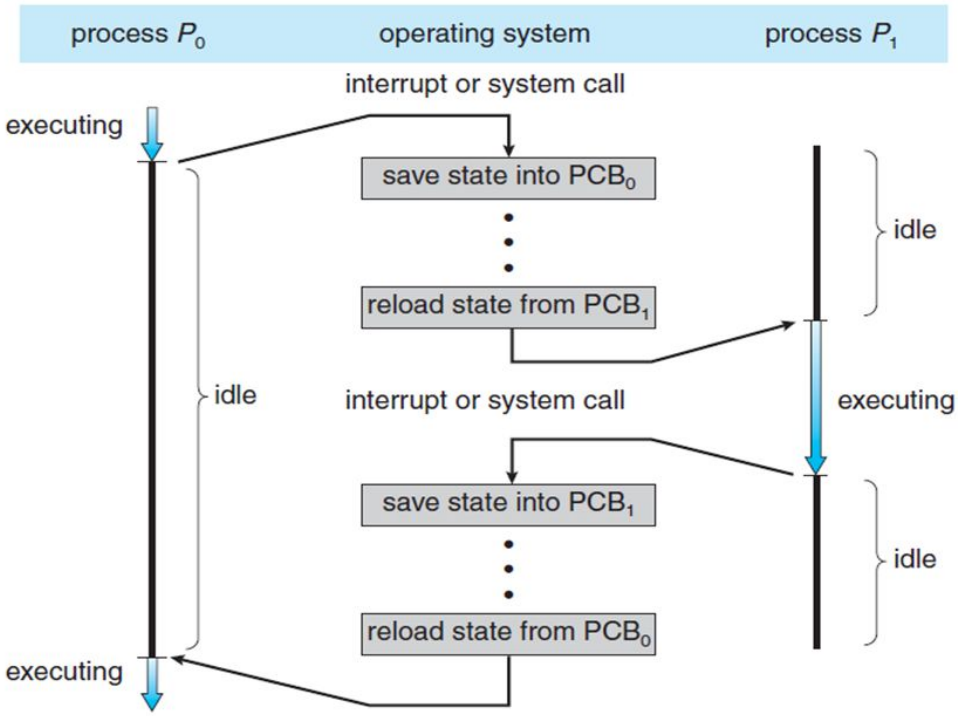


The process control block (PCB) represents each process. It serves the information depending on the process:

- Process state
- Program Counter
- Registers
- Scheduling Information
- Memory Management, I/O, Accounting info



This helps in easily switching between processes:



Process Scheduling

Time sharing is used to keep switching between processes so rapidly that the user does not notice, and multiprogramming allows for maximum CPU utilization. The process scheduler performs both of these and chooses the next process from a set of available processes.
There are many scheduling queues:

- Job queue consists of all processes in the system
- Ready queue has all ready state processes in main memory (Generally linked list pointing to PCB)
- Device queue are waiting for a particular device. I/O queue for processes who made an I/O request etc
- In case a process makes a child process it is usually placed in waiting queue for the child's termination
- An interrupt is caused by the scheduler and the process is sent back to ready queue, it keeps going through the cycle until it terminates and releases all resources

There are two types of schedulers:

- Long Term
    - Select job from pool and load into main memory
    - Controls degree of multiprogramming (dom) if DOM is stable then average rate of process creation = average rate of process departure
    - There are two types of processes mainly, those which rely mostly on I/O and those which do on computation, long term scheduler must mix the two equally

        - Some systems do not require a long term scheduler due to their physical limitation so they rely on the self adjusting nature of humans
    - Short Term
        - Select processes ready in memory
        - Executes with much higher frequency hence should be faster
    - Medium Term
        - In some time sharing systems
        - Beneficial to remove a process from DOM (swapping)

A context switch can occur when the CPU is interrupted and needs to change into kernel mode:

- It saves the state of current process in its PCB and reloads after coming back
- It also occurs when switching between processes
- Context switch time depends a lot on the hardware. For instance Sun UltraSPARC provides multiple sets of registers hence a context switch will only require changing the pointer

Operation on Processes

Most systems give us the freedom to create processes dynamically, they also give the following utilities:

- Process may create subprocesses (creating a parent child tree)
- Process identifier (pid) uniquely identifies the process
- Resources are provided for the process by OS, child can take subset of resources of parents
- When a parent creates a child process:
    - It can run concurrently or wait until children have terminated
    - Similarly the child can be a duplicate of the parent or be a different program loaded from memory
    - Calling fork in unix vs CreateProcess('C:\\ms paint') in windows
- Process termination can be done in the following ways:
    - exit() - deallocate all resources and return status value
    - Caused by another process (only parent) through system call
    - Because it may have exceeded in resource usage
    - No longer required
    - Parent is exiting hence OS will not allow child to exist

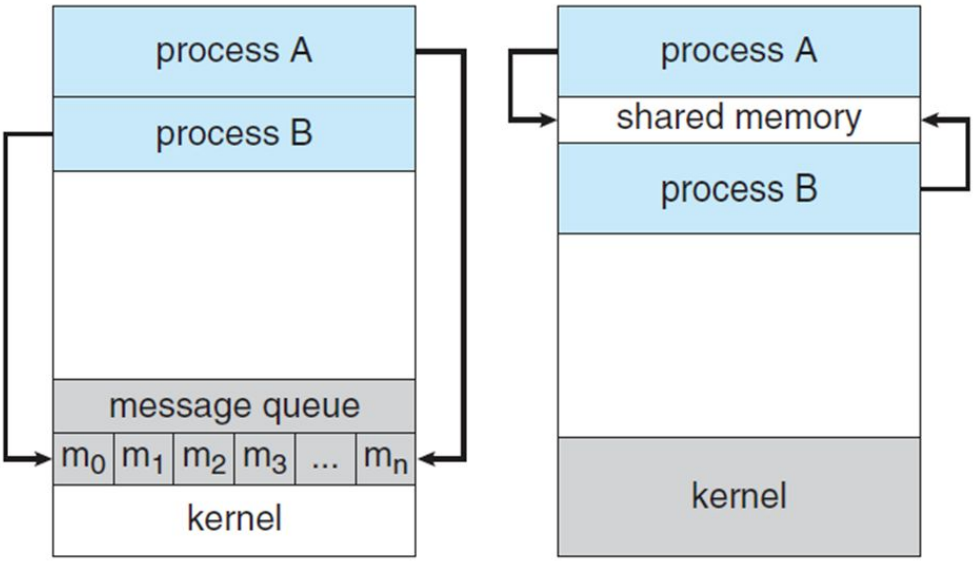
Interprocess Communication

Sometimes processes executing concurrently are not independent but rather cooperating. They can affect each other. There are several reasons for it:

- Information Sharing
- Computation speedup - program is broken into subtasks executing in parallel
- Modularity is achieved
- Convenience for users to execute several tasks in parallel

Hence in Interprocess Communication (IPC) mechanism is required:

- Shared Memory Model
    - A region of memory is created that can be shared
    - Maximum speed and convenience of communication
    - System call only used to establish the region thus no more assistance is required from Kernel
- Message Passing
    - Messages are exchanged using message passing systems
    - Only small amounts of data can be sent
    - Easier to implement in distributed systems and better performance in multicore systems
    - Message passing systems are implemented using system calls

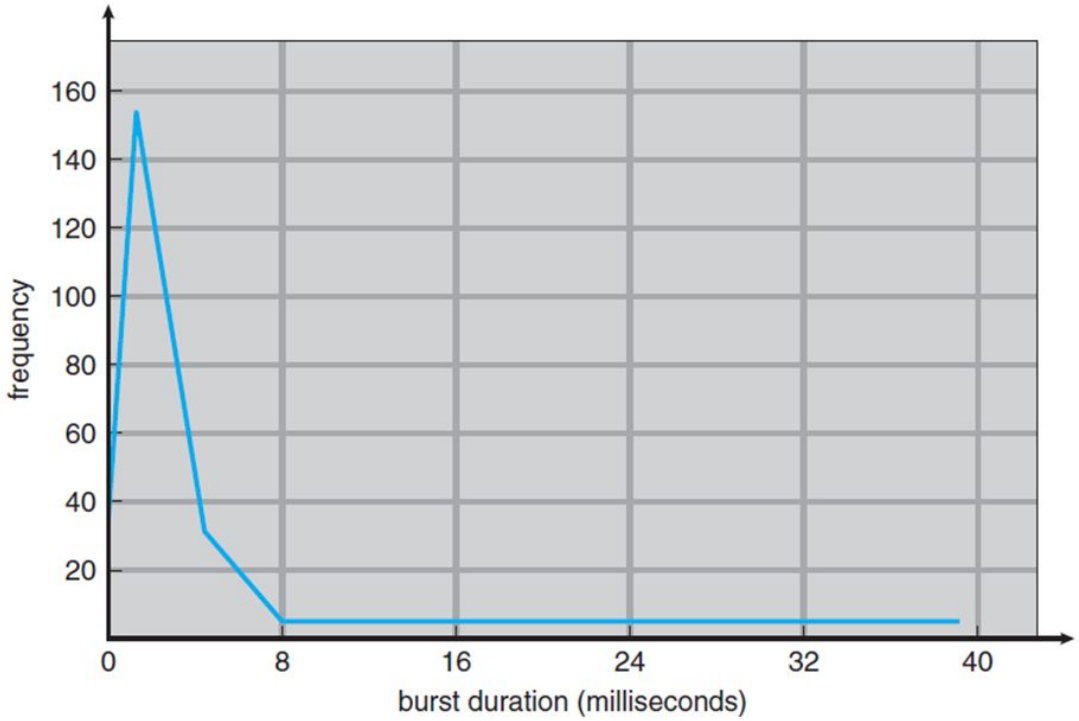


CPU Scheduling

There are different types of scheduling algorithm implemented for maximum efficiency in process execution.

Introduction

- Basis of multiprogramming OS
- In single process environment, CPU is idle when process is waiting for I/O hence multiprogramming helps us use this time productively
- Another program running when the first one is waiting would be a basic CPU scheduling algorithm
- Process execution can consist of CPU execution and I/O wait (CPU burst then I/O burst)
- Duration can be exponential or hyper exponential



- There are either a small number of long CPU bursts (computational process) or a large number of short CPU bursts (I/O driven process).
- A CPU scheduler:
    - Also called the short term scheduler which activates whenever the CPU becomes idle (during I/O burst).
    - The ready queue in the main memory may be FIFO, priority queue, tree or unordered linked list
    - The queue consists of PCB
    - The scheduling can occur for 4 situations:
        - Process switches from running to waiting
        - From running to ready
        - From wait to ready
        - Terminates
    - For 1 and 4 the CPU has no choice (non-preemptive) but for 2 and 3 it has a choice (preemptive). The two can further be differentiated as:
        - In non-preemptive CPU can only release if the process goes into wait state or terminates
        - In preemptive, OS can interrupt the CPU and put process in ready
    - Preemption affects the OS kernel design when processing a system call as it has to wait for completion before preempting hence a poor model for real time computing. Another solution is to disable interrupts.
- The dispatcher:
    - Gives control of CPU to process selected by scheduler
    - It can perform the following:

        - Switch Context
        - Switch to user mode
        - Jumping to proper location to restart the program
    - It is frequently used hence should be fast
    - Dispatch latency is the time taken to stop one process and start the other

Scheduling Criteria

The following criterias are used to select between the algorithms:

- CPU utilization, in real time it is between 40% (lightly loaded) - 90% (heavily loaded)
- Throughput, a system of measurement for processes completed per unit time (work done)
- Turnaround time, time taken from submission to completion + computation + waiting
- Waiting time
- Response time, time taken to respond (independent from speed of output device)

Some desirable characteristics are:

- Maximize 1 and 2
- Minimize 3, 4 and 5

For interactive systems it is necessary to minimize variance in time and have a predictable response time (more appealing to the human nature)

Scheduling Algorithms

First come first serve:

- Implementation in FIFO queue
- PCB of ready process is linked to tail of ready queue
- When CPU is free it is allocated process at the head of the LL
- Average waiting time is long and highly dependant on order of arrival
- Non-preemptive hence not suitable for time sharing system

Shortest Job First:

- Optimal scheduling algorithm as it gives the minimum waiting time
- Problem is in prediction of CPU bursts
    - Past stats can be used to predict but they aren't accurate
    - Formula used is $T_{n+1} = \alpha\, t_n + (1 - \alpha)\, T_n$
    - Where Tn+1 is time for next process tn contains most recent info and Tn contains past info. a controls relative weight and is commonly 1/2
- Hence it is better to be used in the long term scheduler
- Can also be preemptive hence newer process may have shorter time than current process

Priority:

- Priorities are usually indicated by a fixed number (0 for highest priority)
    - Internal priorities are set by OS
    - External priorities are determined by criteria outside of OS control

