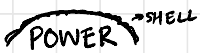
Operation Systems

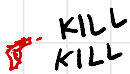
What do i not know?
No giving assignments! (except BU)
attendance doesnt matter much

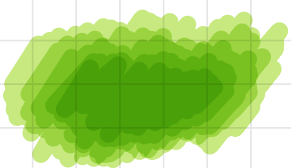
CORNELL NOTEMAKING METHOD
WRONG

ISHMA METHOD

Project + presentation (10%)
Assignments (4 off) (10%)
Mid Term (15+15 = 30%)
Final (50%)







Virtualization:

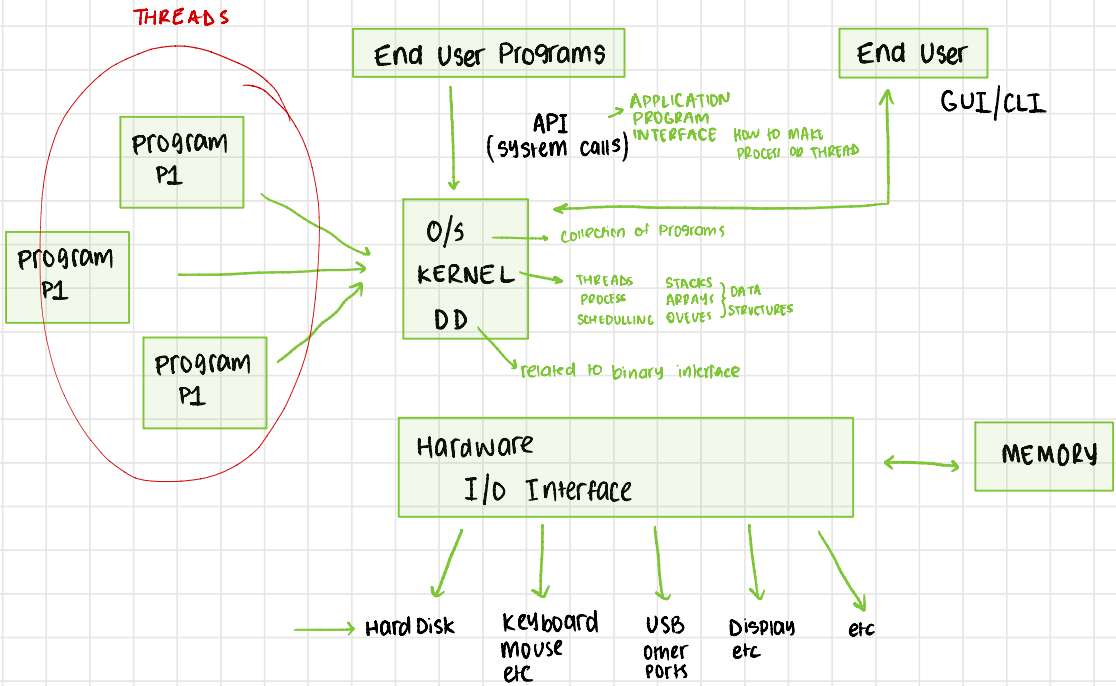
Let us take the most basic of resources, the CPU. Assume there is one physical CPU in a system (though now there are often two or four or more). What virtualization does is take that single CPU and make it look like many virtual CPUs to the applications running on the system. Thus, while each application thinks it has its own CPU to use, there is really only one. And thus the OS has created a beautiful illusion: it has virtualized the CPU.

Concurrency: There are certain types of programs that we call multi-threaded applications; each thread is kind of like an independent agent running around in this program, doing things on the program's behalf. But these threads access memory, If we don't coordinate access to memory between threads, the program won't work as expected. First, the OS must support multi-threaded applications with protection (such as locks and condition variables). Second, the OS itself was the first concurrent program — it must access its own memory very carefully or many strange and terrible things will happen.

Persistence: Making information survive computer crashes, disk failures, or power outages is a tough and interesting challenge and needs persistence.

Operating System

ABI
APPLICATION BINARY INTEFACE

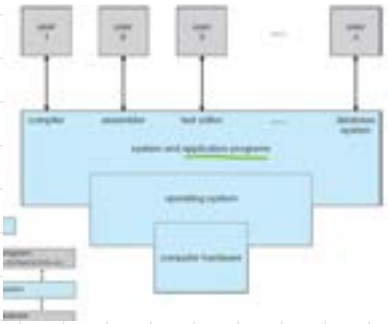


3 CONCEPTS in O/S

- Virtualization → PROCESS IS A VIRTULISATION → not in BATCH, give acess of all process to hardware together
  - architecture, H/W for each user programs
- Concurrency → multiple threads → inside the process
  - ↓ similar to a process but part of a process
  - divide data into chunks
    - T_1 T_2 T_3 → threads
    - 50K 50K 50K
  - for (i=0 200000000
    - a[i] = b[i] + c[i]
- Persistence
  - HOW to run more than one instruction stream in one program using API's
  - ↓ not in memory
    - data/program storage that can survive reboots

CMP 1

- Computer system can be divided into four components:
- **Hardware** – provides basic computing resources
- **Operating system** –
- **Application programs** –
- **Users**



→To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory

Some important terms:

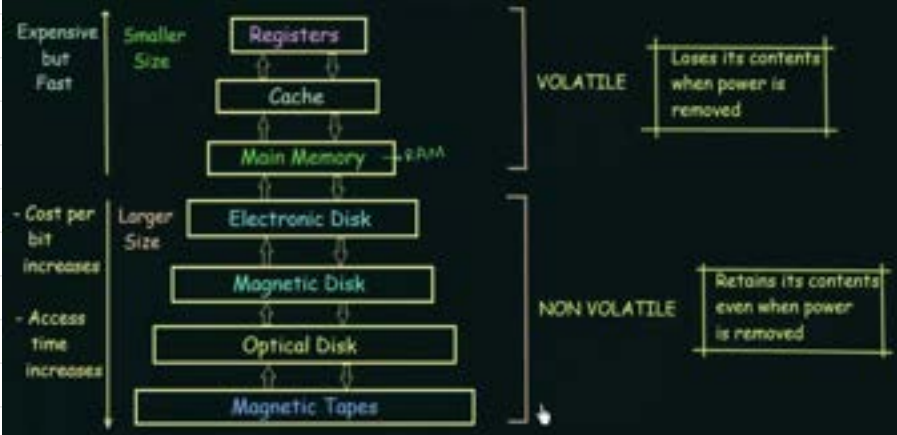
1) Bootstrap Program: →The initial program that runs when a computer is powered up or rebooted.
→It is stored in the ROM.
→It must know how to load the OS and start executing that system.
→It must locate and load into memory the OS Kernel.

2) Interrupt: → The occurence of an event is usually signalled by an Interrupt from Hardware or Software.
→ Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by the way of the system bus.

3) System Call (Monitor call): → Software may trigger an interrupt by executing a special operation called System Call.

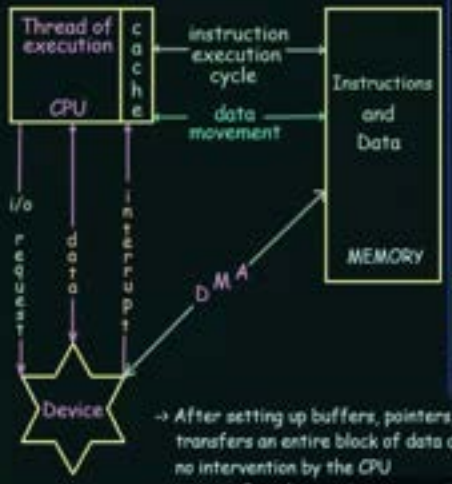
So, that Operating system is already residing or stored

Basics of Operating System (Storage Structure)



which basically are secondary storage devices, are nonvolatile.

