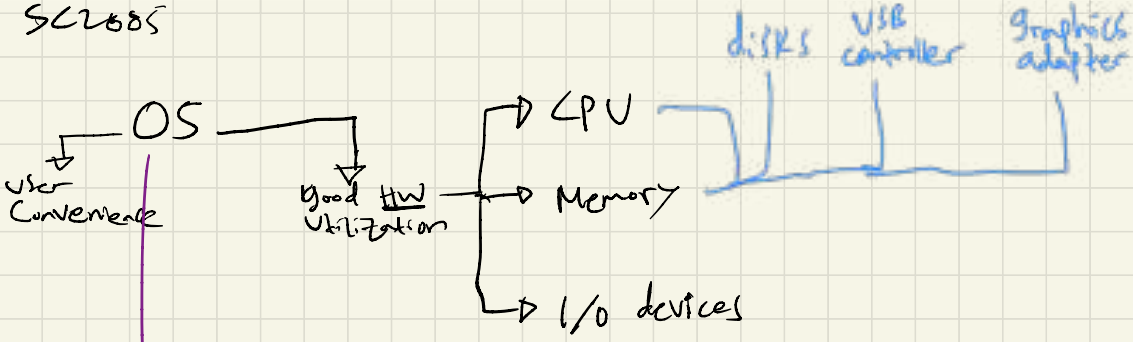
SC2005



Kernel/monitor/supervisor mode/superuser mode
↳ accept commands from users + HW

User mode

use mode bit [0 or 1]

Root/administrator
= user account in OS
↳ Jobs execute in user mode
↳ may execute code in kernel indirectly by loading kernel mod

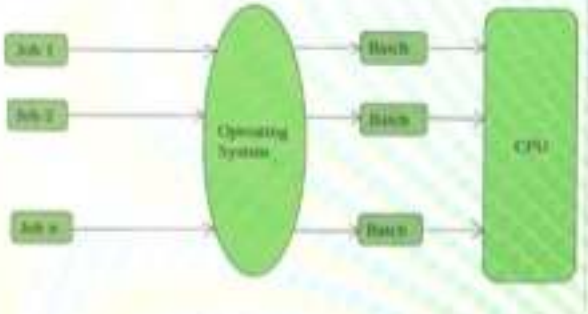
Types of computing system
↳ batch (doesn't interact with comp directly)
- reduce setup time by batching similar jobs
- Only 1 job in memory at any time
- when job waits for I/O, CPU idles
↳ Time-sharing (multiprogrammed)
- Several jobs kept in main mem
- highly interactive, supports multiple online users
- Job swap in n out of mem to hard disk
↳ real time
- Job must be completed within deadline

handheld (phone)
↳ limited memo, slow processors

multicore processor = 1 processor that executes 1 program faster
multiprocessor = 1 or more processor that simultaneously processing

1. Batch Operating System –

This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having the same requirement and group them into batches. It is the responsibility of the operator to sort jobs with similar needs.



Advantages of Batch Operating System:

- It is very difficult to guess or know the time required for any job to complete. Processors of the batch systems know how long the job would be when it is in queue
- Multiple users can share the batch systems
- The idle time for the batch system is very less
- It is easy to manage large work repeatedly in batch systems

Disadvantages of Batch Operating System:

- The computer operators should be well known with batch systems
- Batch systems are hard to debug
- It is sometimes costly
- The other jobs will have to wait for an unknown time if any job fails

Examples of Batch based Operating System: Payroll System, Bank Statements, etc.

2. Time-Sharing Operating Systems –

Each task is given some time to execute so that all the tasks work smoothly. Each user gets the time of CPU as they use a single system. These systems are also known as Multitasking Systems. The task can be from a single user or different users also. The time that each task gets to execute is called quantum. After this time interval is over OS switches over to the next task.

needs:
↳ memory management
↳ CPU scheduling
↳ I/O device scheduling

Advantages of Time-Sharing OS:

- Each task gets an equal opportunity
- Fewer chances of duplication of software
- CPU idle time can be reduced

Disadvantages of Time-Sharing OS:

- Reliability problem
- One must have to take care of the security and integrity of user programs and data
- Data communication problem

→ Windows, macOS, Unix, linux

Examples of Time-Sharing OSs are: Multics, Unix, etc., desktops, servers

4. Network Operating System –

These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions. These types of operating systems allow shared access of files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections, etc. and that's why these computers are popularly known as **tightly coupled systems**.