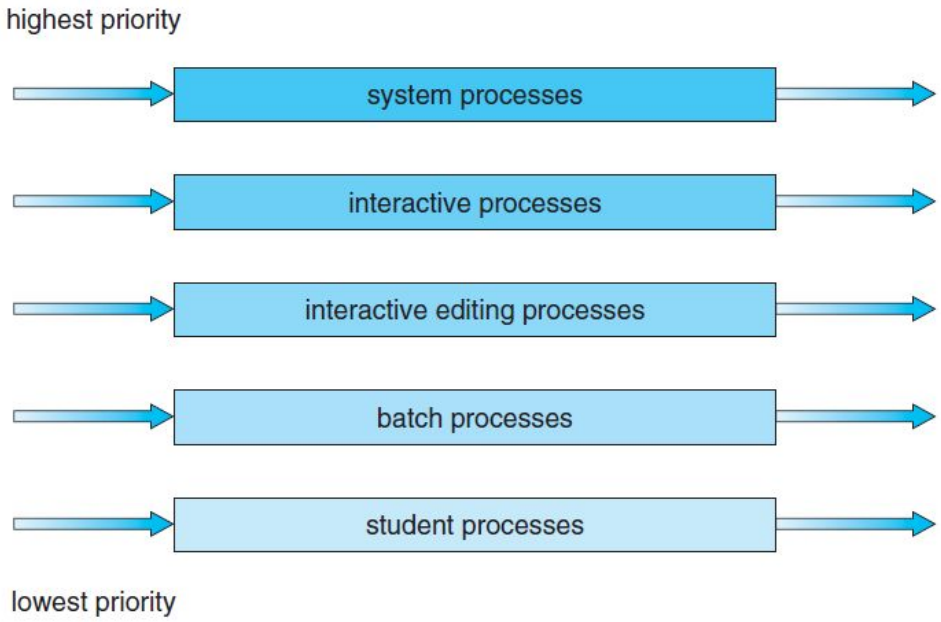
- Can be preemptive or non-preemptive
- Problem is low priority process will have to wait indefinitely (also called indefinite blocking or starvation)
- Aging can be used to solve this problem (gradually increase priority of processes waiting in system for a long time)

Round Robin:

- Designed especially for time sharing algorithm
- Time quantum or slice is defined generally between 10-100ms
- Ready queue is treated as a circular list
- Har process ko waqt do warna wo RR daal dega
- Problems are:
    - Each process gets 1/n of cpu time (n=items in queue)
    - Wait (n-1)*q time units until next quantum
    - For very large time quantum RR is close to FCFS
    - Whereas smaller time quantum would make context switching happen too quickly causing performance problems
    - Hence rule of thumb is quantum should be greater than 80% of CPU burst

Multilevel Queue Scheduling

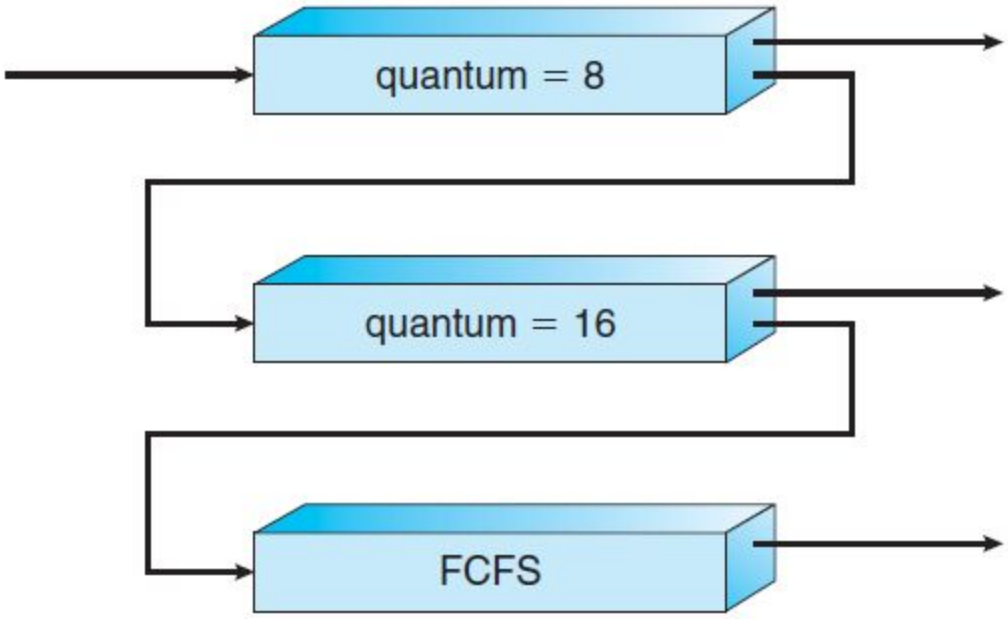
- Partitions ready queue into several separate queues
- Each queue has its own scheduling algorithm
- A common separation would be for foreground (interactive) and background (batch) processes with foreground having priority, externally. Internally, they both may have different scheduling algorithms such as RR for foreground and FCFS for background.
- Scheduling among the queues is commonly priority based (preemptive).
- An example containing 5 queues:



- The batch processes will only run when the system, interactive and interactive editing processes ready queue is empty. Moreover, a process which arrives later in the higher queue will preempt the batch process.

Multilevel Feedback Queue Scheduling

- Normally processes are assigned permanently to the queue but in MFQ processes can move between queues.
- Processes that have longer CPU bursts are moved to lower priority queues. (Aging can be used to prevent starvation)



- A process in the first queue, which does not finish in 8 quantums will be moved to the tail of the lower queue. Will keep moving to the bottom.
- The following parameters are what any MFQ depends on:
    - Number of queues.
    - Scheduling algorithm for each queue.
    - Method to determine where to place new process.
    - Method to determine when to upgrade (aging) or demote an algorithm.
- Most general but complex algorithm, as we require some way to determine the best parameter values.

Thread Scheduling

- There are user level and kernel level threads.
- OS that support threads actually schedule Kernel level threads and not processes.
- User level threads are managed by a thread library and ultimately mapped onto the Kernel level as a Light Weight Process (LWP).

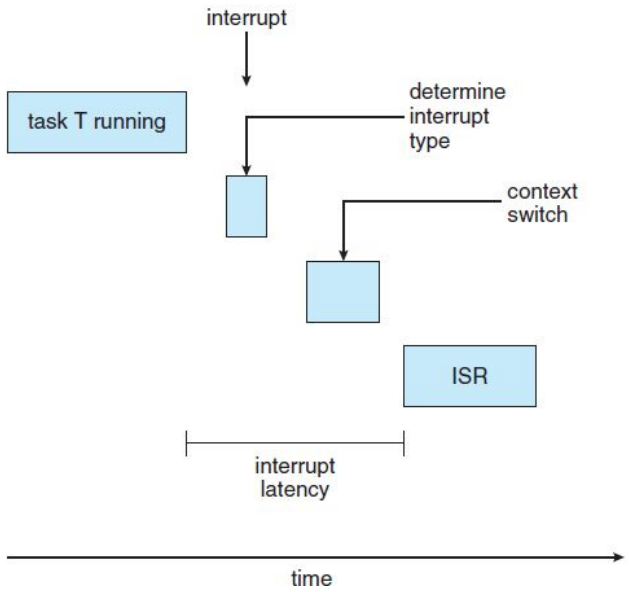
Multiprocessor Scheduling

- With multiple processors load sharing, comes great scheduling complexity.
- There are two approaches to MS:
- Asymmetric Multiprocessing
    - Like threads, two different works are handled by the different CPUs.
    - One processor could access the system data structures reducing the need for data sharing.

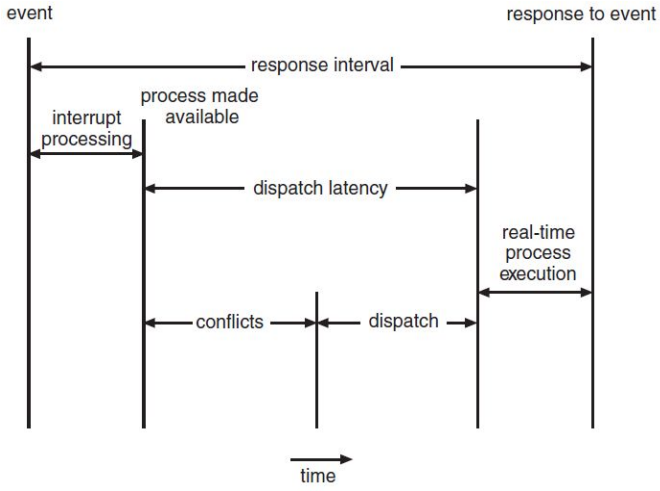
    - All scheduling decisions are made by a single processor known as the master server.
- Symmetric Multiprocessing
    - Each processor is self scheduling.
    - All processes are in a common ready queue or in each processor's private ready queue.
    - Since we have multiple processors, the scheduler must be programmed carefully considering the following factors.
- Factors affecting scheduler
    - Load balance among processors, especially in systems where there are private queues. Usually done by:
        - Specific task which checks load on each processor periodically. Also pushes or moves processes in case of imbalance onto idle or less busy processors (Push Migration)
        - Duty of idle processor to pull waiting task from a busy processor (Pull Migration)
        - These need not be mutually exclusive. Can both exist.
    - Avoid memory stalls as when a processor accesses memory, it may result in a cache miss which entails waiting for the data to be available.
        - Multithreaded (hardware thread) cores are implemented in each processor.
        - When one thread waits for the memory to be available, the core can switch to another thread.
    - Multithreaded Multicore processors requires two different levels of scheduling.
        - OS decides which software thread to run on each processor hardware thread.
        - Each core decides which hardware thread to run.

Real Time CPU Scheduling

- Two types of systems:
    - Soft real-time systems. These give priority to critical tasks but have no guarantee of when it will be scheduled.
    - Hard real-time systems. These have a strict policy of a deadline. Executing after a deadline is the same as not executing at all.
- Problems faced are:
    - Minimizing Latency while facing real time events. A car brake system may have 3-5 ms to respond and control the emergency call whereas an embedded system controlling an airliner might tolerate a few seconds delay.
    - Two types of latencies are:
        - Interrupt latency. Time of arrival of interrupt at the CPU to start of routine that services the interrupt which consists of determining type of interrupt and context switching.



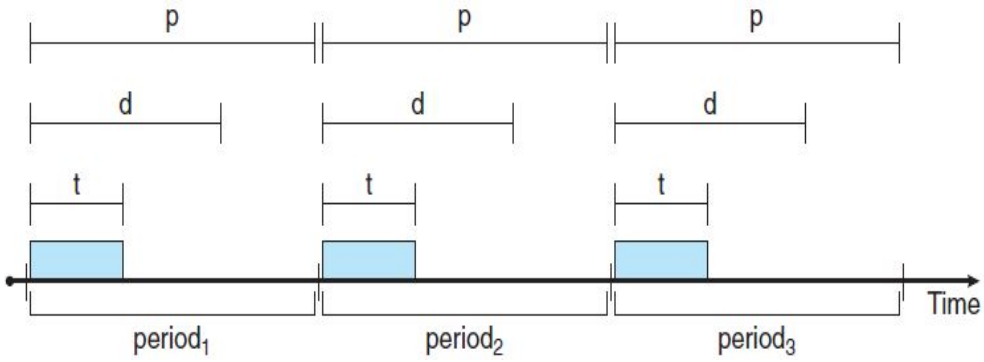
- Dispatch latency. Time required to stop the current task and start the new one. Provide preemptive kernels to minimize this latency.



- Conflict phase has two components:
  - Preemption of any process running in the Kernel
  - Get resources from low priority processes

Real Time Priority Based Scheduling

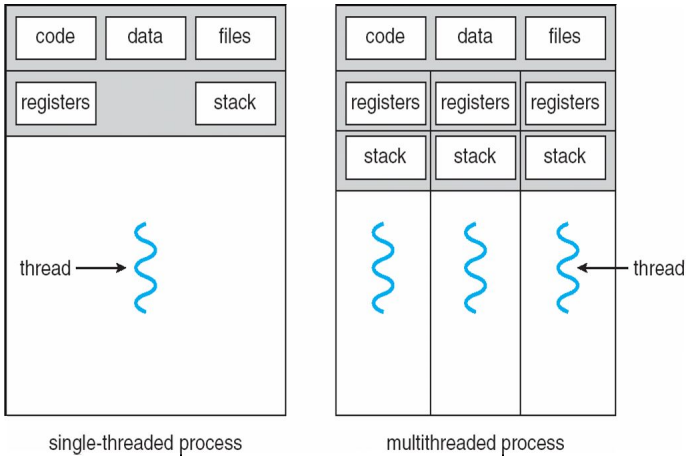
- Must support preemption
- Only guarantee soft real time functionality and requires additional features for hard real time functionality.
- Processes are considered periodic. Process that requires CPU for constant intervals or periods of p has fixed processing time t. Also has a deadline d by which it should be serviced by the CPU and its rate is 1/p. $0 \leq t \leq d \leq p$



Threads

Overview

- A thread is a basic unit of CPU utilization which has:
  - Thread ID
  - Program Counter
  - Register Set
  - Stack
- A single process may contain multiple threads of control.
- Shares all resources with threads of same process



Advantages

- Different threads for different tasks in a single process.
- Creating a separate process is time consuming and resource intensive.
- IPC is more efficient between multithreaded processes (dedicated threads for IPC)

- OS kernels are also multithreaded.
- Response time of interactive programs is improved.
- Resource sharing is simple.
- Threads can run in parallel increasing concurrency

Challenges

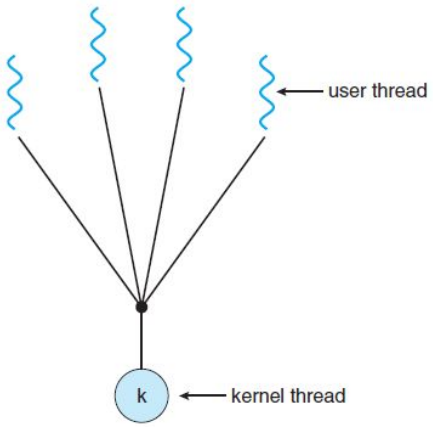
- Programmers must change existing apps or design new ones to fit the multithreading paradigm. Also a problem for system designers.
- There are 5 general areas:
    - Identifying the areas where a task can be divided into separate concurrent tasks. Ideally tasks are independent of one another and can run in parallel on individual cores.
    - Balance of all tasks that are running in parallel. All should be of equal value instead of using a separate execution core to run a task that may not be worth the cost.
    - The data accessed and manipulated by these tasks must be divided to run on separate cores as well.
    - Data accessed by two tasks must be independent. If there is a data dependency then execution of tasks should be synchronized to accommodate the data dependency.
    - Testing and debugging is also difficult as different execution paths are possible.