Working of an I/O Operation:



-> To start an I/O operation, the device driver loads the appropriate registers within the device controller

-> The device controller, in turn, examines the contents of these registers to determine what action to take

-> The controller starts the transfer of data from the device to its local buffer

-> Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation

-> The device driver then returns control to the operating system

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement

To solve this problem, Direct Memory Access (DMA) is used

-> After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU

So, from these registers, the appropriate registers that are required for

Single Processor System

↳ single processor

↳ has multiple special purpose processor which device specific tasks

multiprocessor system

↳ has 2 or more processors

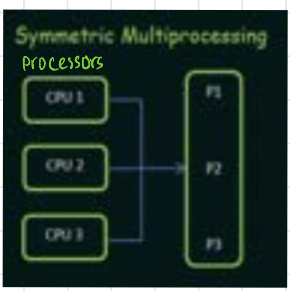
↳ they share clock/memory/devices/bus

ADV

↳ increased throughput → amount of data transfered from 1 place to another → more processors

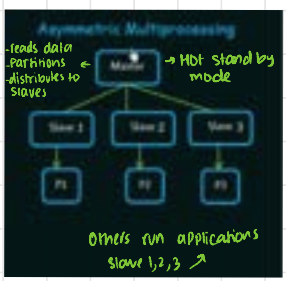
↳ economy of sales → as processors share resources

↳ increased reliability → if 1 processor fails other will work



↳ all cpus similar

↳ all processors acess same memory



↳ master and slave

↳ all processors acess different memory

↳ independent memory

clustered systems

↳ multiple processors

↳ composed of 2 or more individual systems

ADV

↳ high availability → if 1 system fails the others can take care of task

↳ can also be structured symmetrically or asymmetrically

OS TYPES

Multiprogramming

↳ executes multiple jobs

↳ no user interaction with comp system

↳ processes are in RAM

↳ only one process executed at a time

↳ if I/O operation occurs only then other process executed

if didn't exist then 1 job would take all CPU time until completion
↳ when us I/O devices CPU would be idle

Timesharing/Multitasking

↳ CPU executi multiple jobs

↳ switches so quickly that user interaction with programs is possibl

↳ allows many users to share the comp simultaneously

↳ CPU scheduling

↳ processe are in CPU

↳ multiple processes running concurrently and prempted after a fixed time slice

↳ increases CPU utilization

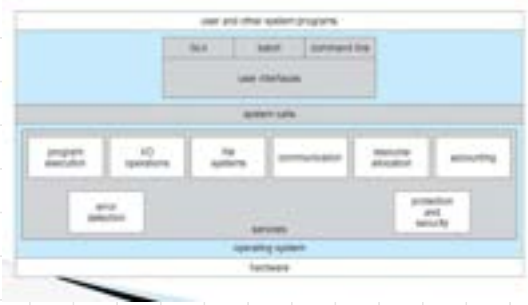
↳ dosent leave CPU idle

OPERATING SYSTEM SERVICES

CHP 2

One set of operating-system services provides functions that are helpful to the user:

- **User interface** - Almost all operating systems have a user interface (UI).
  - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
- **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action
- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
- **Accounting** - To keep track of which users use how much and what kinds of computer resources
- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts



CLI
↳ type
↳ interpreters are known as shells

GUI
↳ icons

IPC

IF PEARS PUC

SYSTEM CALLS

System calls

↳ Provide an interface to the services made available by an OS in kernel mode

↳ Made by programs to acess certain resources

MODES A PROGRAM CAN EXECUTE

User mode

safer mode hence mostly used

no direct acess to

↳ memory / hardware / resources

when needs acess to Privisilled resources it makes a system call to switch from user to kernel mode

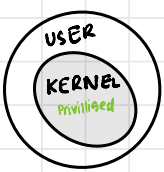
Kernel mode

dosnt happen in User mode

if program crashes during execution whole system comes crashes

direct acess to (Privilege mode)

↳ memory / hardware / resources

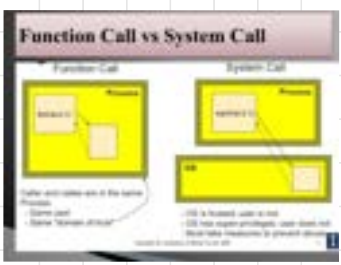


Pass Parameter in OS

↳ registers

↳ Pointers

↳ stack



TYPES OF SYSTEM CALLS

1. **Process Control**
   - load
   - execute
   - create process
     - fork on Unix-like systems or
     - NtCreateProcess in the Windows
   - terminate process
   - get/set process attributes
   - wait for time, wait event, signal event
   - allocate free memory

2. **File management**
   - create file, delete file
   - open, close
   - read, write, reposition
   - get/set file attributes

3. **Device Management**
   - request device, release device
   - read, write, reposition
   - get/set device attributes
   - logically attach or detach devices

4. **Information Maintenance**
   - get/set time or date
   - get/set system data
   - get/set process, file, or device attributes

5. **Communication**
   - create, delete communication connection
   - send, receive messages
   - transfer status information
   - attach or detach remote device

6. **Security**
   - Protection
   - Control access to resources
   - Get and set permissions
   - Allow and deny user access

DIPS FC

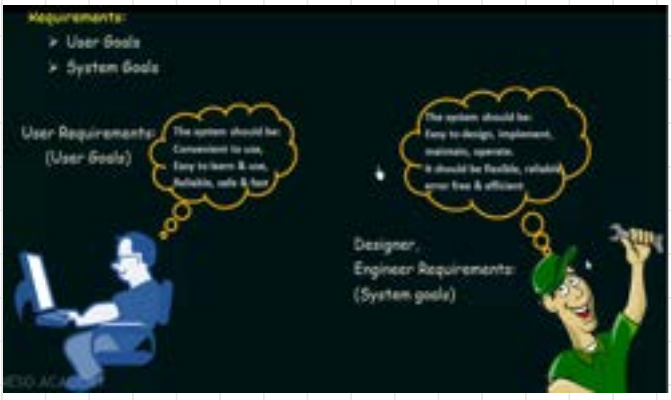
FIPC

SYSTEM PROGRAMS

- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
- **Background Services**
  - Launch at boot time
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as services, subsystems, daemons
- **Application programs**
  - Run by users

FIPC

OS DESIGN and IMPLEMENTATION



OS STRUCTURES

1) Simple structure

↳ if one program fails entire system crashes

↳ not well protected/structure

2) Monolithic structure

↳ too many functions packed in one level → kernel part

↳ hence implementation/maintainece difficult

↳ too change one thing will have to touch entire kernel

3) Layered structure

pros

↳ functionalities not packed into some layer

↳ easy to implement/debug

↳ if one layer problem then only have to look at that layer

↳ hardware protected by layers above/no direct acess by the user

cons

↳ a layer can only use those below the layer

↳ not efficient

↳ when a layer wants to use services of below layer it will have to go down one by one

↳ may not be fast

4) Microkernels structure

pros

↳ micro kernel provides core functionalities

↳ other services are implemented as a user level program

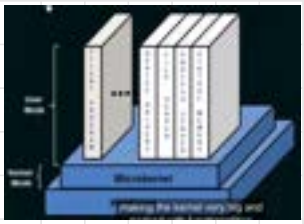
↳ communication b/w client programs and system programs is made through message passing

↳ can continue running mostly in user mode so less chances of system crash

cons

↳ performance decrease due to increase system overhead

↳ due to message passing



5) Module structure

best

↳ uses OOP techiniques

↳ core kernel has core functionalities

↳ other functionalities are in the form of modules

↳ modules loaded into kernel when required

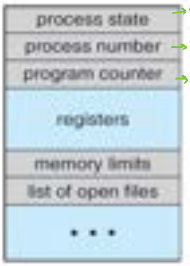
↳ doesnt have to go through all layers like in layer approach

↳ each module can communicate with other module directly

↳ no need for message passing hence no performance decrease

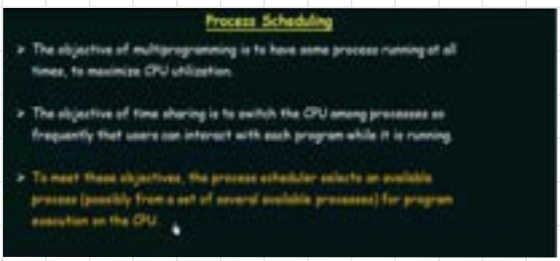
CHP 3

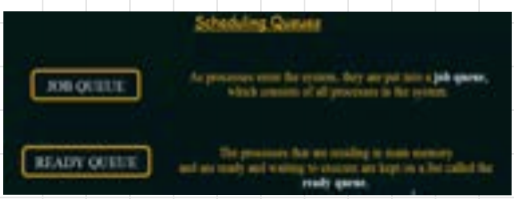
PCB



→ running/waiting/ready...
→ process ID
→ address of next instruction

SNCR





Process alternate b/w these 2 states

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Context switch

Pure overhead → cost (resources/time spent...) that is involved in doing something

↳ does no usefull work while switching

- ↳ P_i executing (current program)
- ↳ interrupt occurs
- ↳ system saves state of P_i context (present in PCB of the process)
- ↳ CPU switches to P_2 and suspends P_i
- ↳ P_2 executes and finishes
- ↳ restore state of P_i
- ↳ resume execution of P_i