Advantages of Network Operating System:

- Highly stable centralized servers
- Security concerns are handled through servers
- New technologies and hardware up-gradations are easily integrated into the system
- Server access is possible remotely from different locations and types of systems

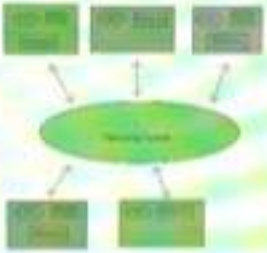
Disadvantages of Network Operating System:

- Servers are costly
- User has to depend on a central location for most operations
- Maintenance and updates are required regularly

Examples of Network Operating System are: Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD, etc.

3. Distributed Operating System -

These types of the operating system is a recent advancement in the world of computer technology and are being widely accepted all over the world and, that too, with a great pace. Various autonomous interconnected computers communicate with each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred to as **loosely coupled systems** or distributed systems. These systems processors differ in size and function. The major benefit of working with these types of the operating system is that it is always possible that one user can access the files or software which are not actually present on his system but some other system connected within this network i.e., remote access is enabled within the devices connected in that network.



Advantages of Distributed Operating System:

- Failure of one will not affect the other network communication, as all systems are independent from each other
- Electronic mail increases the data exchange speed
- Since resources are being shared, computation is highly fast and durable
- Load on host computer reduces
- These systems are easily scalable as many systems can be easily added to the network
- Delay in data processing reduces

Disadvantages of Distributed Operating System:

- Failure of the main network will stop the entire communication
- To establish distributed systems the language which is used are not well defined yet
- These types of systems are not readily available as they are very expensive. Not only that the underlying software is highly complex and not understood well yet

Examples of Distributed Operating System are- LOCUS, etc.

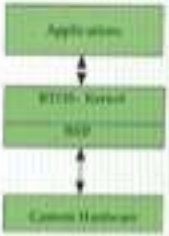
3. Real-Time Operating System – RTOS

These types of OSs serve real-time systems. The time interval required to process and respond to inputs is very small. This time interval is called **response time**.

Real-time systems are used when there are time requirements that are very strict like missile systems, air traffic control systems, robots, etc.

Two types of Real-Time Operating System which are as follows:

- **Hard Real-Time Systems:**
  These OSs are meant for applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or airbags which are required to be readily available in case of any accident. Virtual memory is rarely found in these systems.
- **Soft Real-Time Systems:**
  These OSs are for applications where for time-constraint is less strict.



Advantages of RTOS:

- **Maximum Consumption:** Maximum utilization of devices and system, thus more output from all the resources.
- **Task Shifting:** The time assigned for shifting tasks in these systems are very less. For example, in older systems, it takes about 10 microseconds in shifting one task to another, and in the latest systems, it takes 3 microseconds.
- **Focus on Application:** Focus on running applications and less importance to applications which are in the queue.
- **Real-time operating system in the embedded system:** Since the size of programs are small, RTOS can also be used in embedded systems like in transport and others.
- **Error Free:** These types of systems are error free.
- **Memory Allocation:** Memory allocation is best managed in these types of systems.

Disadvantages of RTOS:

- **Limited Tasks:** Very few tasks run at the same time and their concentration is very less on few applications to avoid errors.
- **Use heavy system resources:** Sometimes the system resources are not so good and they are expensive as well.
- **Complex Algorithms:** The algorithms are very complex and difficult for the designer to write on.
- **Device driver and interrupt signals:** It needs specific device drivers and interrupts signals to respond earliest to interrupts.
- **Thread Priority:** It is not good to set thread priority as these systems are very less prone to switching tasks.

Examples of Real-Time Operating Systems are: Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control system

- interrupt transfer control to Interrupt Service Routine through interrupt vector table
  - ↳ contains addresses of all service routines
- incoming interrupts are disabled while another interrupt is processed to prevent loss of interrupts
- Trap → software error or request (unhandled exceptions in user program)
- Interrupt handling
  - ↳ preserves CPU state by storing reg n program counter (context switch)
  - ↳ determine type of interrupt, then appropriate interrupt service routine
- DMA → used for high-speed I/O devices (transmit info close to memo speed)
  - ↳ 1 interrupt per block → by OS
    - ↳ to move block from buffer to mm by device controller until byte count = 0
  - : granted device access priority

System call = provide interface between user prog n operating system
- ↳ assembly language : or in C/C++
- ↳ require kernel mode