Parallelism

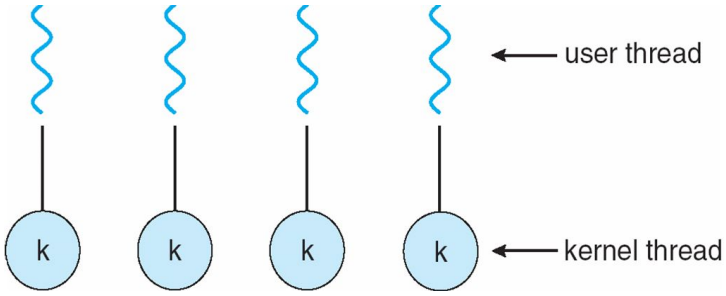
- Data parallelism
    - Focus on distribution of data among multiple cores
- Task Parallelism
    - Focus on distribution of tasks among multiple cores.
- Applications use a hybrid of the two.

Multithreaded Models

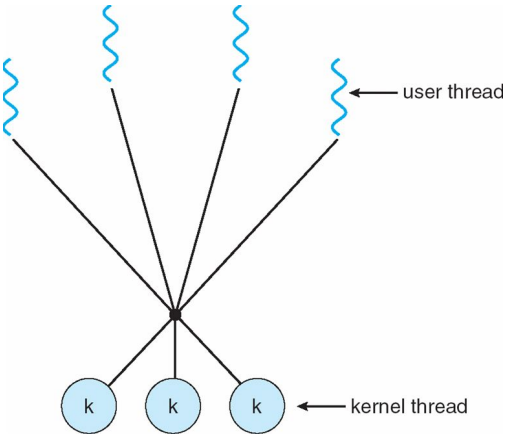
- Many-to-One Model
    - Maps many user-level threads to one kernel thread
    - Thread management is done by thread library
    - If one thread blocks, the whole process will block.
    - Can not run on multiple processors since only one thread can access the kernel at a time.



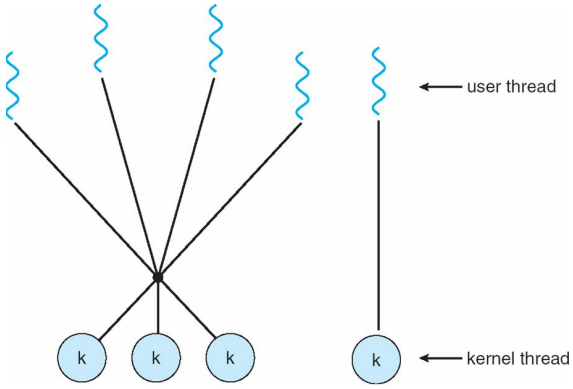
- One-to-one Model
    - Maps each user thread to one kernel thread
    - More concurrency since other threads will run when one is blocked
    - Can run on multiple processors.
    - Creating kernel threads for each user thread can burden the performance of an application. But still implemented by Linux and Windows.



- Many-to-Many Model
    - Maps many user-level threads on smaller or equal number of kernel thread
    - Many to one allows programmers to create many threads but the Kernel only schedules one at a time, one to one restricts the programmer from using too many threads but many to many gives freedom and has less overhead than one to one.
    - In case of blocking, Kernel can schedule another thread



- The two level model
    - A variant of many to many
    - Allows user level thread to be bound to a kernel thread



Thread Libraries

- Provides programmer with api for creating and managing threads
- Two approaches
    - Library without kernel support
    - Kernel-level library specifically for OS
- Examples are POSIX Pthreads, Win32 and Java

Memory Management

Background

- Programs and data must be in main memory before execution
- Several complementary processes are in memory to improve CPU utilization and speed of response to users

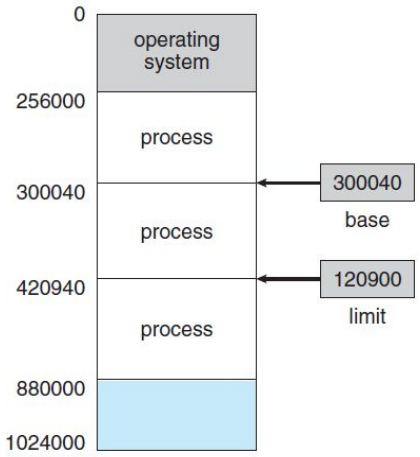
- Memory management schemes help achieve these. It is determined by the hardware specs and design.

Introduction

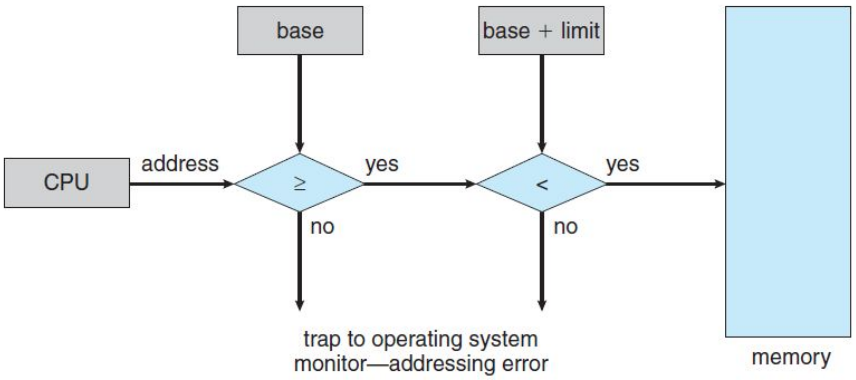
- We need to share memory among processes and find ways of organizing memory.
- A number of ways exist for memory management from paging to segmentation each having its perks.
- Hardware support is required for faster memory management.

Basic Hardware

- Processes should be protected from each other. This protection is provided through hardware.
- Processes use two registers to access only its own legal addresses. Base and limit.
- Base register holds smallest legal physical memory address (minimum) and limit specifies size of range (maximum)



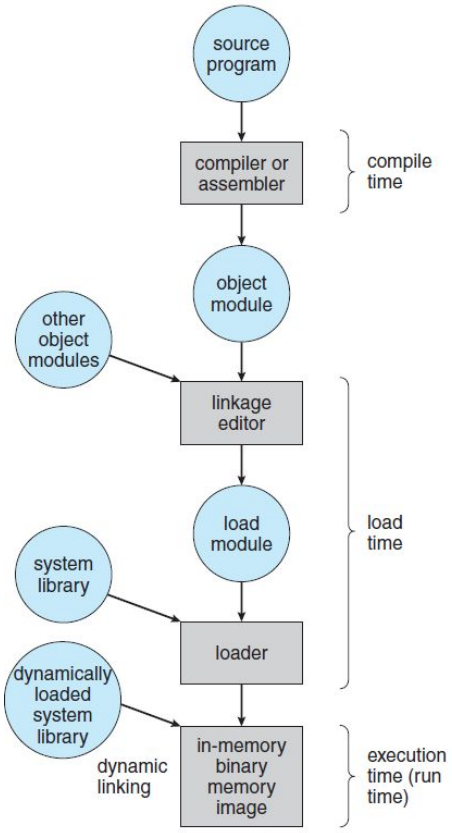
- Attempt to violate legality results in a trap which OS treats as fatal error



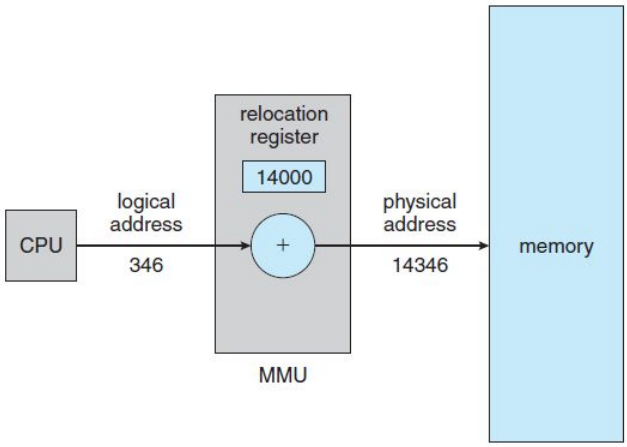
- Base and limit registers can be loaded only by OS using privileged instruction executed in kernel mode.

Address Binding

- Process Cycle
    - Program needs to be brought into memory from disk and placed into process to be executed.
    - Processes may be moved between disk and memory during execution. Processes on disk waiting to be brought into memory form the input queue
    - After execution its space is declared available.
- When a program is compiled the variables are bound to relocatable addresses such as 14 bytes from the beginning of this module.
- The linkage editor or loader in turn binds these to absolute addresses such as 74014
- Binding Cycle
    - If the compiler knows at compile time where the process will reside then the generated compiler code will start at that location. So if the starting location for user processes changes then the program needs to be recompiled. (MS-DOS .COM-format programs)
    - If not known, then the compiler generates relocatable code. Where final binding is delayed until load time. If the starting address changes, we only reload the user code.
    - If the process is moved in and out of memory then the binding is delayed till run time.



- Logical vs Physical Address
    - CPU generated is logical
    - Address loaded into memory-address register is physical
- Memory Management Unit (MMU) does run time mapping from virtual to physical addresses.
- User program never sees the physical address
- Hence logical address ranges from 0 to max and physical is R+0 to R+max for a base value R



Dynamic Loading

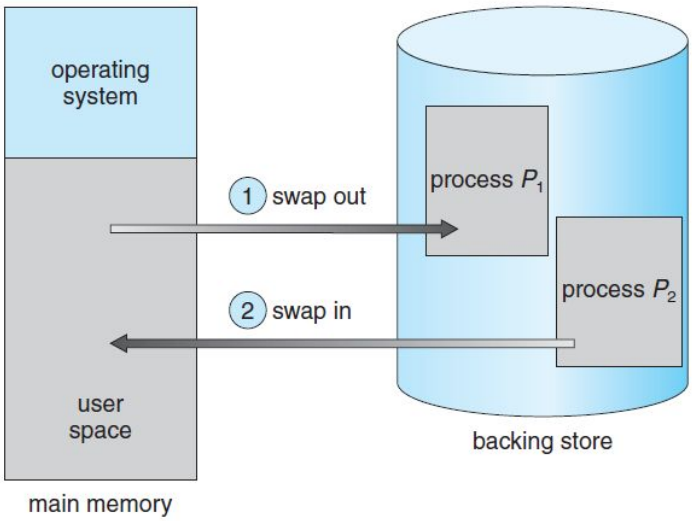
- Size of process in execution is limited to size of physical memory. Dynamic loading allows better memory-space utilization.
- A routine is loaded only when it is referenced.
- Dynamic loading does not require special support from OS but it may provide library routines to implement DL hence users are responsible to design their programs.

Dynamic Linking & Shared Libraries

- Similar to DL
- Linking is postponed till execution time. Hence used for linking system libraries.
- Otherwise each program would have to include the entire library which is a wastage of disk and memory space
- Dependent on OS

Swapping

- A process can be swapped out of memory to a backing store temporarily to increase degree of multiprogramming.
- A backing store is a fast disk hence it makes it possible for total physical address space for all processes to exceed physical memory.
- There are different types
    - Standard swapping
        - System maintains ready queue of all processes in memory or backing store.
        - Dispatcher checks presence of next process in memory and acts accordingly
        - Swap in and out results in high context switch time
        - Major part of swap time is transfer time (amount of memory)
        - A memory to be swapped must be completely idle (not waiting for asynchronous I/O operation)
        - Modified versions of swapping are used in many versions of unix and are used when free memory falls below a threshold



- Swapping on Mobile Systems
    - Do not support swapping since flash memory can only tolerate limited number of writes
    - Hence IOS asks applications to voluntarily relinquish allocated memory and read only data is removed (modified data such as stack is not)
    - Apps that do not comply are terminated.