Kernels actions in context switch

- ↳ Saving the current Process state
- ↳ selecting the next process
- ↳ loading the next process state
  - ↳ restoring saved process
- ↳ Handing control to next Process
  - ↳ resuming execution

fork():

↳ creates a seperate duplicate process

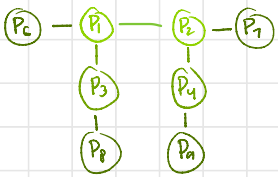
↳ Parent —its duplicate→ child

↳ ID is different

main(){

fork()
fork()
fork()
}

$2^3 = 8$ processes created (3 → no of fork calls)



exec():

↳ program in parameter will replace entire process

↳ ID remains same

Interrupt

cause OS to change CPU from current task and to run a kernel routine

CHP5

CPU SCHEDULER

↳ whenever CPU becomes idle

↳ OS selects a process in ready queue to execute

CPU scheduling

Dispatcher

↳ giving control to process selected by short term scheduler (CPU scheduler)

↳ time to stop one process and start another is dispatch latency

CPU scheduling circumstances

1. switch from running to waiting (I/O request)
2. switch from running to ready (interrup occurs)
3. switch from waiting to ready (completion of I/O)
4. when process terminates

choice given will CPU given to same process OR to some other process

no choice of scheduling as a new process must be selected

2 ways CPU SCHEDULING TAKES PLACE

Preemptive scheduling

↳ circumstance 2 and 3

↳ CPU can be taken away from a process before it has

↳ went into ready state

non preemptive scheduling/cooperative

↳ circumstance 1 and 4

↳ CPU will never be taken away from a process until and unless it has

↳ completed execution (terminated)

↳ went into waiting state

CONS

shared memory

↳ P_1 writing a sentence in shared memory

↳ P_2 came, P_1 preempted/halted

↳ CPU given to P_2

↳ P_2 when reads from shared memory

↳ P_2 reads inconsisted data as P_1 wasnt done writing

FCFS (First Come First serve)

- ↳ when process enters ready queue
  PCB linked to tail of the queue
- ↳ avg waiting time long
- ↳ non preemptive

turnaround time = Completion time - Arrival time

waiting time = turnaround time - Burst time

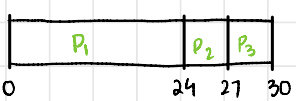
CONS

- ↳ dissasterous to allow one process CPU for extended time
- ↳ CONVOY EFFECT
  - ↳ higer burst time arrive before smaller
  - ↳ smaller process have to wait long time

Q)

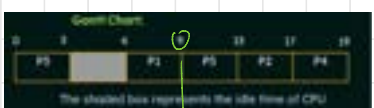
process	Burst Time ms
P_1	24
P_2	3
P_3	3

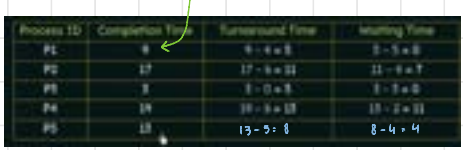
Gantt Chart



waiting time = 0 + 24 + 27 = 51 ms

avg waiting time = 51/3 = 17 ms





avg turnaround time = $\frac{5+11+3+13+8}{5}$ = 8 units

avg waiting time = $\frac{0+7+0+11+4}{5}$ = 4.4 units

SJF (shortest job first)

↳ if CPU burst are same FCFS is used to break tie

↳ preemptive or non preemptive

turnaround time = Completion time - Arrival time

waiting time = total waiting time - ms process executed - Arrival time

CONS

↳ no way to know length of next CPU burst

SOLUTION

↳ we can approximate

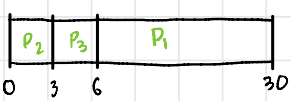
↳ we can predict using previous CPU bursts

NON PREEMPIVE

Q)

process	Burst Time ms
P_1	24
P_2	3
P_3	3

Gantt Chart



waiting time = 0+3+6 = 9 ms (as no arrival time given assume 0)

avg waiting time = 9/3 = 3 ms

PREEMPTIVE

Process ID	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

waittime: P_1 (10-1-0) + (1-0-1) + (17-0-2) + (5-0-3)

avg = 26/4 = 6.5

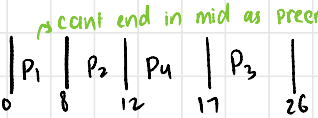
turnaround time = (17-0) + (5-1) + (26-2) + (10-3)

avg = 52/4 = 13

NON PREEMPTIVE

(we)

(cant end in mid as preemph)



INTER PROCESS COMMUNICATION (IPC)

↳ Mechanisim for Processes executing concurrently to communicate and synchronise their actions

MESSAGE PASSING

↳ TWO OPERATIONS

↳ send message → fixed size message/variable

↳ receive message

In order to communicate using message passing

↳ establish a communication link b/w them

↳ Physical (shared memory/ hardware bus)

↳ logical (logical properties)

↳ exchange messages via send/receive

Priority Scheduling

↳ CPU allocated by priority

CON: Starvation → low priority may never execute

SOLUTION: Aging → increase priority as time progresses

turn around time: Completion – Arrival time

Waiting time: Total waiting time – ms process executed – Arrival time

↓ Preemptive

Waiting time: turn around time – burst time

↓ non preemptive

PREMPTIVE

Process ID	Arrival Time	Burst Time	Priority
P1	0	~~10~~ 9	2
P2	5	28	0
P3	12	2	3
P4	2	~~10~~ 7	1
P5	9	16	4

Gantt Chart

0	2	5	9	12	14	21	37
P1	P4	P2	P5	P3	P5	P2	

NON PREMPTIVE

Waiting time: (40-2-0) + (5-0-5) + (49-0-12) + (33-3-2) + (51-0-9)

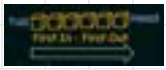
avg = 29/5

→ best for time sharing

Round Robin (RR) Scheduling

↳ small time slices for each process (time quantum)

↳ FCFS but has premption



FIFO in ready queue

Tail → new processes

Head → nex process to get CPU

2 Possibilities

↳ burst time < time quantum

↳ CPU released

↳ CPU scheduler gets next process in ready queue

↳ burst time > time quantum

↳ timer goess off

↳ interrupt to OS

↳ context switch, process moved to tail of ready queue

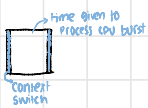
↳ CPU scheduler gets next process in ready queue

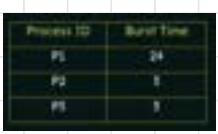
CONS

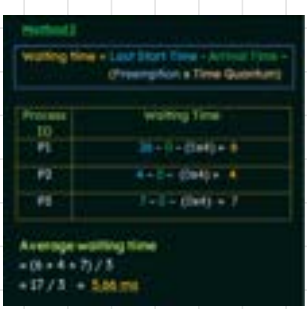
time quantum

↳ too small : too many context switches

↳ too big : starve for long time







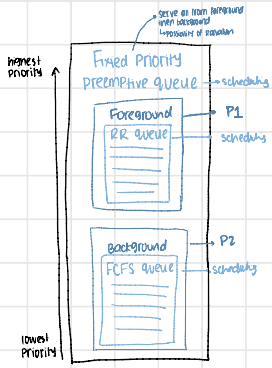
NO TALKING CLASS SAY BAHIR

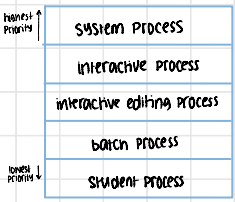
Multi-level Queue Scheduling

Scheduling algorithms for situations in which processes are classified in different groups

E.g

Foreground (interactive)	Background (batch)
↳ need to be quick	↳ dosent need to be as fast
↓	↳ FCFS
Priority will be higher	
↳ Round Robin	





↳ Partions ready queue into several seperate queues

↳ Scheduling takes place within and among these queues

↳ Each process permamentaly assigned to 1 queue

based on memory size, Piority and process type

↳ Each queue has it's own scheduling algorithm (FCFS, SJF, RR...)