↓ fork ↓ exit

I/O → privileged instruction → OS
- ↳ to ensure correctness n legality

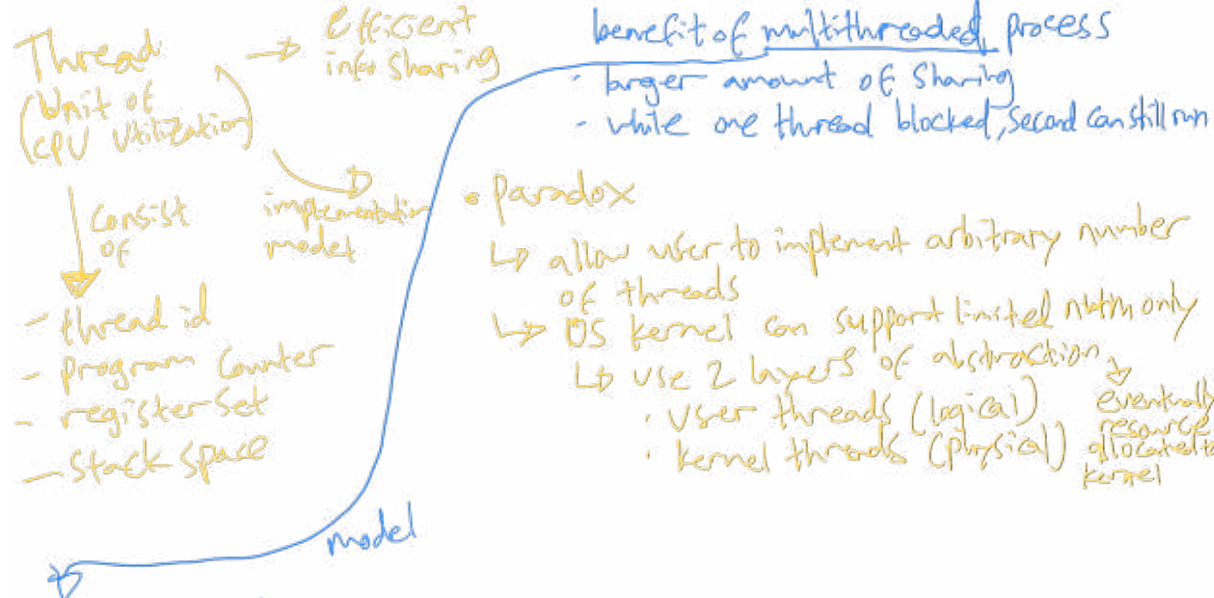
CPU generates trap if I/O operations try to bypass OS

- all queues are stored in MM (kernel memory space)

Process (program in execution)
- ↳ State
  - New
  - Ready
  - Running
  - waiting
  - terminating

→ independent process = not affected by other
→ cooperating process = mid communicate with others by
- ↳ message passing
  - less memo
  - program control issues
- ↳ Shared memo
  - more memo
  - exclusive access

Scheduler
- long term → Job — invoke infrequently
  - select processes from disk n load to MM for execution
  - the rest store in the memo
- short term → CPU — invoke frequent
  - select among processes that are ready to execute, n allocates the CPU to one of them
- Medium-term
  - when heavy load
    - partially executed process swap from memo → harddisk
  - when light load
    - swap back into mm



- many to 1
  dis :- if one block, all block

- 1 to 1
  dis :- Create too many kernel threads
  - Create 1 user thread require 1 kernel thread
  exp : window 95/98/NT/2000, OS/2, linux

- many to many
  dis : - not easy to decide the mapping
  exp : Solaris 2, windows NT/2000

(3)

- to keep CPU busy → I/O or event wait or terminate

Type of CPU Scheduler

↳ Non preemptive = CPU has been allocated to process till release by CPU w/ terminated (5) or I/O, event wait (1) (new admitted job (4), when CPU core idle)

↳ preemptive = CPU can be taken away from running process at any time by scheduler or happens (2), (3), (4) any time

- System-wide objectives (Criteria)

↳ Max. CPU Utilization = % time CPU cores are busy = $\frac{\Sigma \text{execution time on each core}}{\text{total time} \times \text{number of cores}}$

↳ Max. Throughput = Num of processes completed their execution per unit of time

- Individual process objectives:

↳ Min. Turnaround Time (TAT) = amount of time to execute particular process from creation (4) to termination (5)
TAT = CT - AT