Contiguous Memory Allocation

- Memory is divided into 2 parts
    - Residing OS
    - User processes
- OS is generally kept in lower memory
- Available memory is allocated to processes contiguously

Memory Allocation

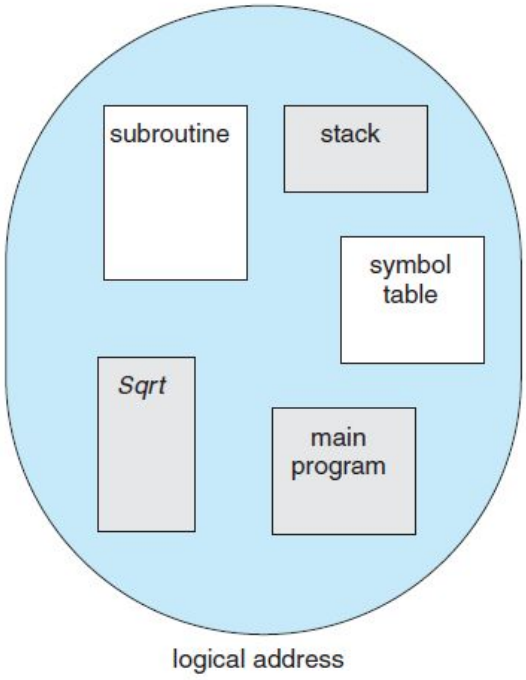
- Simplest method to allocate memory is to divide into several fixed-size partitions each containing one process.
- DOM is bound by number of partitions
- When a partition is free load new process
- OS keeps track of free spaces (holes) in a table
- Usually consists of variable sized holes. How to satisfy a request of size n from a list of free holes?
    - First fit: Allocate first hole big enough
    - Best fit: Allocate smallest hole big enough to contain the process
    - Worst fit: Allocate largest hole available
    - First and best fit are better than worst fit in terms of decreasing time and storage utilization and first fit is generally faster than best fit

Fragmentation

- Both first fit and best fit suffer from external fragmentation - large number of small holes in memory (left after execution)
- Internal fragmentation is when memory has fixed size blocks and allocates multiples of these. Allocated size may be larger than requested size (occurs during execution).
- Statistically, for N blocks allocated through first fit, another 0.5N blocks will be lost to external fragmentation rendering 1/3rd of the memory unusable.
- Solve external by compaction - shuffle free memory spaces to form one large contiguous block. Or by making physical address space noncontiguous using paging or segmentation.

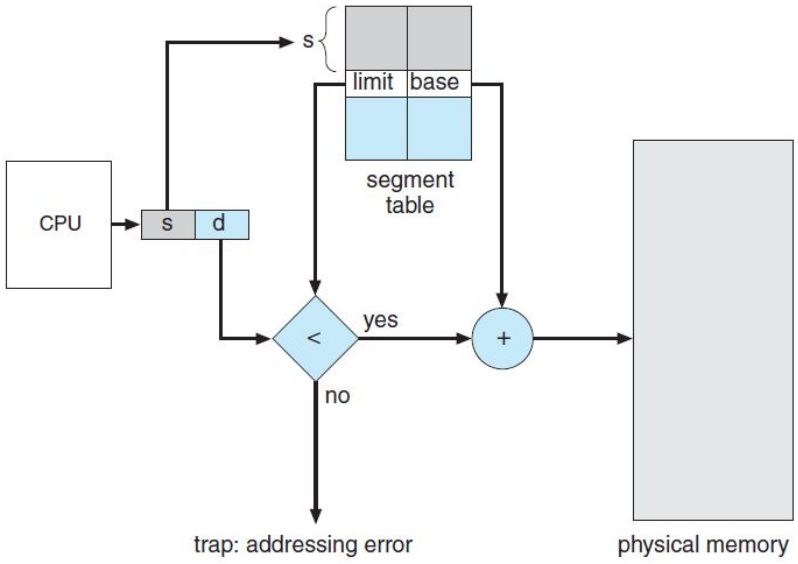
Segmentation

- A programmer sees memory as variable sized segments
- A C compiler creates separate segments for:
    - Code
    - Global vars
    - Heap
    - Stacks for each thread
    - Standard C library

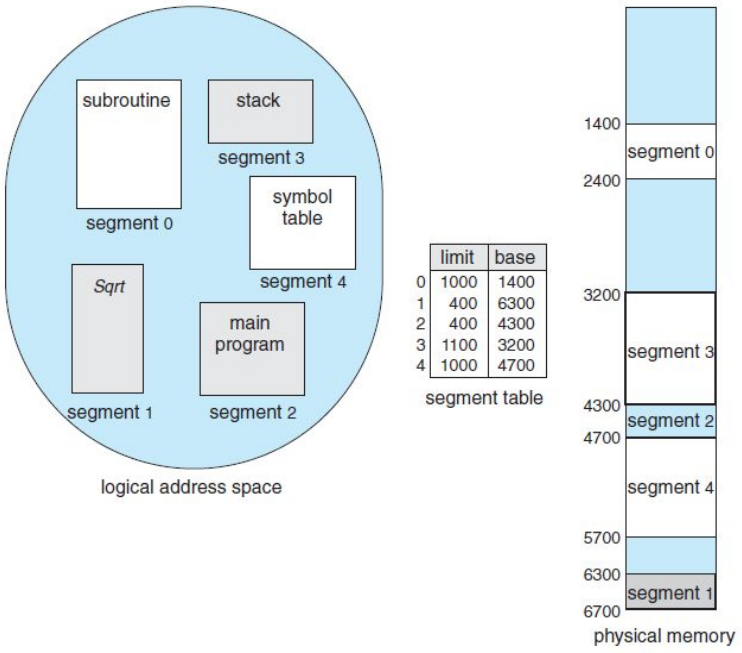


- Segmentation hardware

- A 2 dimensional (segment+offset) logical address needs to be mapped to 1 dimensional physical address
- Mapping is performed through segment table
- Each entry has segment base and limit - segment number s is used to index table and d is offset between 0 and limit

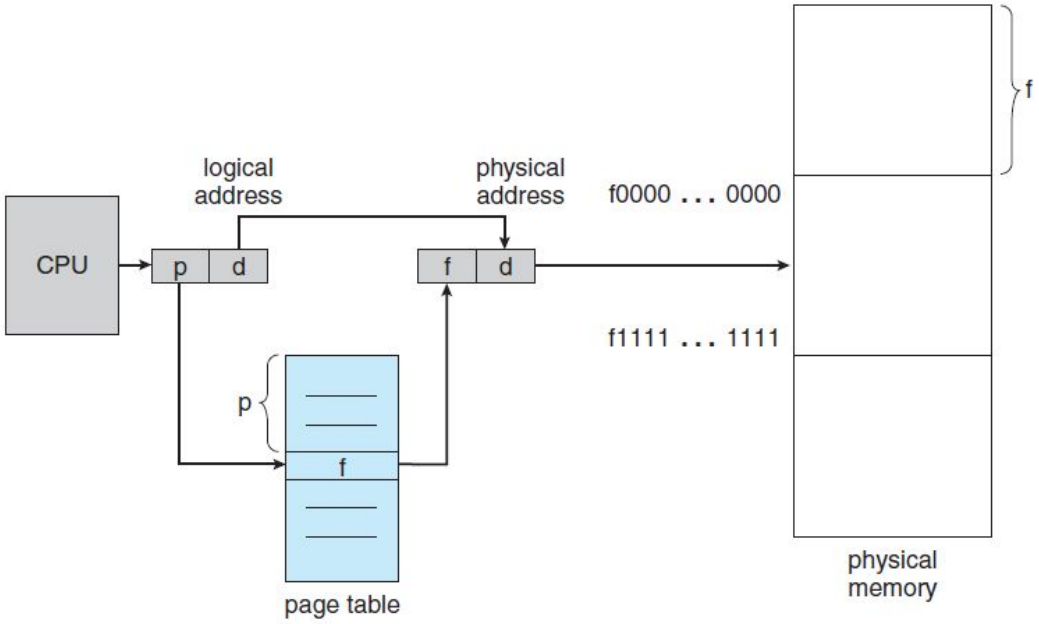


- Example



Paging

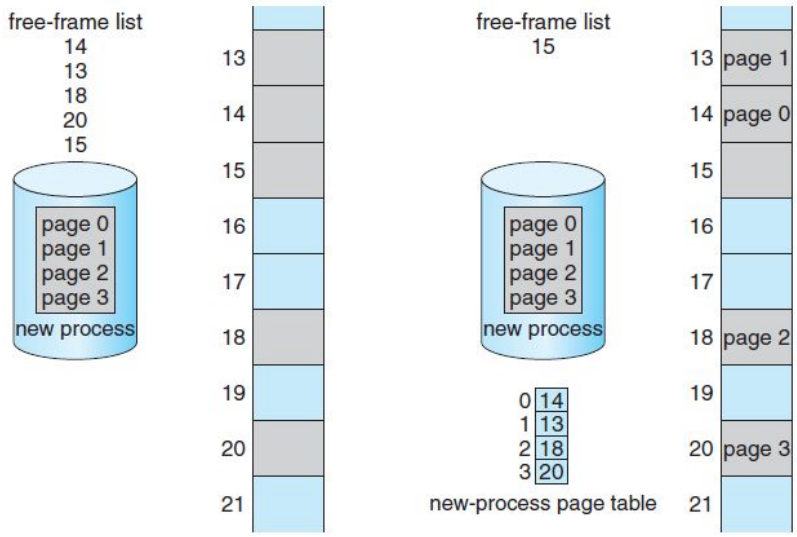
- Basic Method
  - Physical memory is broken into fixed sized blocks - frames
  - Logical memory is further partitioned into fixed sized block - pages
  - When a process is to be executed, pages are loaded from backing store (also fixed size) into available memory frames.
- Hardware Support
  - Every address has two parts: Page number (p - used to index in page table which contains base address of all pages in physical memory) and page offset (d - combined with base address to get physical memory address)



  - Page size is defined by hardware typically a power of 2
  - If size of logical address space is 2^m and page size is 2^n then m-n is number of the higher order bits designating page offset.

page number	page offset
p	*d*
m - n	*n*

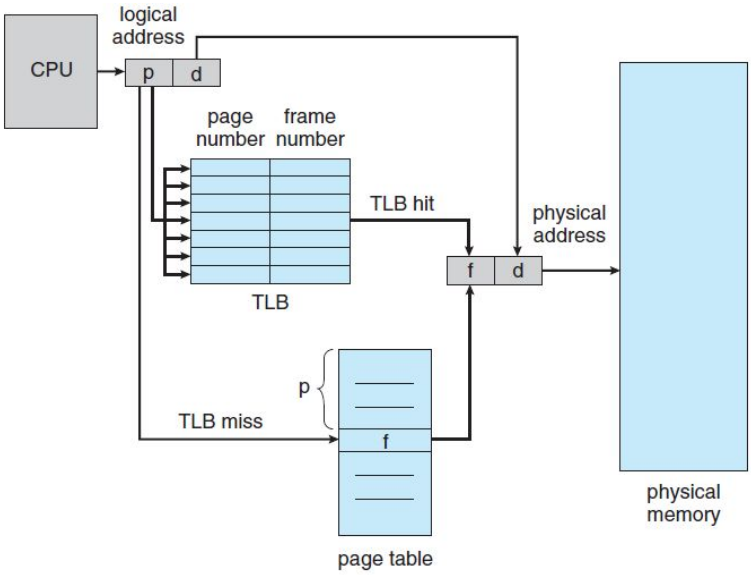
- P is index to page and d is displacement in page hence a form of dynamic relocation
- No external fragmentation but maybe internal fragmentation as last allocated frame may not be completely full. Average is half page per process (smaller page sizes are better but overhead of page table increases as well)



- Frame table is maintained by OS - most OS give a page table to each process and a pointer to that table is stored in PCB.
- Paging increases context-switching time.

Hardware Implementation

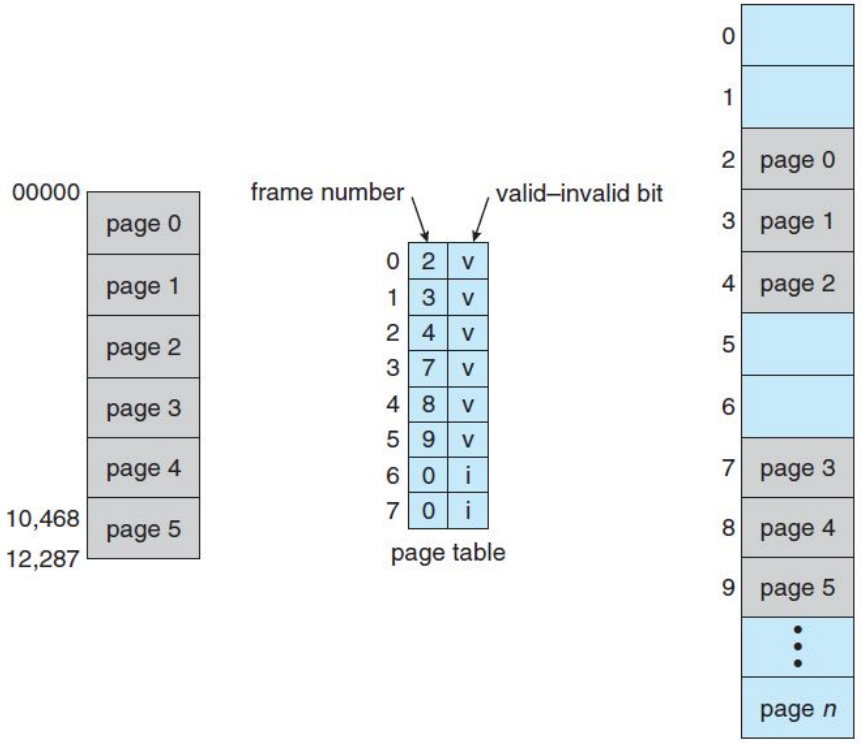
- Simplest case:
    - Page table is implemented as a set of dedicated registers built with high speed logic circuits to achieve fast address translation.
    - Registers are reloaded by dispatcher when context switching
    - Feasible only if small page table
- Modern:
    - Page table is kept in main memory
    - Page table base register points to page table of the process
    - Changing of page table requires only changing PTBR.
    - Two memory accesses are needed to reach memory hence use a fast lookup hardware cache - translation look-aside buffer (TLB)
    - TLB works like cache - fast but expensive has between 64 and 1024 entries