Multi-level Feedback Queue Scheduling

↳ allows processes to move between queues

↳ seperate processes according to their CPU bursts

↳ If a process uses too much CPU time

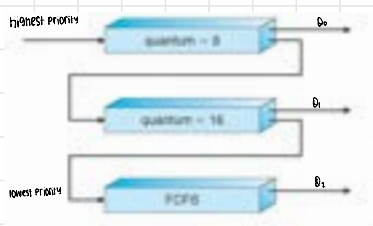
then moved to lower priority queue (so everybody gets a chance)

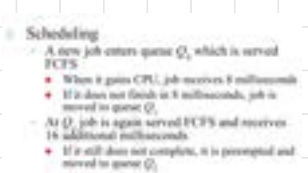
↳ I/O and interactive processes are in higher Priority queues

as they need quick responses

↳ A process in lower priority queue for too long → aging

may be moved to higher priority queue (prevents starvation)




Scheduling

- A new job enters queue Q_0 which is served FCFS
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue Q_2

Parameters

↳ no. of queues

↳ scheduling algo for each queue

↳ method to upgrade process to higher priority

↳ method to demote process to lower priority

↳ method to determine queue when process needs service

↳ demote/upgrade

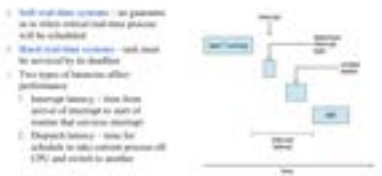
Multi-Processor Scheduling

NOTES MISSING

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
- **soft affinity:** When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so known as **soft affinity**.
- **hard affinity:** allowing a process to specify a subset of processors on which it may run.

Understanding

Real-Time CPU Scheduling



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

CONFUSED

LECTURE 19,20,21

Process Concept

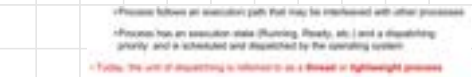
↳ Process are executed programs that have

↳ Resource Ownership — unit → process/task

↳ Process includes virtual space → to hold process image

↳ OS prevents unwanted interference b/w processes

↳ Scheduling / Execution



MOTIVATION

- Most modern applications are multithreaded
- Threads run within application (process)
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Thread control block

↳ information associated with each thread

↳ Program counter

↳ CPU registers

↳ CPU scheduling info

↳ Pending I/O info

Process control block (PCB)

↳ information associated with each process

↳ Memory management info

↳ Accounting info

Thread operations

↳ Spawn : a thread within a process may spawn another thread

↳ provides new thread instruction pointer, arguments, register, stack

↳ Block : a thread needs to wait for an event

↳ saves its user registers, program counter, stack pointers

↳ Unblock: when the event for which the block occurs

↳ Finish: when thread completes

↳ its register context and stacks are dellocated

Thread states

↳ running

↳ ready

↳ blocked

Process

↳ A program in execution

↳ can have 1 or more threads

Threads

↳ unit of execution within a process

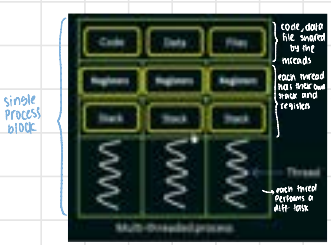
↳ It compromises of

1. Thread ID
2. Program counter
3. Register set
4. Stack

↳ shares code section, data section, os resources with other threads of same processes

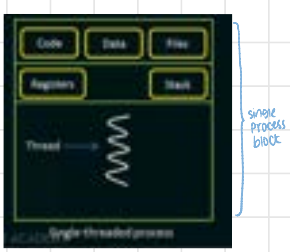
Multi Thread

↳ Performs multiple tasks at a time



Single Thread

↳ Performs one task at a time



MULTI-THREADING BENEFITS

↳ Responsiveness: may continue execution if process is blocked/performing lengthy operation

dedicated threads for handling user events

↳ Resource Sharing: threads share memory and recources of a process → allows diff threads activity within same address space

easier than shared memory/message passing

↳ Economy: cheaper than process creation as threds share recources

allocating memory/recources is costly

thread switching has lower overhead than context switching

↳ Scalibility: utilization of multiple cores for parallel execution

increases concurrency

Multicore Programing

single core { T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 ... }

multicore { core 1: T_1 T_3 T_1 T_3 ... ; core 2: T_2 T_4 T_2 T_4 ... } parallel execution

PARALLELISM

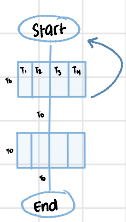
↳ act of managing multiple computations simultaneously

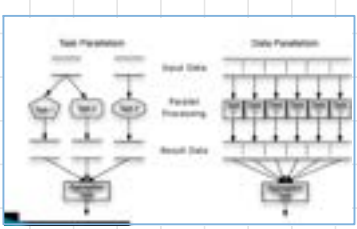
DATA PARALLELISM ← IMP → TASK PARALLISIM

↳ focus on distributing data across diff parallel computing nodes

↳ focus on distributing threads across diff parallel computing nodes

Data Parallelism	Task Parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum load balance on multi processor system.	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.

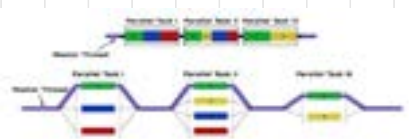




CONCURRENCY

↳ act of managing multiple computations at the same time but not simultaneously → control is switched

Fork-Join model



```
solve(problem)
  if problem is small enough
    solve problem directly (sequential algorithm)
  else
    for part in subdivide(problem)
      fork subtask to solve part
    join all subtasks spawned in previous loop
    combine results from subtasks
```

- Multicore systems putting pressure on programmers, challenges include
  - Dividing activities
    - What tasks can be separated to run on different processors
  - Balance
    - Balance work on all processors
  - Data splitting
    - Separate data to run with the tasks
  - Data dependency
    - Watch for dependences between tasks
  - Testing and debugging
    - Harder!!!

AMDAHL'S LAW

↳ speed up in latency of a task execution

$$\text{Speed UP} \leq \frac{1}{S + \frac{1-S}{N}}$$

S → serial, N → no of cores, 1 → parallel

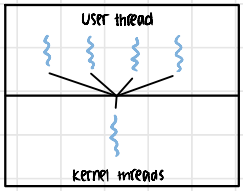
B) S: 25%, N: 2

$$\text{Speed UP} \leq \frac{1}{0.25 + \frac{0.75}{2}} = 2.28$$

MultiThreading → multiple threads at the same time

establish relationship between user and kernel thread

Many to one



PROS

↳ multiple user threads mapped to 1 kernel thread

↳ thread management is done by threaded library → so efficient

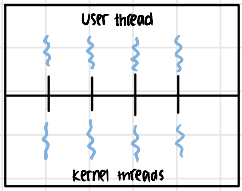
CONS

↳ if 1 thread makes a blocking system call entire process will be blocked

↳ only 1 thread can acess kernel at a time.

↳ multiple threads are unable to run in parallel on multiprocessors

One to one



PROS

↳ maps each user thread to kernel thread

↳ more concourrency → as allows another thread to run when a thread makes a blocking system call

↳ allows multiple threads to run parallel on multi processors

CONS

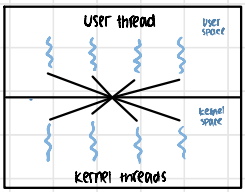
↳ each user thread requires a corresponding kernel thread

↳ overhead of creating kernel threads burdens performance

restricts no. of threads supported by the system

↳ e.g ↳ 4 core processor
↳ 5 threads
then only 4 threads work parallel
hence no of threads restricted

Many to Many



PROS

↳ kernel threads are ≤ to user threads

↳ more concorcuncy

↳ no limit to creating user threads

↳ kernel threads can run parallel on a multi processor

Hyper Threading / SMT (Simultaneous Multi Threading)

↳ more than 1 multithreading going on in the same system

e.g ↳ 4 core → virtually/logically divided into multiple processors
↳ 8 threads support

TYPES OF THREADS

User threads

- ↳ implemented by user
- ↳ requires no hardware support
- ↳ easy to implement
- ↳ if one thread blocked, entire process blocked

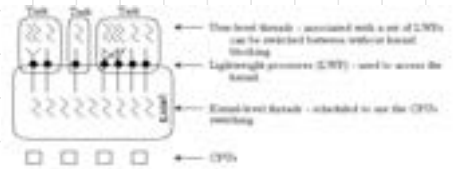
Kernel threads

- ↳ implemented by kernel
- ↳ requires hardware support
- ↳ difficult to implement
- ↳ if one thread blocked, another thread continues execution

POSIX PThreads

- ↳ Win 32 Threads
- ↳ Java Threads

- **POSIX Pthreads**
  - May be provided either as user-level or kernel-level
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - API specifies behavior of the thread library, implementation is up to development of the library
- **Win32**
  - Kernel-level library on Windows system
- **Java**
  - Java threads are managed by the JVM
  - Typically implemented using the threads model provided by underlying OS



- Linux kernel has processes and lightweight processes (LWP).
- Difference: LWPs share same address space and other resources like open files etc. As some resources are shared so these processes are considered to be light weight as compared to other normal processes and hence the name light weight processes.
- Therefore, effectively we can say that threads and light weight processes are same.
- Thread is the term used at the user level while LWP is a term used at kernel level.