↳ Min Waiting Time (WT) = amount of time process has been waiting in ready state
WT = TAT - BT

↳ min response Time = Time from admitted (1) to first response

↳ Completion Time (CT) = time when process complete its execution

↳ arrival Time (AT) = time when process arrived in ready state

↳ burst time (BT) = time required by process for its execution (running)

I/O burst (waiting)
CPU burst (running)
waiting time (ready)

at the end, these will be a CPU burst to terminate

CPU Scheduler → the one that select the process in ready list

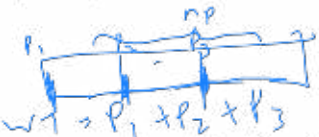
dispatcher → module that give control of CPU to process

↳ dispatch latency = time to stop process n start new one

Scheduling algo:

Uniprocessor System

FCFS (First come first served)
- Non preemptive
- Convoy effect → short process suffers in waiting times due to an earlier arrived long process
- background processes



WT → $P_1 + P_2 + P_3$

Average WT = $(P_1 + P_2 + P_3)/np$

SJF (Shortest job first) → optimal for avg waiting time
prioritize processes based on their CPU burst lengths (planning)
Shorter → higher priority
solve FCFS convoy effect
- Non preemptive (SJF)
  - once process start must wait till completes its CPU burst
- Preemptive (Shortest Remaining Time First/SRTF)
  - new process with CPU burst length less than remaining current running process, preemption occurs

Priority based Scheduling
- highest priority
- Preemptive & Non Preemptive

Prob of Starvation
Lower priority process nvr get executed
↳ solutions (aging) → as time progresses, slowly increase priority of process not able to execute

Round Robin (RR)
Process allocated to CPU every fixed q time (preempted then enter end and inserted at the end of the ready queue)
↳ usually and 10–100 milliseconds

lead to
↳ Large q → go to FCFS
↳ Small q → alot of context switches; high overhead
- Foreground processes

if n processes in ready queue
mean waiting times ≤ (n-1)q
- higher avg waiting/turnaround time than SJF but better response
- Preemptive

Multilevel Queue Scheduling
Foreground (RR) → handling I/O nit to be interactive (interactive)
background (FCFS) → nid not be interactive sch overhead can be low (batch)

Solution:
- Ready queue is partitioned into several queues, each has their own scheduling algorithm

2 Scheme for inter-queue scheduling (Scheduling among queues)
- fixed priority scheduling (foreground then background) (Look then type)
  - ↳ Starvation for low priority queues
- time-slice based scheduling (each queue type gets fixed time (eg. foreground 80ms, background 20ms)

Partitioned Scheduling (APP)
(asymmetric)
- each process mapped to 1 core
- separate uniprocessor CPU scheduling for each core

adv:
- easy & simple extension of uniprocessor to multiprocessor scheduling

global Scheduling
multiprocessor
- Symmetrical
  - when any core become idle, CPU scheduler runs and allocates next process
  - selection based on global strategy (FCFS, SJF, etc)
  - same process could execute on different cores at diff times (need serialized process to solve)

adv
↳ good CPU utilization
↳ fair to all process

disadv
↳ not scalable
↳ locking needed in scheduler
↳ high complexity
↳ high synchronization overhead
↳ overhead is high

④ Producer - Consumer prob = Synchronization prob in OS due to

Producer → process which is able to produce data/item

Consumer → process which is able to consume data/item by producer from buffer

one resource is shared between more than 1 process at the same time which lead to data inconsistency (Producer shouldn't produce data if full, Consumer shouldn't consume data if empty, access to buffer should be exclusive)

producer → [OS] buffer → Consumer

↳ Solution : semaphores (Software based synchronize solution)

non negative

atomic = non-interruptible

accessible to wait(s) P (to test)

signal (s) V (to increment)

2 types of semaphores

↳ counting semaphore unrestricted (used for multiple instance)

↳ binary semaphore 0 or 1

pro

no context switching

con

inefficient if critical sec are long

incorrect usage will lead to starvation → process stuck in sem queue list

↳ example = priority based LILO

release lock ⇓ lock = false

Semaphore for producer consumer

- mutex → binary for mutual exclusion

- full → counting for consumer to identify buffer to consume

empty → counting for producer to identify empty buffer to fill

Race condition = undesirable situation when a device attempts to perform 2 or more operation at same time

while(1) {
entry section
critical section
exit section
remainder section }

- mutual exclusion ①
- progress ②
- bounded waiting ③

(changing common var, updating table, writing a file)