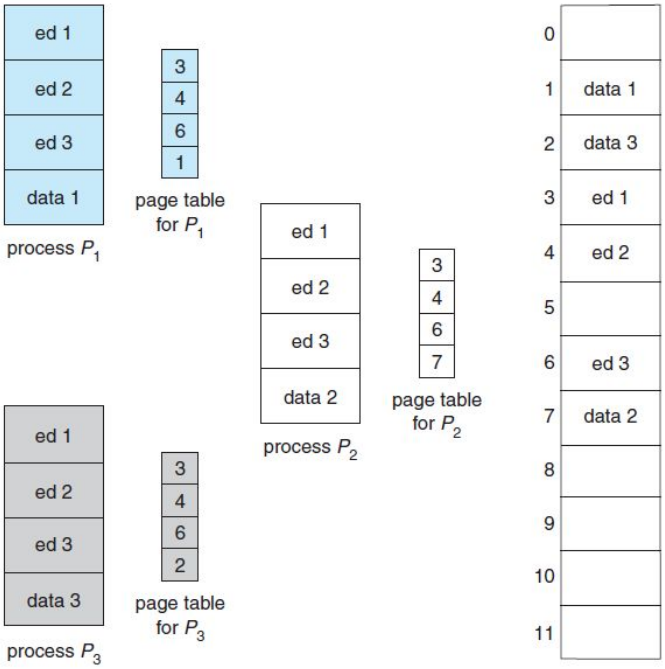
- When TLB is full use replacement policy such as LRU. Some TLBs allow entries to be wired down. Some entries can not be removed such as Kernel Code.

Protection

- Protection bits for each page frame is stored in the page table
- Valid invalid show whether page is in address space of particular process

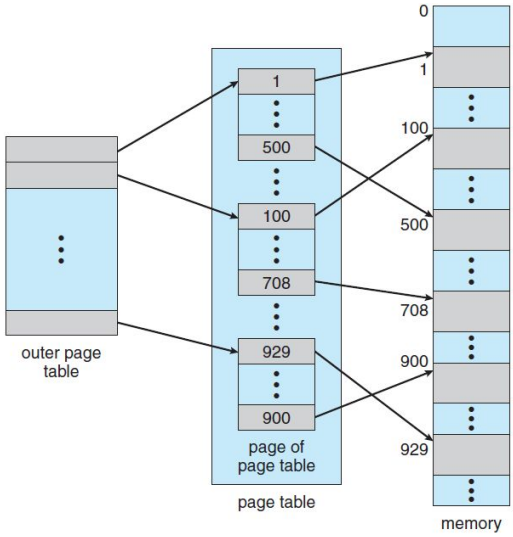


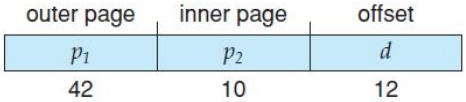
- Shared pages allow sharing of common code. Reentrant code is usually used (non-self-modifying code that does not change during execution)



Structure of Page Table

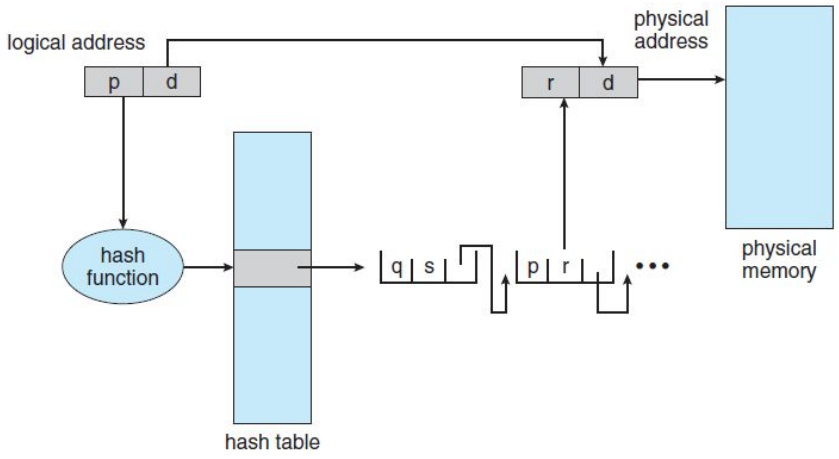
- Hierarchical Paging
  - A large logical address space results in large page table
  - Divide into smaller pieces and implement using two-level paging



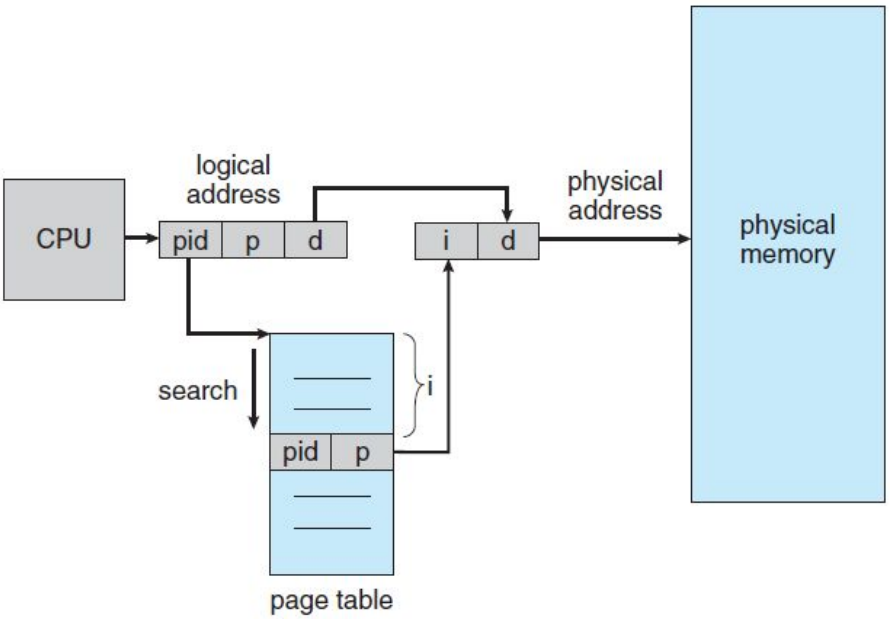


  - Can be repeated n times but causes large number of memory accesses.

- Hashed page table
    - Can be used for spaces larger than 32 bits



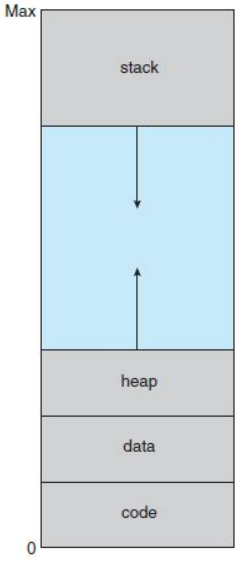
- Clustered page tables can be suitable for 64 bit address spaces
- Each entry in hash would refer to several pages in clustered page table
- Inverted Page Table
    - Contains one entry for each frame of memory and has id of process that owns the page
    - Only one page table exists in system
    - Increases amount of time to search



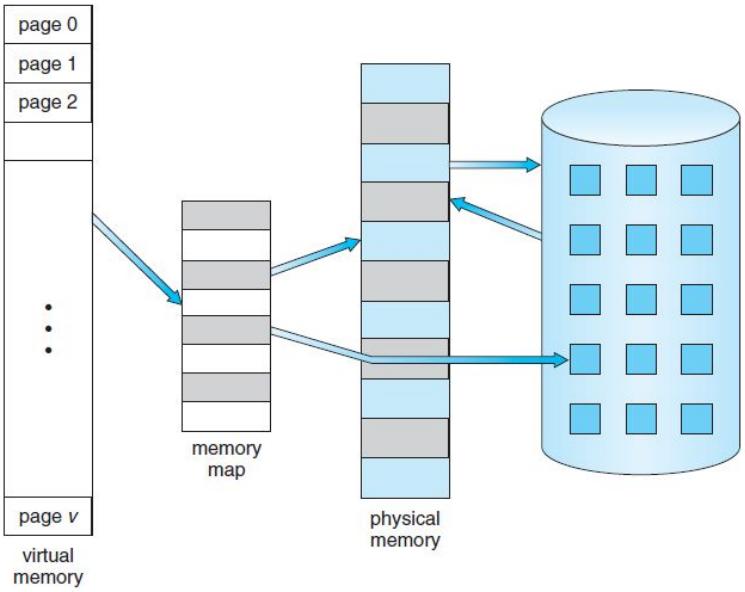
Virtual Memory

Introduction

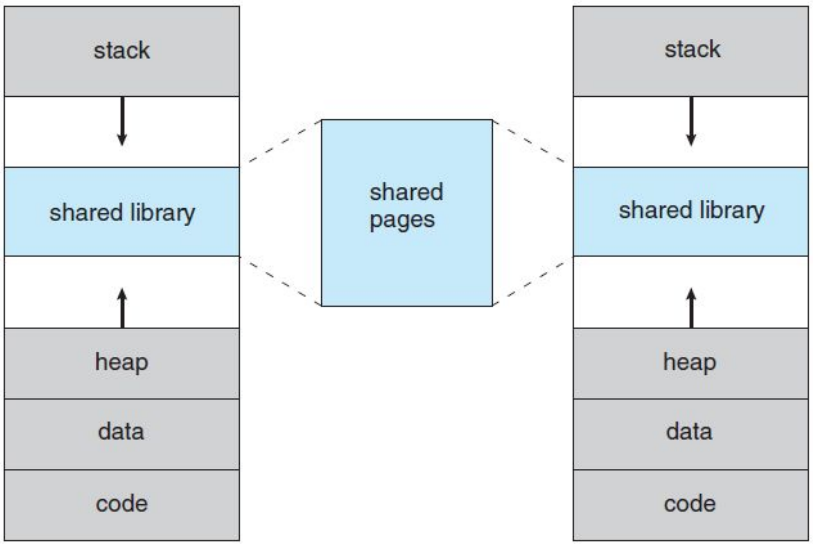
- Multiprogramming requires entire process is in memory before execution
- Virtual memory allows execution without process being completely in memory. Hence bigger programs but decreases performance.



- But by removing unused parts, less I/O is needed to load or swap user programs

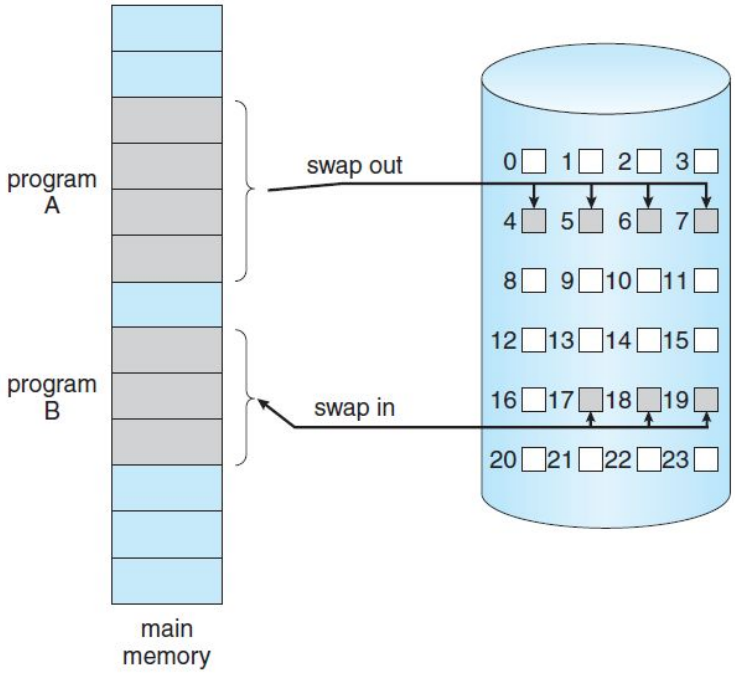


- Page sharing is implemented more easily and inter process communication is implemented.