Threaded Libraries

$$SUM = \sum_{i=1}^{N} i$$

Asynchronus Threading

- ↳ once parent creates child thread, parent resumes execution
- ↳ they execute concurrently and independently of one another
- ↳ Threads are independent → no dependency
- ↳ little data sharing

Synchronus Threading

- ↳ Once parent creates 1 or more children
  - ↳ it waits all child to terminate before it resumes
- ↳ Threads by parent work concurrently
- ↳ significant data sharing among threads

Race condition

↳ when several processes manipulate /acess the same data concurrently and outcome depends on order in which acess take place

results in inconsistency

SOLUTION

Process Synchronization

The orderly execution of coperating process that share an address space

Producer and consumer run cocurrently using

BUFFER

Unbounded buffer

↳ has a size on buffer

↳ consumer waits for new items

↳ producer can always produce new items

bounded buffer

↳ fixed buffer size

↳ consumer must wait if buffer empty

↳ producer must wait till buffer full

↳ use counter variable (++ produced, -- consumed)

Critical section problem

↳ a system with n processes

↳ each process has a segment of code called critical section

↳ where process may be changing common variables
updating table
writing a file

Three requirements

↳ **mutual exclusion:**

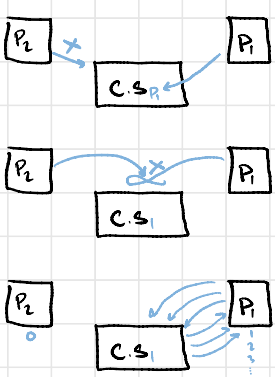
If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

↳ **Progress:**

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

↳ **bounded waiting:**

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.



Peterson's solution → humble

↳ software solution to the critical section problem

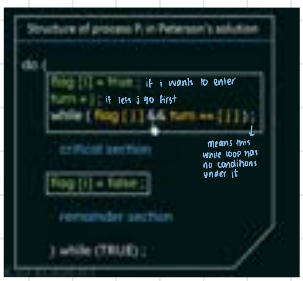
↳ 2 processes that alternate execution → b/w critical section, remainder section

↳ The 2 processes share 2 variables

↳ int turn; → indicates whose turn to enter critical section

↳ boolean flag[2]; → indicates if process ready to enter critical section

↳ satisfies all 3 critical section requirements



→ satisfies mutual exclusion

Test and set Lock

↳ hardware solution to the critical section problem

↳ Processes share shared lock variable = 0 → unlocked, 1 → lock } helps satisfy mutual exclusion

* lock = 0 in begining

↳ If lock = 1, then wait till it becomes free

↳ If lock = 0, then execute critical section and lock = 1

↳ satisfies mutual exclusion

↳ does not satisfy bounded-waiting

when while = 0 → F
= 1 → T

Atomic Operation → hardware instruction

↳ Single, uninterrupted operation

eg

TestAndSet() is run as a single operation which can not be interrupted

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&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);
```

Compare and Swap Instruction

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

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

   return temp;
}
```

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

   /* critical section */

   lock = 0;

   /* remainder section */
}
```

- If two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

On Intel x86 architectures

```
lock cmpxchg <destination operand>, <source operand>
```

Mutex locks/spin lock

↳ to protect critical section/prevent race conditions

↳ aquire() } both calls must be atomic

↳ release()

↳ requires busy waiting

while process in critical section any other process that tries to enter critical section must loop continuously to check if condition true

↳ wastes CPU cycles (that some other process could use productively)

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

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

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

Pthreads Synchronization – Mutex Lock

```
#include <pthread.h>
pthread_mutex_t mutex;

/* create the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

1st Arg: Mutex initializer
2nd Arg: Attributes, NULL means no error checks will be performed