Critical - Section problem = how to ensure at most 1 process is executing its critical sec at given time

- Test n Set ① ②
  ↳ used a shared boolean var lock
  - execute atomically
  - bf entering critical sec, process inquires lock
  - if lock = true wait till false
  - if lock = false, process can enter (critical sec n set lock = true)
    ↳ other process can executes
  - lock reset to false only in exit
  - not satisfy bounded waiting

- Semaphore
  - Variable shared : semaphore mutex
  - if mutex = 1 ⇒ execute wait(mutex) ⇒ can enter critical sec (acquire)
  - signal(s) (release)

bounded-buffer problem (semaphores)

Producer
wait (empty)
wait (mutex)
add()
signal (mutex)
signal (full)

Consumer
wait (full)
wait (mutex)
consume()
signal (mutex)
signal (empty)

→ if swap dead lock if buffer empty

deadlock = 2/more process are waiting for event that can be caused by only one of waiting process

Readers - writers problem (semaphore)

↳ if writer n (writer or reader) access database simultaneously ⇒ problem

Shared data
readcount → track readers in database

- Semaphores
  mutex = 1 → binary to protects access to readcount
  wrt = 1 → binary for mutual exclusion to database

writer

```
wait(wrt);
write()
signal(wrt)
```

reader

```
wait(mutex)
readcount ++
wait(wrt)
signal(mutex)
read()
wait(mutex)
readcount --
signal(wrt)
signal(mutex)
```

Dining philosophers prob (semaphore)

Semaphore resourse[n] → number of resources (R = resourse)
↳ initialized as 1 for all resources

dining philosophers

```
wait(R[i])
wait[(i+1) % n] → for nth loop
task 1
signal(R[i])
signal[(i+1) % n]
task -
```

deadlock!

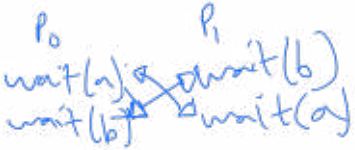
↳ to prevent:
↳ at most 4 philosophers to be hungry at time
↳ allow philosopher to pick up only if both chopsticks available
↳ use asymmetric solution
  ↳ odd = left → right
  ↳ even = right → left

disadv of semaphores

↳ deadlock n starvation

Spinlock = process spinning in loop cause by busy waiting

(5) deadlock



Resource-allocation graph

V-Vertices :
- P = process ○
- R = resource ⚄ → instances

E - edges :
- Request edge P → R ○→⚄
- Assignment edge R → P ⚄→○

- if graph
  - Contain no cycle → no deadlock
  - Contain cycle →
    - 1 instance/resource → deadlock
    - > 1 instance/resource → possible deadlock

- deadlock conditions (Possible to happen)
  - mutual exclusion
  - hold n wait
  - no preemption
  - Circular wait

  } Prevent at least 1 of 4 deadlock conditions

- OS ignored deadlock
- System is in Safe State if there exists Safe completion route with no deadlock
  - P.request <= available + ΣP.hold (prev process ∆)
- Unsafe → will Possibly deadlock (P.request > available)
  - if process release R before its completion ⇒ deadlock

deadlock avoidance (ensure system nvr enter unsafe state)
↳ Banker's algo ⇒ Check if satisfaction of resource request would lead to unsafe state

n = num of process
m = num of resource type

Available[j] (vector of length m) = number of instances of Resource R_j

Max[i,j] = number of instances P_i may request R_j

Allocation[i,j] = number of instances R_j allocated to P_i

need[i,j] = number of instances P_i need from R_j to complete its task = Max[i,j] − Allocation[i,j]

Check need VS Available Process by process

	allocation $R_0 \to R_j$	max $R_0 \to R_j$	Available $R_0 \to R_j$	need $R_0 \to R_j$
P_0				
⇩				
P_i				

! every process satisfy ⇒ Available = prev Available + Allocation
if there is sequence → safe

Resource-request algo

Request$_i$[j] = num of instances P_i want from R_j

Step

1. Request$_i \leq$ need$_i$, else raise error
2. Request$_i \leq$ avail, else P_i must wait
3. Pretend allocation to P_i by
   Avail = Avail - request
   allocation$_i$ = allocation$_i$ + Request$_i$
   need$_i$ = need$_i$ - Request$_i$
4. run safety algo
   - safe → resource allocated to P_i
   - unsafe → P_i must wait, restore state

disk Scheduling

1) FCFS

adv = no indefinite postpone

dis-adv : doesn't try to optimize seek time
- may not provide ↑ service
- may have wild swings back n forth