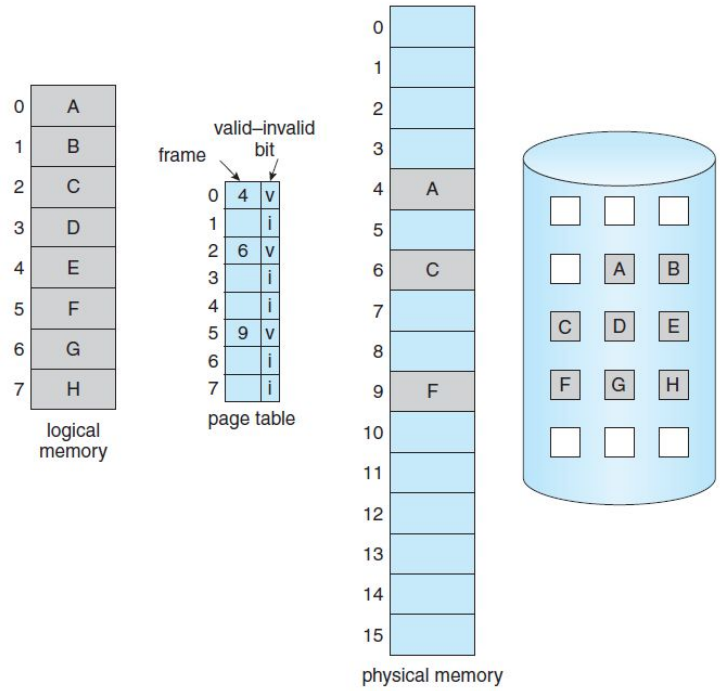


Demand Paging

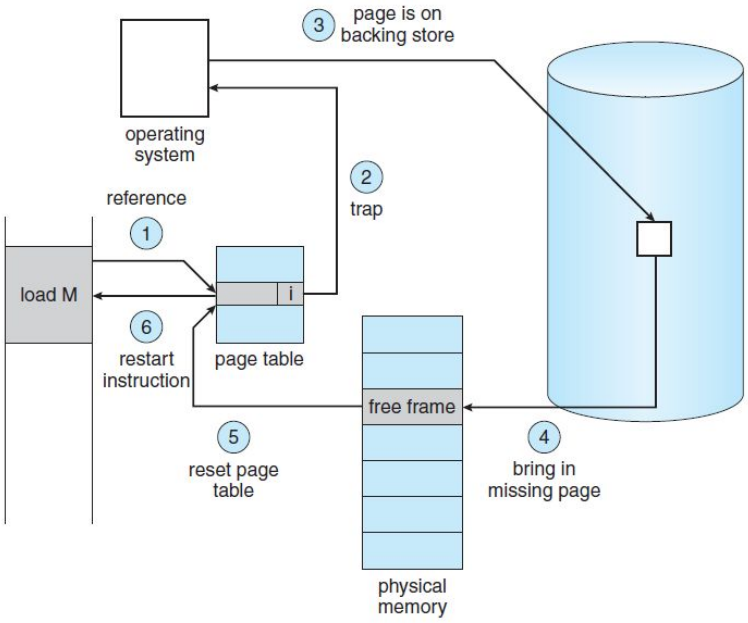
- Load pages of process as necessary
- Page is swapped into memory by lazy swapper



- If access is limited to memory-resident pages execution is normally, when process tries to access a page not brought into memory it causes a page-fault trap.



- The following steps are followed when a page is referenced:



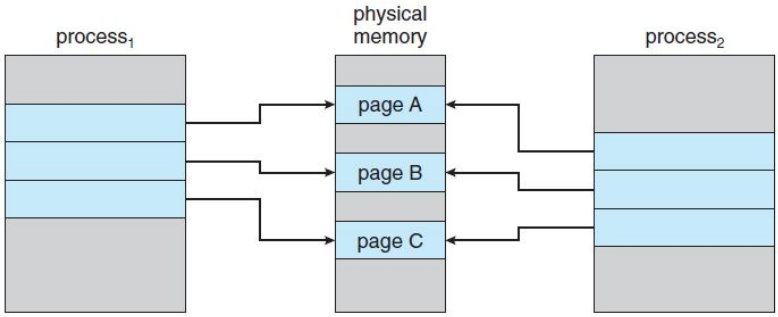
- Pure Demand Paging
    - Process starts with no pages
    - Keeps faulting to load required pages
    - When a page fault occurs, the instruction is to be restarted (fetch decode and execute)
    - Problem occurs if page fault occurs while moving something (half of it has moved) hence it should first attempt to access both ends of the block before

anything is modified. Or store overwritten values in temporary registers. If a fault occurs write them back.

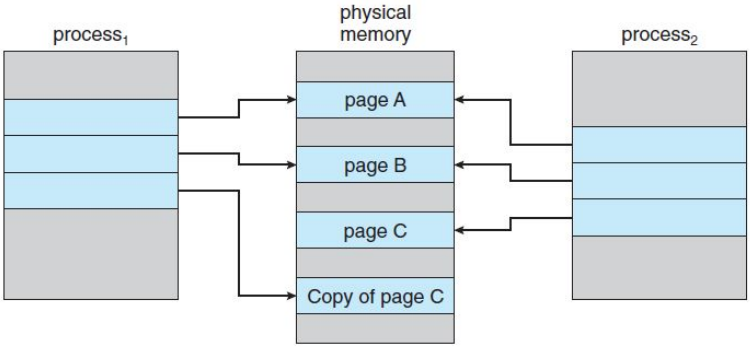
- Page Fault Service sequence of steps:
    - Trap to OS
    - Save Process state and user registers
    - Determine interrupt was a page fault
    - Check if page reference is legal and determine location on disk
    - Issue a read from disk to free frame
    - While waiting for response, allocate CPU to some other user
    - Receive interrupt from disk
    - Save registers for the other user
    - Determine interrupt was from disk
    - Correct page table to show desired page in memory
    - Wait for CPU to be allocated to this process again
    - Restore save state and resume interrupted instruction
- Page fault rate should be kept low in a demand paging system

Copy On Write

- Traditionally, when a page made a child process, it would be given a copied space in the memory but most child processes were only created to use exec()
- Therefore copy on write is used, child and parent share the same page



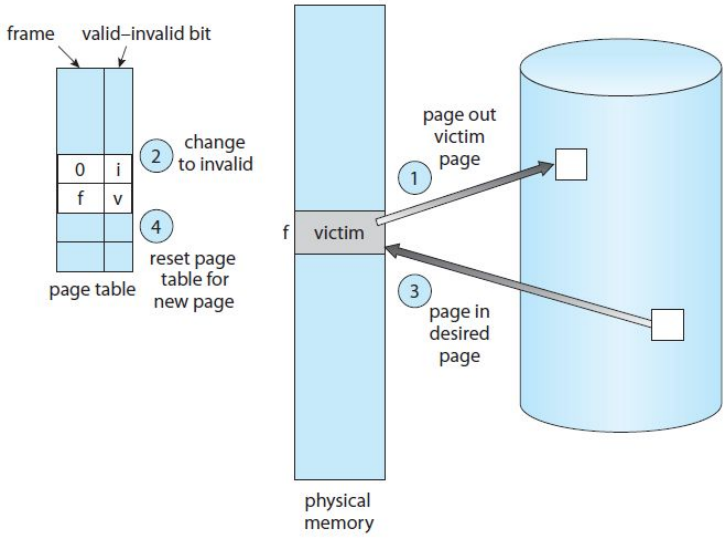
- But when the child tries to write in the memory, a copy is created.



- Many OS provide a pool of free pages for these immediate allocations (zero-fill-on-demand page)

Page Replacement

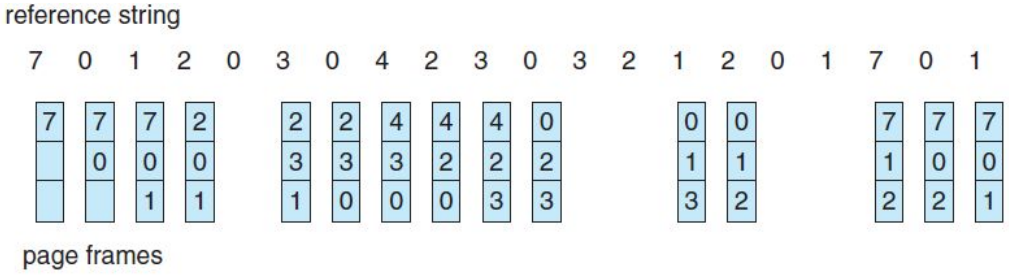
- What happens if page fault occurs and there are no free frames available.
- Basic Page replacement
    - If no frame is free, select a frame not currently in use
    - Write contents to swap space and change contents of page table.



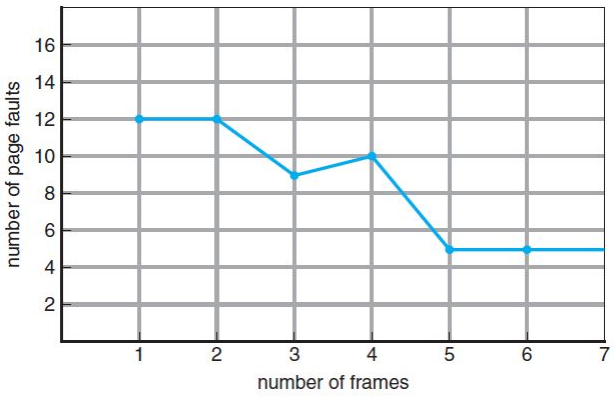
    - Modify or dirty bit can be used to improve page-fault service time

Types of Page Replacement

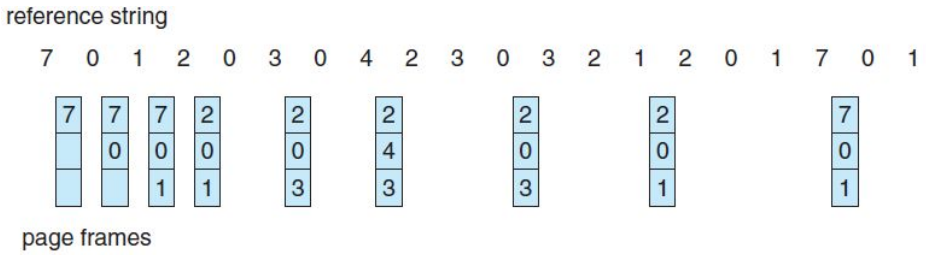
- FIFO



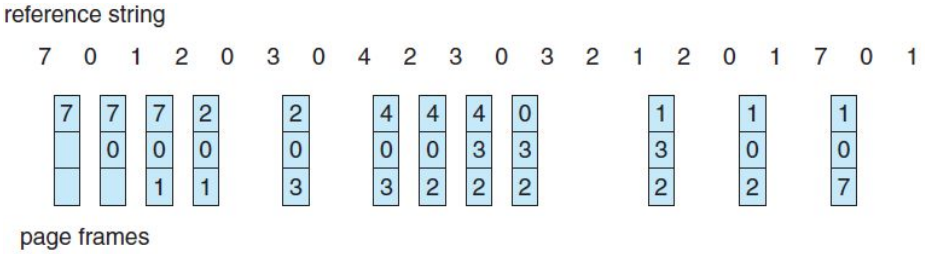
    - Remove oldest page
    - Performance is not always good
    - Heavily used page may be replaced hence suffers from Belady's anomaly



- Optimal Page Replacement



  - Lowest page-fault rate of all
  - Replace page that WILL not be used for longest period of time
  - Requires future knowledge
- LRU



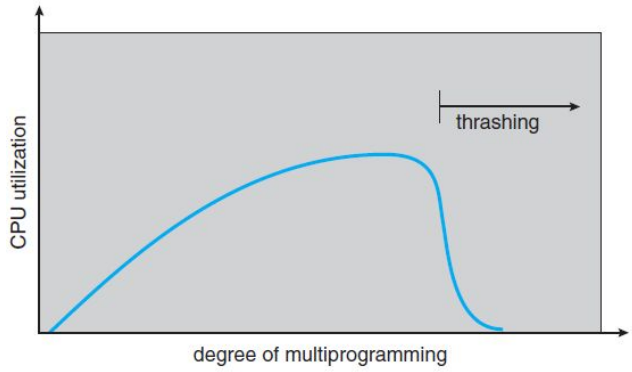
  - Least Recently Used - looks backward in time

Global vs Local Allocation

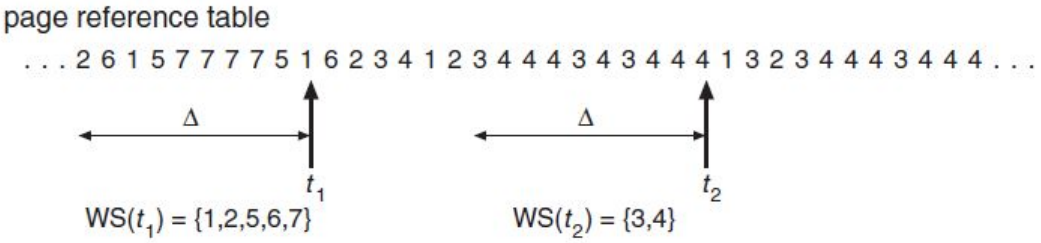
- Global frames means any process can take a frame from another process; the pool is global.
- Local replacement means separate local pools are allocated for separate processes.
- An example would be allowing higher priority processes to take frames from its own pool plus from lower priority processes.
- Hence global is preferred.

Thrashing

- When number of frames is too limited, page faults occur more frequently.
- If more time paging than execution then process is thrashing (high paging activity)
- Hence CPU utilization decreases.
- Main cause is global replacement algorithm:
    - When CPU utilization is less, DOM is increased by executing more processes at once
    - These processes may take away frames from lower priority processes
    - This causes thrashing in the lower processes which in turn decreases the CPU utilization once again (only context switching is occuring) - the cycle continues
- Hence we need local back - when a process starts thrashing it can't steal from other processes causing thrashing in the other one as well.