Pthreads Synchronization – Mutex Lock

- The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions.
- If the mutex lock is unavailable when pthread_mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_unlock().

```
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

Semaphore

↳ Software solution to the critical section problem

↳ technique to manage concurrent processes

↳ an int variable S (always +ve) shared b/w threads
(S → semaphore)

↳ S can only be accessed through 2 atomic operations

↳ wait() → to decrement
- S<=0 → some process already in critical section so can't use shared resource → wait
- S>0 → can use shared resource → don't wait

↳ signal() → to increment → tell other processes that shared resource is free to be used

* S=1 in begining

TWO TYPES OF SEMAPHORES

Binary Semaphore

↳ S can only have 2 values
- 0 → some process already executing so not free to use shared resource
- 1 → free to use shared resource

↳ behave similarly to mutex locks (provide mutual exclusion)

↳ can implement a counting semaphore S as a binary semaphore

Counting Semaphore

↳ S can multiple values

↳ used to control acess to a resource that has multiple instances

↳ S = no. of instances of a resource

SEMAPHORE DISADVANTAGE

↳ requires **BUSY WAITING**

↳ While a process is in critical section any other process that tries to enter critical section must loop continously to check if condition true

↳ wastes CPU cycles (that some other process could use productively)

* This type of semaphore also known a spinlock (as process 'spins' waiting for lock)

SOLUTION TO BUSY WAITING

↳ modify wait() and signal()

to block() → Places process (waiting state) in waiting queue

to wake up() → move process from waiting to ready queue

↳ instead of busy waiting, block itself

transfers control to CPU scheduler (which selects another process to execute)

hence no wastage of CPU time

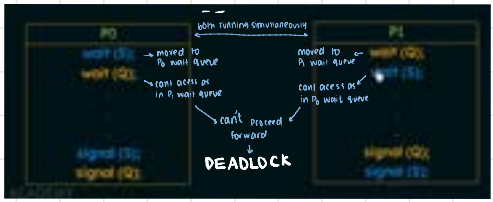
PROBLEM

↳ **Deadlock**

↳ set of blocked processes, each holding a resource and waiting to aquire a resource, held by another process

↳ **Starvation**

↳ when a process is postponed because it requires a resource to run, that is never allocated to this process



Indivisibility

When 1 process modifies semaphore

no other process can simultaneously modify that semaphore value

DEADLOCK VS STARVATION

↳ process stuck in circular waiting for the resources

↳ process waits for a resource indefinitely

deadlock implies starvation BUT starvation does not impy deadlock

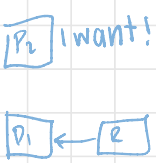
Priority Inversion

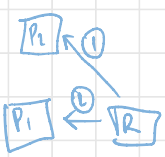
↳ scheduling problem when lower priority process holds a lock needed by a higher priority process

SOLUTION →

Priority Inheritance Protocol

↳ when a job blocks a higher priority job it ignores the current lower priority job executes higher priority job's critical section then releases lock and returns to the current lower priority job





CLASSICAL PROBLEMS OF SYNCHRONIZATION

1. Bounded buffer Problem

↳ shared buffer

↳ buffer of n slots, each capable of storing one unit of data

↳ Producer process → insert data in empty slot of buffer
↳ not insert data when buffer is full

↳ consumer process → remove data from filled slot of buffer
↳ not remove data when buffer is empty

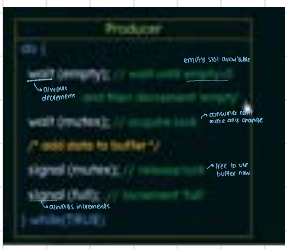
Solution using semaphores

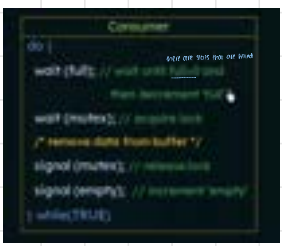
↳ n buffers → each can hold one item

↳ m(mutex) → binary semaphore → acquire lock / release lock

↳ empty → counting semaphore → initial value = no of slots in buffer since initially all slots are empty

↳ full → counting semaphore → initial value = 0





2. The Readers Writers Problem

↳ shared database

↳ Readers → only want to read

↳ Writers → want to read and write

↳ if a writer and a writer/reader access database simultaneously → PROBLEM

↳ give exclusive access to database → SOLUTION

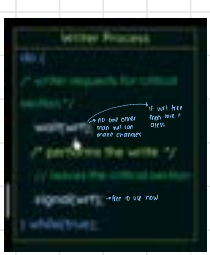
Solution using semaphores

↳ mutex → a semaphore → initialize to 1

↳ wrt → a semaphore → initialized to 1

↳ readcount → counts how many readers reading same data → initialized to 0

↳ whenever modified wait(mutex) to be called



→ used in OS resource allocation → Processes → resources

3. The Dinning-Philosophers Problem

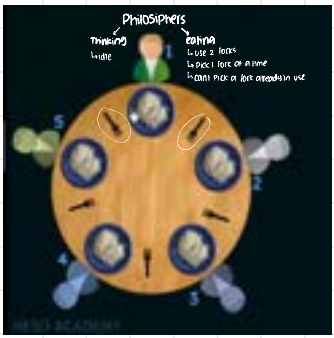
↳ 5 Philosophers, 5 forks

↳ Eating → use 2 forks for eating; Pick 1 fork at a time; can't pick a fork already in use } can't eat unless has 2 forks

↳ Thinking → idle

↳ when a philosophers eat he uses 2 forks

↳ NO 2 adjacent Philosophers try to eat at the same time (as forks are limited) → SOLUTION



SOLUTION USING SEMAPHORES

↳ fork[5] (as 5 forks) → a binary semaphore array (all elements initialized to 1; 1 → free, 0 → in use)

↳ grab fork → wait() → used on 2 forks

↳ release fork → signal()

↳ can lead to DEADLOCK

all 5 hungry and grab left fork

all elements of fork = 0

when grabbing right fork, they will all be delayed forever

↳ SOLUTIONS TO DEADLOCK

1. allow at most 4 philosophers (no of forks will remain 5)
2. allow to pick forks ONLY if both forks available
3. use asymmetric solution
   - ↳ an odd P first picks right fork then left fork
   - ↳ an even P first picks left fork then right fork

deadlock implies starvation

BUT

starvation does not imply deadlock

DEADLOCKS

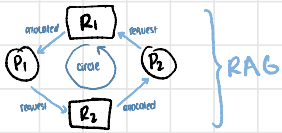
CHP 8

↳ set of blocked processes, each holding a resource and waiting to aquire a resource, held by another process

IF ANY 1 NOT THERE → NO DEADLOCK SITUATION

4 Deadlock necessary conditions

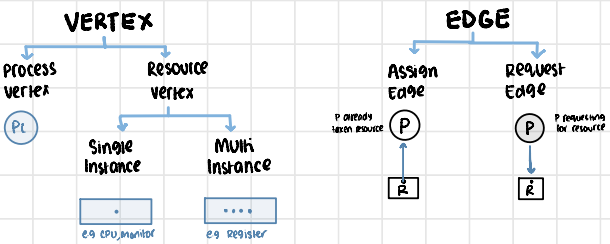
1. Mutual Exclusion — no resource can be used by more than 1 process at a time
2. NO Preemption — a process is holding a resource and second process comes and we preempt (recourse) it if this shouldn't occur
3. Hold and wait — P1 holding R1, waiting for R2; P2 holding R2, waiting for R1
4. Circular wait — loop must exist



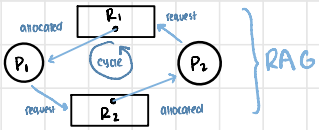
Resource Allocation Graph (RAG)

→ TO DETECT DEADLOCKS

↳ weather there is a deadlock or not, to represent we use RAG



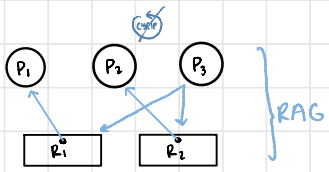
Single Instance RAG



	Allocate		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0
	0	0		

Availability = (0, 0) → means no resource available

DEAD LOCK SITUATION



only for this

Singular Instance + Circular wait

DEAD LOCK

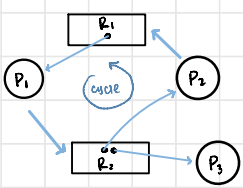
	Allocate		Request		
	R_1	R_2	R_1	R_2	