2) SSTF

adv - Avg response time ↓
- ↑ throughput

dis-adv - may cause Starvation
- overhead to calculate seek time

3) scan (1 direction only) (visit smallest 0 or largest depend on direction)

adv - high throughput
- low variance of response time
- Avg response time

dis - long waiting

4) C-Scan (try 1 direction) (visit 0) (visit largest IEE 200 track 199)

adv - more uniform wait time than scan

5) Look & C look (same as scan but not visiting largest/smallest)

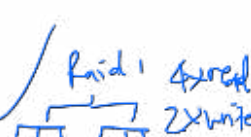
Look: Step by Step

C look: From largest to smallest

disk reliability

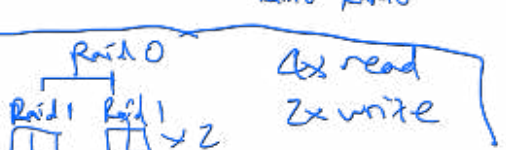
Raid 0 multi disks act as 1

Raid 1 (mirroring) write 2 disks X 2



Raid 0+1 (mirror of strip) 01

Raid 1+0 (strip of mirrors)/Raid 10



- Classic algo (FCFS, SJF, RR) fail cause they don't prioritize deadlines
- Real-time CPU scheduling (short-term scheduler).
  - Fixed-priority scheduling
    - priority fixed for each recurrent process
    - Rate monotonic (RM)
      - simple n good for small n popular in real-time OS industry, predicability even when overload
      - assign priorities based on process periods/minimum release-separation time (T)
        - Shorter T = ↑ priority
        - ties broken arbitrarily
      - con - doesn't always prioritize urgent process
    - Deadline Monotonic (DM)
      - better than RM - but cannot change priority across process
      - assign priorities based on deadlines
        - Shorter D = ↑ priority
        - ties are broken arbitrarily
  - Dynamic-priority scheduling
    - Earliest Deadline first (EDF)
      - more powerful than RM n DM but harder implementation n misbehaviour during overload
      - assign priority based on process instance deadline
        - instance with shorter deadline = ↑ priority
          - D ≠ 0
        - Ties are broken arbitrarily

- Virtualization = technique that uses software called Hypervisor (VM managers) to create abstraction of HW
  - HW divide into multiple VM, each VM runs own OS (guest OS)
  - function: - allows more efficient utilization of HW
    - cost effective
    - low latency n agile execution environment
    - feature replication
    - key tech driving cloud computing
      - cost effectively
      - enable concept of platform as service
  - challenges: - require hypervisor layer
    - real-time OSs require diff solutions
  - levels: - Full virtualization (VMware esxi, Microsoft virtual Servers)
    - Para-virtualization (Xen, Vmware, Xtratum)
      - VM doesn't virtualize hard to implement parts of H/W instruction set

- function of Hypervisor → usually 2 orders of magnitude smaller than general OS
  - Creation n management of VM
  - communication between VMs
  - VM Migrations

(7) Compiler = source code to object code
Linker = combine object modules to resolve references
Loader = - create process image
- allocate memory
- load process image to allocated memory

memory allocated during compile time is called static memo
(cannot increase / decrease size of memory allocated)

memory allocated during execution time is called dynamic mem

used in process image | relocation register

absolute address format (in compile time/load time binding)

if address generated by CPU < base
≥ base + limit ⇒ error

else memory address = address generated by CPU

relative address format (in execution time binding)

if address generated by CPU ≥ limit ⇒ error

else memory address = address generated by CPU + base

- address space = all address accessible by a process
- Logical address = address used in code, generated by CPU while executing
- Physical address = address used to access physical memory (seen by memory unit)

2 approaches to allocate memory among process:

↳ Contiguous allocation
(logical address remains in physical memory)

- Fixed partitioning
  - memory partitioned into regions with fixed boundaries

- Dynamic Partitioning
  - Hole = block of available memory
  - Various size hole scatter in memory, new process get assign appropriate hole
  - OS maintain info about
    - allocated partition
    - free partitions (hole)

Dynamic Storage-allocation problem:
how to satisfy request of size n from list of free holes:

- First fit → allocate first hole that is big enough
- Best fit → allocate smallest hole that is big enough (least overhead)
  (produce smallest leftover hole)
- worst fit → allocate largest hole
  (produce largest leftover hole)

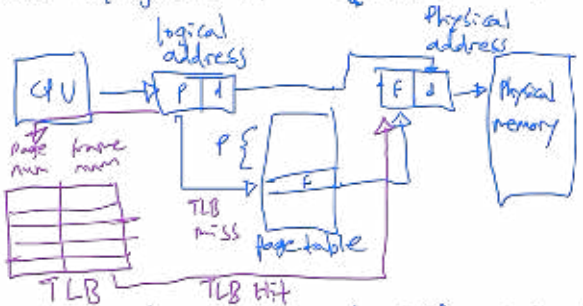
Fragmentation

↳ External = enough total memory space exists to satisfy request but it is not contiguous
Happen outside a partition

↳ reduce by compaction
↳ shuffle memory contents to place free memory tgt in 1 large block
↳ possible only if re-locatable address format is used in process image
binding done during execution time

↳ Internal = allocated memory slightly larger than requested memory

- Paging
- Frames = divide Physical memory into fixed-sized blocks
- Pages = divide logical memory into blocks of same size
- Page table = use for translate logical to physical addresses
- external fragmentation eliminated
- internal fragmentation possible (last page many not occupy the entire frame)



Page size = 2^n byte

logical address space = 2^m byte

Number of pages = 2^{m-n} byte

m-n bits; n bits

p | d

page num | Page offset

- Page table base register (PTBR) points to page table
- Page table length register (PTLR) indicates size of page table
- effective memory access time = 2 μ
  - 1 for page table & 1 for data/instruction
  - μ → memory cycle time
- $2^{\text{Page num}}$ entries in page table

→ can be reduce by use of a fast lookup HW cache called associative reg or translation look-aside buffers (TLB)

- TLBs lookup = ε time unit
- Hit ratio = α = % of time TLB hit
- Effective Access Time (EAT) = $(\mu + \varepsilon)\alpha + (2\mu + \varepsilon)(1-\alpha)$
  $= (2-\alpha)\mu + \varepsilon$

1 byte = 8 bits

1 Kbyte = 2^{10} byte = 1024 bytes

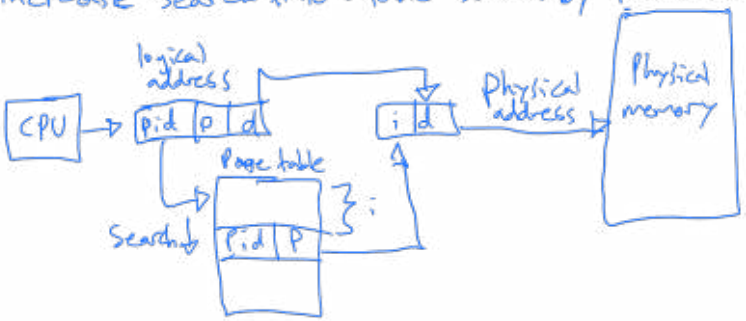
1 Mbyte = 2^{20} byte

1 Gbyte = 2^{30} byte

1 Tbyte = 2^{40} byte

Inverted Page table

- Single table with 1 entry for each physical frame <Process-id, Page no>
  logical address <Process-id, page no, offset>
- increase search time : table sorted by physical address but lookups on logical address

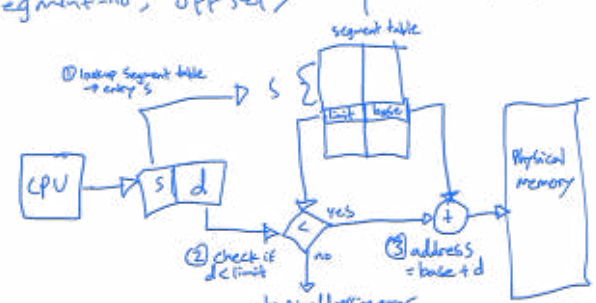


Segmentation

↳ memory management scheme that break program up into its logical segments n allocating space for each segments
↳ can be variable size

Address Translation <segment-no, offset>

base = Contains starting physical address
Limit = Specifies length of the segment



STBR (segment-table base reg) → points to segment table's location in memory
STLR (segment-table length reg) → indicates num of segments used by a program
Suffer external fragmentation