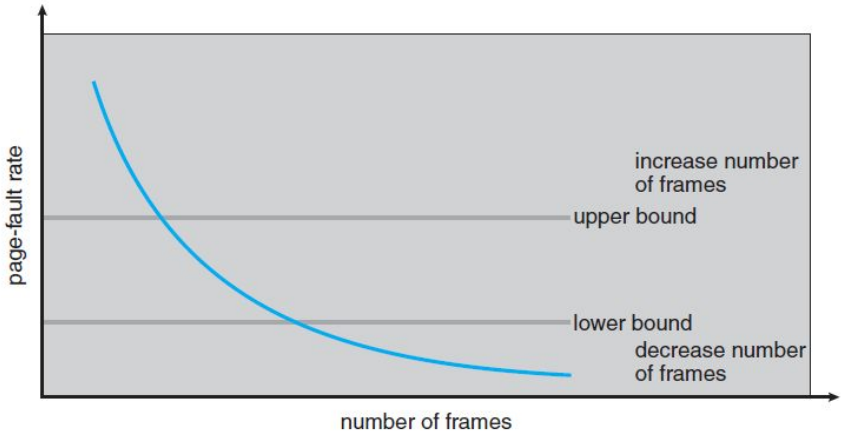


- Prevent thrashing altogether by providing the process with as many frames as necessary
- To find the exact number, we use 'working set strategy'
    - It is based on the assumption of locality that as a process executes, it moves from locality to locality - local variables for each function etc
    - It uses parameter Δ to define the working set window
    - The set of pages in the most recent Δ page reference are part of the working set



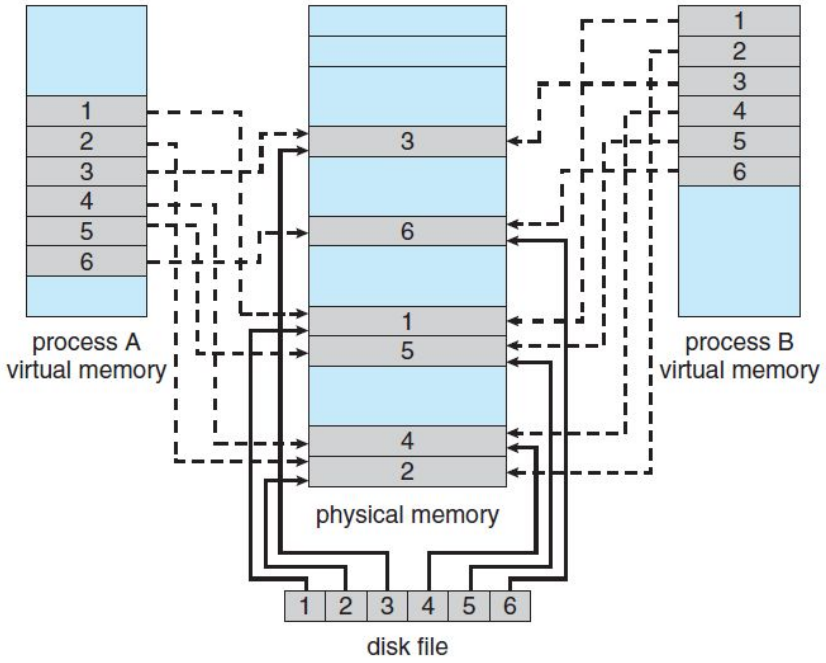
    - Accuracy depends on size of Δ, and it defines D = ∑ WSSi
    - Where D is total demand for frames for all processes, if greater than total number of frames (D>m), thrashing will occur. Hence some processes are suspended.
- Page Fault Frequency is also used to control thrashing
    - If high, processes need more frames; if low then too many frames.

- Process is suspended through selection if page fault rate increases beyond our defined upper boundary



Memory Mapped Files

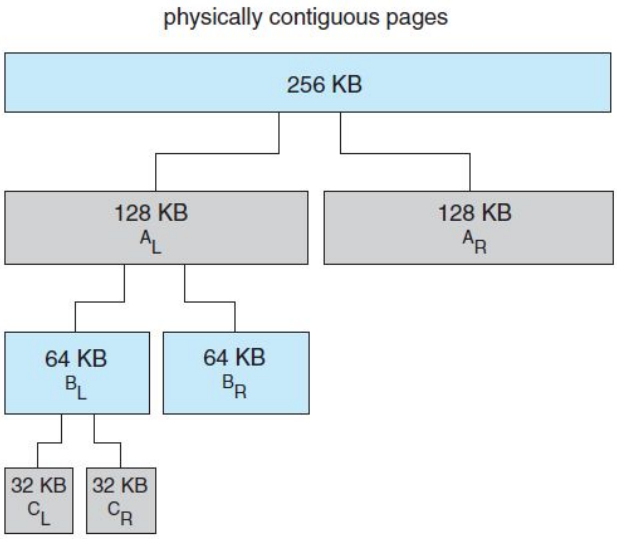
- File I/O can be done using fopen and all that but it can also be done using pointer (use mmap function - similar to malloc) and mapping the contents of the file to the virtual memory.
- Any changes made to the pointer will result in changing the contents in the virtual memory/primary memory (as process keeps shifting between the two).
- Hence changes will not be directly made but will occur after file is closed.
- This approach will help reduce direct secondary memory accesses and increase efficiency of the program.
- Some OS update contents of disk periodically.
- Multiple processes may be allowed to map the same file concurrently to allow sharing of data - hence any changes are updated for all processes.
- This sharing is implemented through page tables.



- Similarly memory mapped I/O devices can also be used.
- Usually done using polling; where the CPU keeps looping to check whether device is ready (control bit is cleared by device). Also called programmed I/O (PIO)
- Interrupt driven I/O might also be used.

Allocating Kernel Memory

- Free memory for kernel is always contiguous and is not subject to paging.
- Because kernel must use memory conservatively minimizing wastage due to fragmentation.
- Also, hardware devices may require contiguous physical memory.
- Two strategies used for memory allocation to kernel process:
    - Buddy System
        - Always allocate fixed sized segments of physically contiguous pages - keep dividing to min required (power of 2 allocator)
        - The advantage is adjacent buddies can be combined to form larger segments reducing external fragmentation.
        - However this will obviously cause internal fragmentation within allocated segments.
        - Smaller min block sizes will solve this but will cause larger data tracking



- Slab Allocation
    - Made up of one or more physically contiguous pages
    - Cache consists of one or more slabs
    - Single slab for each unique kernel data structure

Process Synchronization

Critical Section Problem

Producer consumer problem is a good example:

- Both producer and consumer are correct separately but may give incorrect results if executed concurrently
- When two functions are allowed to manipulate shared variables concurrently, incorrect results may occur which is known as race condition

For set of processes:

- Each process has a segment of code called critical section
- Both processes are changing a common variable or writing to a shared file
- Hence we design a protocol so that the processes can cooperate by requesting permission to enter the critical section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Any solution must fulfill the following criterions:

- Mutual Exclusion: If P1 is executing CS, no other process can access CS.
- Progress: Select one of the waiting processes if there are no processes executing CS.
- Bounded Waiting: There should be a bound amount of processes allowed to enter their CSs after a process has made a request to enter its CS (limited waiting line)

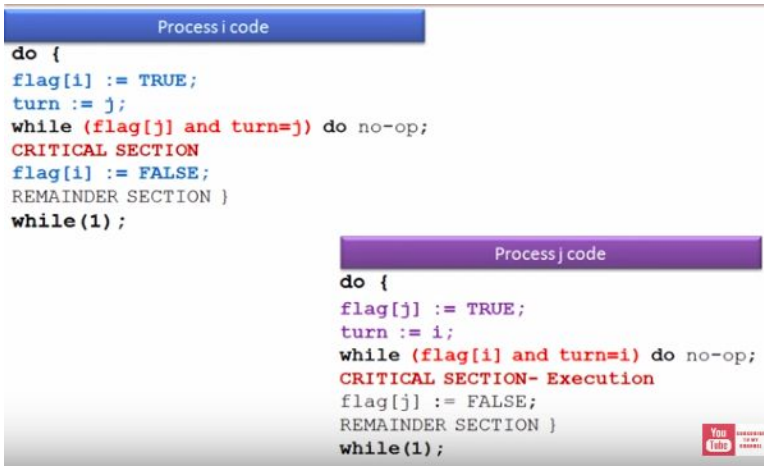
Kernel code may also suffer from race conditions hence OS solves using 2 approaches:

- Preemptive Kernel
    - Allows process to be preempted while it is running in Kernel mode
    - May suffer from race conditions hence needs to be designed carefully (SMP)
    - More suitable for real time programming
- Non-preemptive kernel
    - Does not allow preemption while in Kernel mode
    - Free from race condition

Preemptive kernels are more difficult to design hence Peterson's solution (software based sol):

- Implemented on two processes p[2]
- Two variables: a bool flag[2] which will tell the intention of entering the CS for both processes and the other is int turn which will indicate whose turn it is to enter

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



- The process with the intention always gives the other process a chance and waits if the flag was already set by the other process first.
- If the other process sets the turn to i, process[i] will exit the loop and start executing critical section
- Exit condition for p[i] is that it will set the flag[i] to false which will make process[j] exit its loop and execute its critical section.

It satisfies the 3 conditions:

- Mutual exclusion since both processes will only execute when flag[i]=flag[j]=true but only one will execute the critical section depending on turn (assuming each instruction is atomic and thread will keep switching)
- Progress since after one is done the other one will execute
- Bounded waiting since only one process to wait for

Peterson's solution is software based and can not work on current architectures with multiple instructions running in a thread before switching.

Hardware Synchronization

One solution is to disable interrupts while a shared variable is being modified (done by non preemptive kernels) but does not work in a multiprocessor environment.
Any solution to the problem requires a lock:

- It can be set by one process to ensure no other process enters the CS.
- However this has its own problem that two or more processes could enter the lock function at a time. Hence we need an atomic operation on the hardware side to ensure it is done by only one process at a time.
- One example is the test and set instruction which simultaneously sets the boolean lock and returns its previous value:

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 5.3 The definition of the `TestAndSet()` instruction.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with `TestAndSet()`.

- Another is compare and swap which works with 3 operands and is more practical to switch the booleans if the target meets the expected value

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

- These examples however do not guarantee the order in which the processes will gain access to their CS hence bounded waiting

- Another solution is to use test and set with two shared variables, bool lock and bool waiting[n] where n is number of processes.

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

- The value of key will only change when the lock is released
- Before exiting, the process makes sure to set the waiting to false for the next process in the array

Programmers can not always access these low level instructions hence other apis are implemented in languages such as:

- Mutex locks: Simplest synchronization tools
    - They can be used as:

```
do {

   acquire lock

      critical section

   release lock

      remainder section

} while (TRUE);
```

    - The implementation is quite simple:

Acquire:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

Release:

```
release() {
   available = true;
}
```

- These types of locks are referred to as spinlock since the processes spin (wait), wasting CPU cycles while waiting for the lock to become available.
- However no context switch is required hence they are useful for short amount of times.

- Semaphores: A more robust alternatives
  - Is an integer which can be accessed using two atomic operations wait (increment) and signal (decrement)

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

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

  - Semaphores can be used as mutexes taking on a value of 0 or 1 in systems which do not have a separate mutex mechanism (binary semaphores)

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

- We can overcome the problem of busy waiting by implementing the definition of signal and wait in the following way:

Semaphore Structure:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Wait Operation:

```
wait(semaphore *S) {
          S->value--;
          if (S->value < 0) {
                add this process to S->list;
                block();
          }
}
```

Signal Operation:

```
signal(semaphore *S) {
       S->value++;
       if (S->value <= 0) {
             remove a process P from S->list;
             wakeup(P);
       }
}
```

- The magnitude of the number of processes in waiting is shown by how negative the value of S is.
- A major problem that can arise however is a deadlock when two processes are waiting for the same resource that they each have:

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

    - Another problem is starvation. In a FIFO queue, there will be no problems but in LIFO, the first process could get blocked indefinitely.
- The biggest problem that might occur is priority inversion
    - A higher priority process will have to wait for a lower priority process to finish.
    - One solution is to have only two priorities in the OS but it is insufficient in today's systems
    - Hence the solution is to have priority-inheritance
    - All lower priority processes accessing resources required by higher priority process become higher priority processes themselves until they are finished with the resource

Classical Problems of Synchronization

These problems are used to test any new proposed synchronization scheme.

- Bounded Buffer Problem
    - Producer and consumer processes are different
    - Only one must have access to the array buffer at a time
    - We use two semaphores empty and full to keep track of the buffer

```
do {
      . . .
   // produce an item in nextp
      . . .
   wait(empty);
   wait(mutex);
      . . .
   // add nextp to buffer
      . . .
   signal(mutex);
   signal(full);
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
}while (TRUE);
```

Figure 5.10 The structure of the consumer process.