P_1	1	0	0	0	1 → execute as requests no resources ↳ will then terminate and release resources
P_2	0	1	0	0	2 → execute as requests no resources ↳ will then terminate and release resources
P_3	0	0	1	1	3 → execute as R_1, R_2 both available
	0	0			

Availability = (0, 0) → (1, 0) → (1, 1)

$P_1 \rightarrow P_2 \rightarrow P_3$

NO DEADLOCK

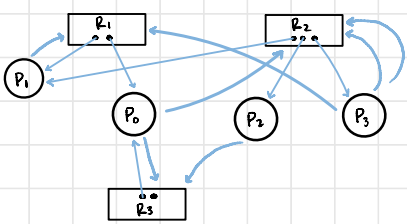
Multi Instance RAG



	Allocate		Request		
	R_1	R_2	R_1	R_2	
P_1	1	0	0	1	2 → needs 1 R_2 to execute
P_2	0	1	1	0	3 → as R_1 available
P_3	0	1	0	0	1 → as requests nothing executed first

Curr Availability = (0, 0) → (0, 1) → (1, 1) — resources freed; as P_2 had 2 instances and now 1 is free

$P_3 \rightarrow P_2 \rightarrow P_1$



	Allocated			Request			
	R_1	R_2	R_3	R_1	R_2	R_3	
P_0	1	0	1	0	1	1	1 → as R_2 R_3 free
P_1	1	1	0	1	0	0	3 → as R_1 free
P_2	0	1	0	0	0	1	4 → as it request R_3 which is free
P_3	0	1	0	1	2	0	4 → as R_1 and 2 R_2 free

Curr Availability (0, 0, 1) → (0, 1, 1) → (1, 1, 2) → (2, 2, 2) → (2, 3, 2)

$P_2 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$

Various methods to handle deadlock

1. Deadlock ignorance (ostrich method)

widely used

↳ just ignore deadlock as deadlock occurs rarely

so why write an algo...
which will reduce performance → as we want more speed
if occurs just restart system

2. Deadlock Prevention

↳ before deadlock occurs try to find solution
by removing all 4 or any 1 of the necessary deadlock conditions

FAIL ANY ONE

1. Mutual Exclusion

→ can be faked by CPU scheduling all types are sharable but not for printers

↳ make all resources sharable

2. No Preemption

↳ make preempted using a time quantum

3. Hold and wait

↳ give a process all its demanding before doing anything

↳ if it can't get all resource then get none

4. Circular wait

↳ order all resources
↳ assign each resource a priority
↳ make processes aquire resources in priority order

Resources
1 Printer
2 Scanner
3 CPU
4 Monitor

4. Deadlock Detection & Recovery

↳ allow system to enter deadlock state
↳ Detect Deadlock
↳ Recover Deadlock

↳ Kill the process
then check if still deadlock
if yes then repeat

↳ Resource Preemption
preempt a resource from a process

3. Deadlock Avoidance (Bankers Algo)

↳ when we give resources to a process, we check if it's safe or not

1. Process Initiation Denial → not optimal as assumes the worst

↳ process only started if max claims
of all curren + new processes can be met

2. Resource Allocation Denial

↳ aka Bankers Algorithm

1. safe state → no deadlocks
2. unsafe state → possibility of deadlock
3. Avoidance → ensures ensures system will never enter unsafe state

Bankers Algo

used by → Deadlock Avoidance, Deadlock Detection

3 diff type resources

A=10, B=5, C=7

freed allocated resources

10 5 7 → total

− 7 2 5 → allocated

3 3 2

Current Available

max need − allocation = Remaining Need

Process	Allocation (CPU A, memory B, printer C)			Max need			Current Available			Remaining Need			
Resources	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	0	1	0	7	5	3	3 (+2)	3 (0)	2 (0)	7	4	3	3 →
P_2	2	0	0	3	2	2	5 (+2)	3 (1)	2 (1)	1	2	2	1 →
P_3	3	0	2	9	0	2	7 (+0)	4 (1)	3 (0)	6	0	0	4 →
P_4	2	1	1	4	2	2	7 (+3)	5 (0)	3 (1)	2	1	1	2 →
P_5	0	0	2	5	3	3	10 (+0)	5 (0)	5 (2)	5	3	1	5 →
	7	2	5				10	5	7				

as max and remaining possible

↳ fill this section last using remaining need

safe sequence: $P_2 \rightarrow P_4 \rightarrow P_1 \rightarrow P_3 \rightarrow P_5$ → can have multiple sequences

Livelock

↳ similar to deadlock except

processes when realise they are blocked

they change their regard

- For example, consider two processes each waiting for a resource the other has but waiting in a non-blocking manner.
- When each knows their current attempt they release their held resource and sleep for 50 seconds, then they retrieve their original resource followed by trying to the resource the other process held; they left, then reacquired.
- Since both processes are trying to cope (not badly), this is a livelock.

Memory Management

CHP 9

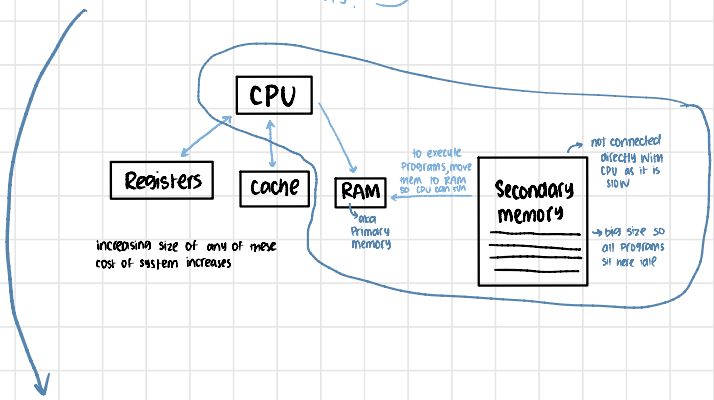
5.1

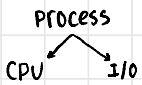
HOW DEGREE OF MULILPROGRAMING RELATES TO MEMORY MANAGEMENT

HOW TO MANAGE ALL RESOURCES IN EFFICIENT WAY BY

Method of managing Primary memory — as majority of memory is RAM

Why?





Degree of Multiprograming

↳ moving more than 1 process in RAM → so whenever CPU needs process to execute, it will have plenty process available so CPU not idle

↳ try to keep as many processes in RAM

↳ greater ↑RAM, greater no. of processes in RAM

greater degree, ↑greater CPU utilization — hence increase RAM

OS

ALLOCATION/DEALLOCATION

SECURITY

E.g

RAM 4mb
Process 4mb

$\frac{4}{4}$ = 1 process

RAM: P_1

time for I/O operation
k = 70%.

CPU Utilization = $(1-k)$ → 30%.

RAM 8mb
Process 4mb

$\frac{8}{4}$ = 2 Processes

RAM: P_1, P_2

time for I/O operation
k = 70%.

CPU Utilization = $(1-k^2)$ (2 processes) → 76%.

RAM 16mb
Process 4mb

$\frac{16}{4}$ = 4 Processes

RAM: P_1, P_2, P_3, P_4

time for I/O operation
k = 70%.

CPU Utilization = $(1-k^4)$ (4 processes) → 94%.

Address Binding

the binding of instructions and data to memory address

1. compile time

↳ if you know at compile time
where the process will reside in memory
then absolute code can be generated

↳ if starting location changes,
then necessary to recompile this code

2. load time

↳ if you don't know at compile time
where the process will reside in memory
then relocable code must be generated

↳ if starting location changes,
then only reload user code

↳ final binding delayed until load time

3. Execution time

↳ If process can be moved during execution
from one memory segment to another
then binding delayed until run time

Static linking – system libraries and program code combined by the loader into the binary program image

Dynamic linking –linking postponed until execution time

Small piece of code, **stub**, used to locate the appropriate memory-resident library routine

Stub replaces itself with the address of the routine, and executes the routine

Dynamic linking is particularly useful for libraries

Static Linking vs Dynamic Linking

Windows:
.lib vs .dll files

Linux:
.a vs .so files

Logical address (LA)

↳ aka virtual address

↳ always generated by CPU

Physical address (PA)

↳ aka absolute address

↳ address seen by m/m

Protection of Process Address space

1. make sure each process has a seperate memory space
   as protects processes from each other
2. Protection is provided by these two registers