- Readers-writers problem
    - Processes which only read are readers and which can also write are writers
    - A writer and another process should not access the data simultaneously
    - There are two variations:
        i. No reader should be kept waiting unless a writer has permission to use the shared object
        ii. Once a writer is in the queue, it should perform its write asap. It is given the highest priority so no reader can read, if a writer is waiting
    - Solution to any of these problems may result in starvation as writers may starve in the 1st case and readers in the 2nd.
    - This solution assumes multiple readers:

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

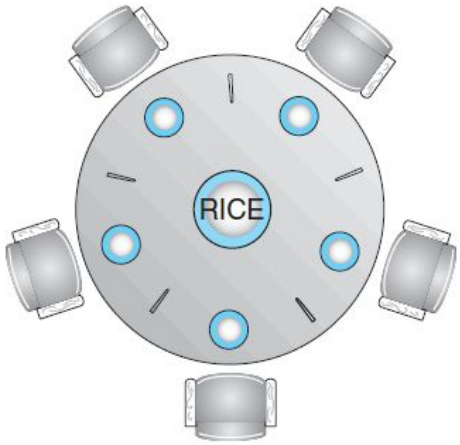
Figure 5.11 The structure of a writer process.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

- Read_count keeps track of the number of readers
- The lock is only released once all of the readers are finished reading (read_count == 0)
- The solution is useful in cases where we can identify reading and writing processes and readers > writers

- Dining-Philosophers Problem
    - One chopstick on left and right and two are required by both:



- They wait in the following manner:

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

- The problem is if all of them take one chopstick on the left then there will be no chopsticks left on the right, causing all of them to go in a deadlock
- One solution is to allow at most 4 philosophers to be sitting at the table at once (limited processes)
- Another is to allow them to pick one only if both are available (all or nothing - pickup chopsticks in critical section)
- For odd number of philosophers, the even numbered pick up their right first then their left and the order is reversed for odd numbered philosophers.
- While these solutions may prevent a deadlock, they may end up in starvation.

Deadlocks

A deadlock occurs when a process requests a resource and goes indefinitely into a wait state.

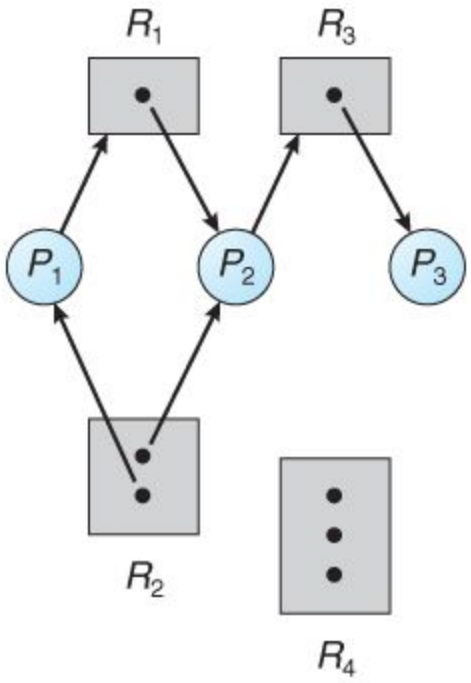
Characterization

A deadlock can be identified if all four conditions are met:

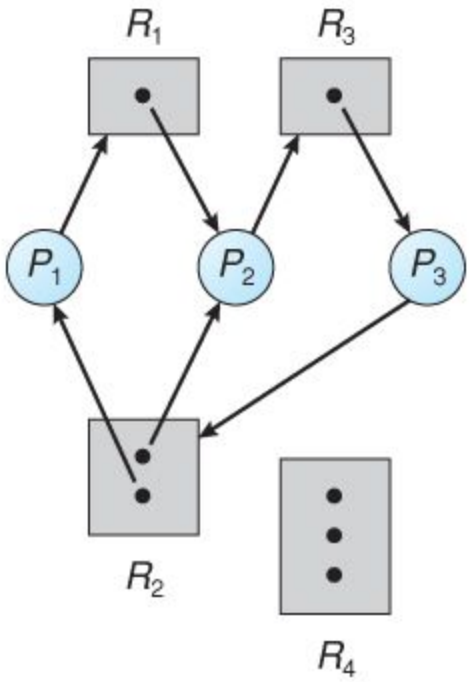
1. Mutual Exclusion: At least one resource must be held in a non-shareable mode. If a process requests for it, it must wait before it is released
2. Hold and wait: A process is holding a resource and is also waiting for at least one other resource to be released by another process
3. No preemption: The process cannot be preempted to release the resource
4. Circular wait: A set of processes {p1, p2, …, pN} exists such that p[i] is waiting for p[(i+1)%(n+1)]

Resource Allocation Graph

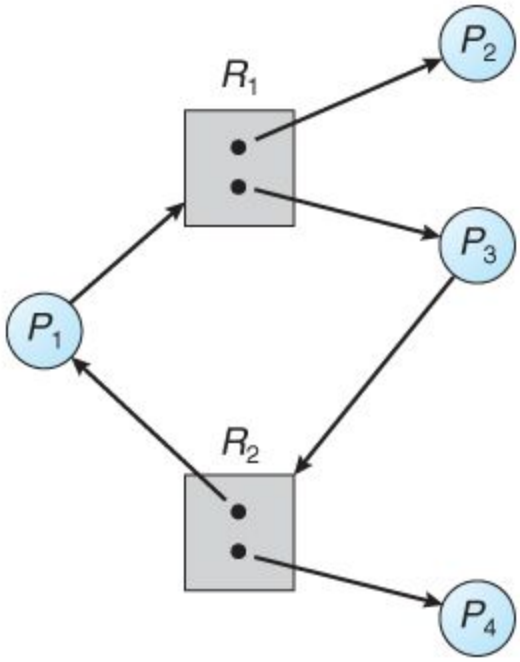
Deadlocks are better detected using graphs:



If the graph contains no cycles then there are no deadlocks.
If the graph contains a cycle AND there is only one instance of the resource available then a deadlock exists.
If the graph contains a cycle and there are multiple instances of the resource then a possibility of a deadlock exists:



Cycle exists AND deadlock exists



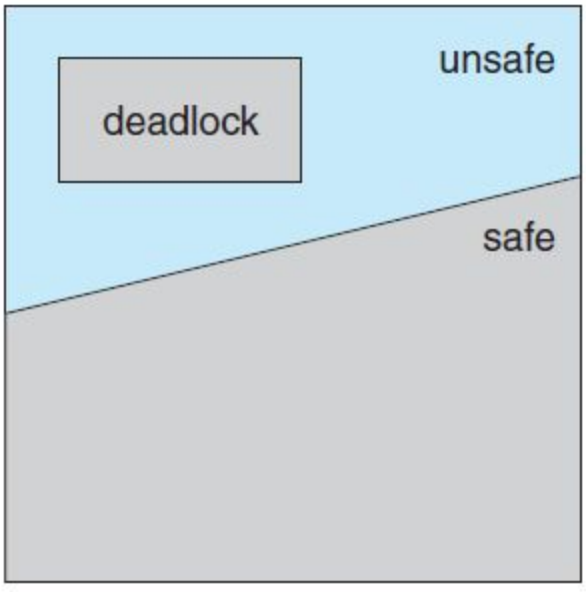
Cycle exists BUT no deadlock

Handling Deadlocks

There are 3 main methods used by OS today:

1. Avoid or prevent deadlock from occurring

a. This requires processes to give information in advance about the resources they need
b. It is necessary to ensure that at least one of the 4 conditions mentioned above are not true
    i. Mutual exclusion can not always be denied as resources such as printer etc can not be shared
    ii. Hold and wait can be denied by assuring if a process requests a resource it should not be holding any other resources - either allow it to request only when it has none or place all system calls for resources in the beginning of the program. Starvation is definitely an issue
    iii. No preemption can be denied for the resources of a process if it requests more resources that can not be immediately granted - which will force the process to re-request any resource preempted while it was waiting
    iv. Circular waiting can be avoided by numbering all resources and require that processes can only request resources in strictly increasing order. If a process tries to request a resource[i] where i < j, it will have to release all the resources less than j and greater than i.
c. Safe State
    i. System runs deadlock avoidance algorithm on set of processes to ensure that deadlock will not occur - it will always allocate processes in a safe sequence
    ii. An unsafe state may lead to a deadlock



- Safe sequence is determined by granting the resources to the least demanding process: https://youtu.be/c1E20T60PmU
- If no safe sequence exists then the system is unsafe
- Resource allocation graph can also be used if there is only a single instance of each resource.

- Banker's algorithm is used for systems with multiple instances of a resource: https://youtu.be/7gMLNiEz3nw

2. Recover from a deadlock state
    a. Can be done by using banker's algorithm as deadlock detection algorithm
    b. Called frequently if deadlock occurs frequently - one extreme is to invoke every time a resource can not be granted
    c. So invoke only when CPU utilization drops below a certain point
    d. When a deadlock is detected OS can:
        i. Abort one or more of the processes to break the circular wait - either all or one at a time until deadlock breaks
        ii. Preempt some resources from deadlocked processes - the issues faced are selecting a victim, rolling back processes to a safe state and starvation. Starvation can be solved by using a priority system and increasing priority of process whose resources keep getting preempted.
3. Ignore the problem

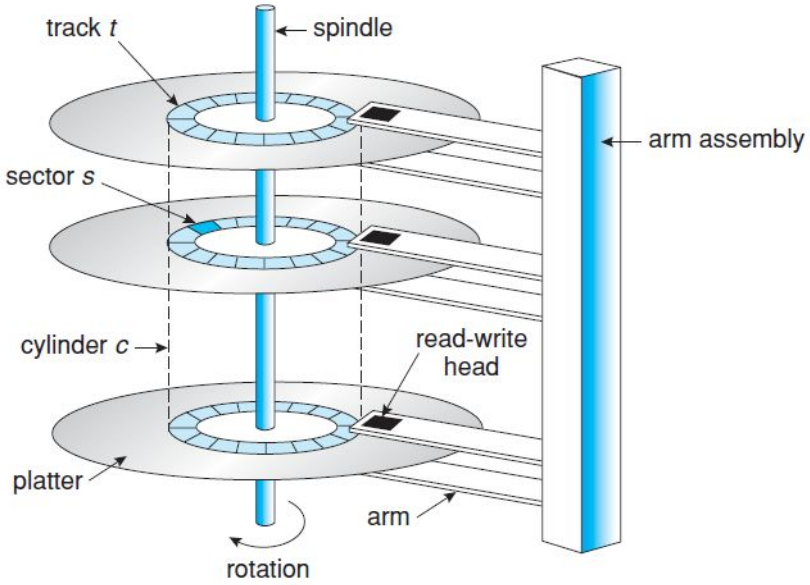
All three approaches are combined and used where necessary.

Storage Management

Disk Storage

Disk drives are structured in the following way:

- Large one dimensional array of logical disk blocks
- A typical magnetic has the following structure: https://youtu.be/Ep-yM894mQQ



- Request for disk I/O is generated by file system and virtual memory system

- Disk scheduling algorithms are used to improve bandwidth, average response time and variance in response time
- Disks can suffer from external fragmentation

Disk Scheduling

Some terminology:

- Seek time is time for disk arm to move the heads to the cylinder containing the desired sector
- Rotational latency is additional time for disk to rotate desired sector to disk head
- Bandwidth is bytes transferred / time

Both access time and bandwidth can be improved based on the algorithm we use to order which I/O service is requested:

- FCFS
    - First come first serve
    - Fair but not the fastest
- SSTF
    - Shortest seek time first
    - Service requests closest to head
    - Not optimal and starvation may occur
- SCAN
    - Disk arm starts at one end and moves towards the end servicing requests as it reaches each cylinder
    - Direction of head movement reverses after reaching 0 or MAX
    - Also called elevator algorithm
- C-SCAN
    - Same as SCAN
    - Head returns to 0 after reaching end
- LOOK
    - Same as SCAN
    - Disk arm goes as far as the final request in each direction
    - Another variant is C-LOOK

Choice of algorithm depends on:

- File allocation method
- Location of directories
- Behavior of application for generating disk requests

A RAID is redundant array of inexpensive disk, and was designed to use cheap small disks in place of larger and more expensive ones.

Protection & Security

The two terms are closely related:

- Protection means to protect system by limiting access for users. Only authorized processes can operate on the resource.
- Security ensures authentication of system users to protect information stored in the system.
- Protection is an internal problem to protect processes against each other and prevent violation of access restriction by users whereas security problem to prevent unauthorized access or alteration of data

Protection

The following principles are used for protection:

- Principle of least privilege
    - Used to give users, programs and systems just enough privileges to perform their task
    - System calls can be used to get indirect access to privileged resources
    - This helps create a secure computing environment and limits any damage
- Access matrix
    - A general model of protection
    - Rows represent domains which is a set of access rights and columns represent objects
    - Entries represent what access level the domain has to that resource

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

    - OS ensures that the matrix is only manipulated by authorized agents
    - Processes should be able to switch from one domain to another
    - Language based protection is implemented to make the protection scheme available to the app developer - which would result in a compiler based enforcement

Security

The following terms are used:

- Cracker: Intruder attempting to breach security

- Threat: Potential security violation
- Attack: Attempt to break security

The following types of security breaches exist - could be intentional or accidental:

- Breach of confidentiality: Unauthorized reading of data or theft of information
- Breach of integrity: Unauthorized modification of data
- Breach of availability: Unauthorized destruction of data
- Theft of service: Unauthorized use of resources
- Denial of service: Prevent legitimate use of system

Common methods to breach security are:

- Masquerading: Participant in communication pretends to be someone else - breach of authentication
- Replay attack: Malicious or fraudulent repeat of a valid data transmission
- Man in the middle attack: Attacker sits in data flow of communication - might masquerade as sender to receiver
- Phishing: Legitimate looking email or webpage misleads a user to enter confidential data
- Dumpster diving: Attempting to gather info to gain unauthorized access to system

Security measures are taken at 4 levels:

1. Physical: Army bulalo
2. Human: Authorization is done carefully to only allow appropriate users to access the system
3. OS: System must protect itself from security breaches
4. Network: Data might be intercepted

A system must provide protection to allow implementation of security features