   ↳ base register: holds smallest Physical memory address
   ↳ limit register: specifies size of range
3. Only OS can load base and limit registers
   ↳ prevents user programs from changing register contents

Relocation of Address

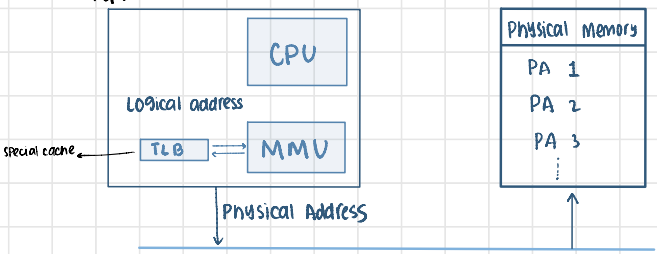
* base register aka relocation register

↳ value of relocation register is added
to every address generated by a user process

Memory management unit (MMU)

↳ converts LA into PA

↳ Page Table



↳ keep as many no of processes in RAM → process in RAM are in ready state

Memory Management Techniques

NO SPAN
CONSECUTIVE

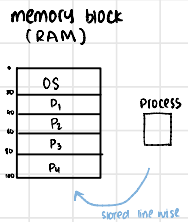
CONTINOUS

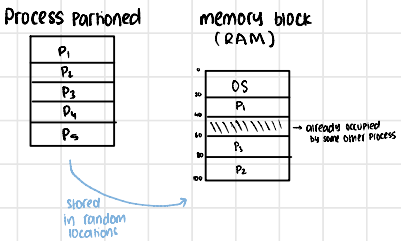
Static
- fixed partition
- variable partition (Dynamic)

SPAN
NOT CONSECUTIVE

NON CONTINOUS

- Paging
- Multilevel Paging
- Invalid Paging
- Segmentation
- Segmented Paging





NO SPAN → keep some portion of a process in one partion and the other in another

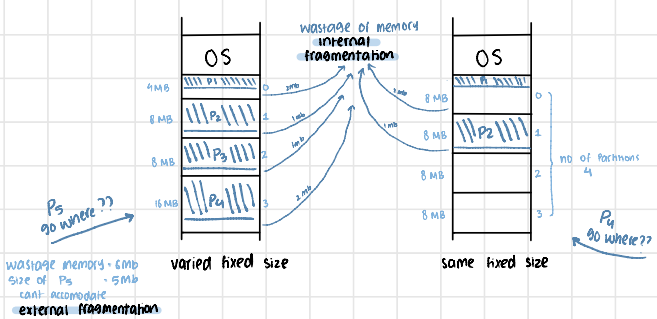
CONTINOUS MEMORY ALLOCATION

1. FIXED PARTITIONING (static partition) → EASY TO IMPLEMENT

- ↳ no. of partitions are fixed
- ↳ size of partition may or may not vary
- ↳ stored line wise

LIMITATIONS

1. Internal Fragmentation → wastage of memory
2. Limit in Process Size → eg cant have process greater than 16MB as it max size
3. Limitation on Degree of Multiprogramming → eg cant accomodate more than 4 processes as no of partitions are 4
4. External Fragmentation → sum of all space = available space but still not able to accomodate because of continous method



Processes	
P_1	2 MB
P_2	7 MB
P_3	7MB
P_4	14MB
P_5	5MB

5.4

2. VARIABLE PARTITIONING (Dynamic Partition)

PROS

↳ NO internal fragmentation

↳ NO Limit on Degree of Multiprogramming

↳ NO Limit in Process size

↳ stored line wise

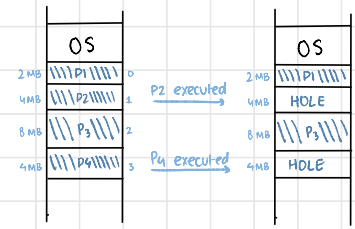
LIMITATIONS

1. External Fragmentation → sum of all space = available space but still not able to accomodate because of continous method

2. Allocation/Deallocation → complex as memory allocated dynamically at runtime holes

Processes

P_1	2 Mb
P_2	4 Mb
P_3	8 Mb
P_4	4 Mb



now P_6 = 8Mb comes
where to put it?

External Fragmentation

SOLUTION

COMPACTION →

↳ removes external fragmentation

↳ moves all holes together

CONS

↳ Will have to stop all processes to move

↳ will have to move by copying which will take a lot of time

5.5

Dynamic
4 ALGOs to allocate processes in hole

First-Fit (most convinient)

↳ allocate the first hole that is big enough

↳ always search from top

PRO

fast

CON

Next-Fit

↳ same as first fit but start search from last allocated hole

PROS

faster than first fit as searches from last allocation

CONS

Best-Fit

allocate the smallest hole that is big enough

PROS

less internal fragmentation

CONS

creates very tiny hole → so further processes can't fit in it

SLOW → as searches entire list

Worst-Fit

allocate the largest hole that is big enough

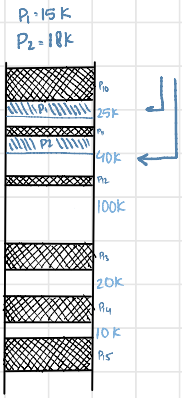
PROS

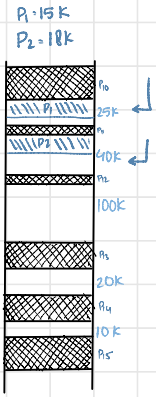
creates big holes → so further processes can fit in it

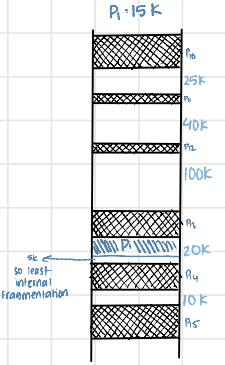
CONS

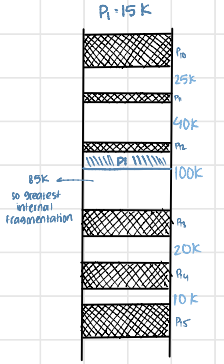
great internal fragmentation

SLOW → as searches entire list









58

SPAN — keep some portion of a process in one partition and the other in another

NON CONTINOUS MEMORY ALLOCATION

↳ holes created dynamically (keep changing in size number)

↳ dividing processes in run time

TIME CONSUMING

SOLUTION

Paging

↳ removes external fragmentation

↳ divide processes into pages

↳ each and every process has a PT

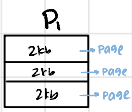
↳ bring some of those pages into M/M (which is divided into frames)

↳ PT will be in M/M

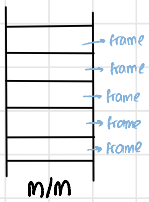
↳ when CPU demands a page/process

↳ search PT to find page frame

Process divided before hand in secondry memory



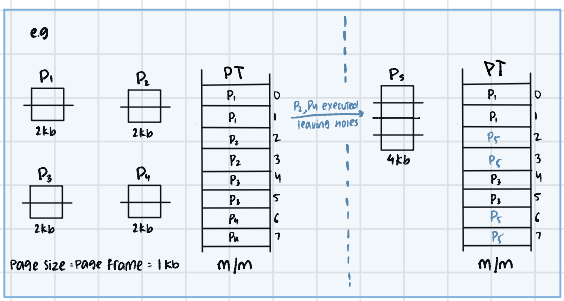
main memory divided before hand



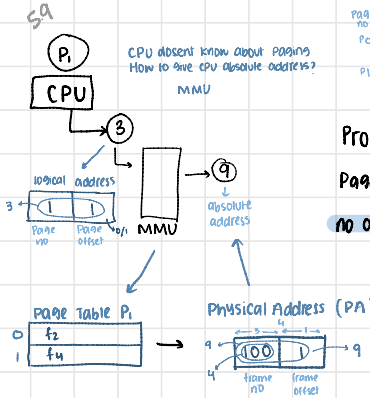
Page Size = frame size

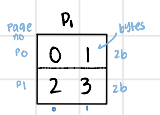
as pages go inside frame so perfect fit

e.g



59

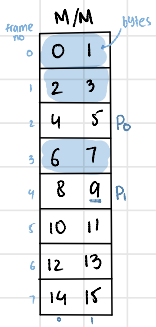




Process size = 4b

Page Size = 2b

no of pages = $\frac{4}{2}$ = 2



M/M size = 16b

frame size = 2b

no of frames = $\frac{16}{2}$ = 8 frames

5.10

8) LAS = 4GB = $2^2 \times 2^{30} = 2^{32}$ → LA (represents size of process)

PAS = 64MB = $2^6 \times 2^{20} = 2^{26}$ → PA

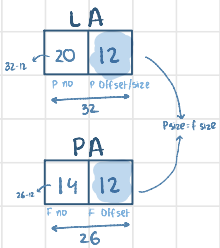
Page size = 4kB = $2^2 \times 2^{10} = 2^{12}$

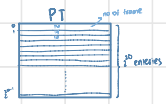
a) no of pages: 32 − 12 = 20 = 2^{20}

b) no of frames: 26 − 12 = 14 = 2^{14}

c) no of addresses in page table = 2^{20} as no of pages = 2^{20}

d) size of page table = $2^{20} \times 14$ =





Page size = frame size (as pages go inside frames)

no of enteries in PT = no of pages

PT size = no of enteries x size of entry (no of frames bits)

word = Byte (unless mentioned)

$2^{5\text{bit}} = 32$ (5 bits)

B = 8 bits → so no need to convert

K = 2^{10} bits

M = 2^{20} bits

G = 2^{30} bits

5.11

8) LA = 7 bits

PA = 6 bits

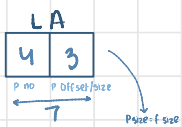
Page size = 8 words = 2^3 = 3 bits

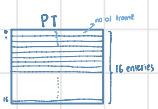
a) no of pages: 7 − 3 = 4 = 2^4 = 16

b) no of frames: 6 − 3 = 3 = 2^3 = 8

c) no of enteries in PT: 16

d) size of page table: 16 x 3 =





MEMORY MANAGEMENT UNIT (MMU)

5.12

↳ Page Tables

↳ maps logical address to physical address

↳ every process has it's own page table

- **Page fault** ↳ does page exist in spot? ↳ as pages move in and out
- **Used by OS** ↳ Read ↳ Write ↳ Execute (Permissions)
- **Swap pages in/out** ↳ LRU
- **Cache Page** ↳ when a data used multiple times we cache it ↳ take min time
- **Page modified by User** ↳ Update modified page in harddisk

Frame no	valid (1) Invalid (0)	Protection R W X	Reference (first time in memory (0/1) been in memory before)	Caching enable/disable	Dirty / Modified

mandatory field ← → optional fields

5.13

2 level Paging / Hierachical Paging

↳ if page table size > page frame size

↳ so cant fit

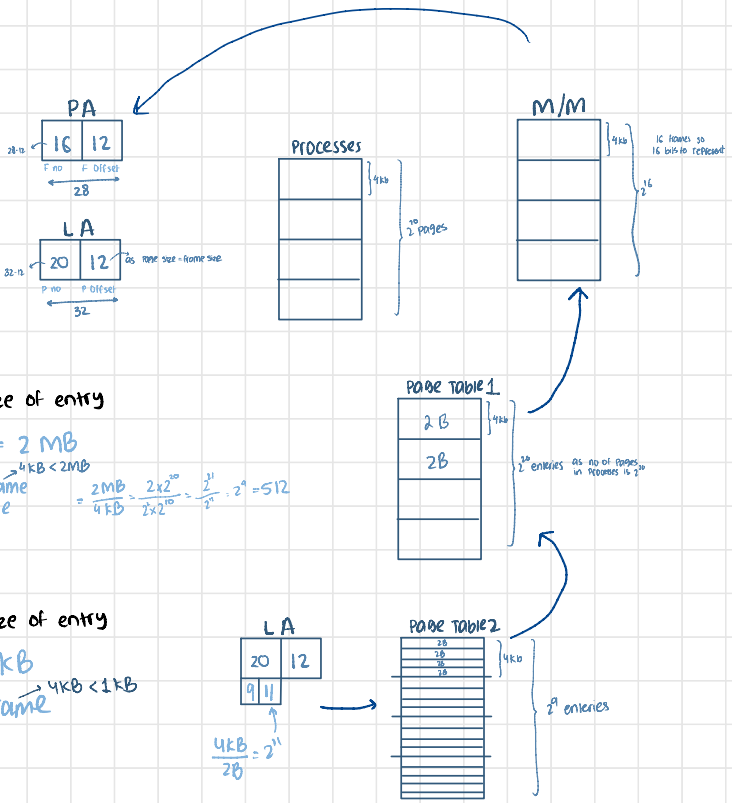
↳ so divide page table to smaller pages

Q) PA/MM = 256 MB → 28 bits

LA = 4GB → 32 bits

Frame size = 4 kB → 4K = 12 bits

Page Table Entry = 2B



Page table size = no of enteries x size of entry

$= 2^{20} \times 2B = 2MB$

↳ can not fit in Frame ($4kB < 2MB$) so divide page table further into pages

$= \frac{2MB}{4KB} = \frac{2 \times 2^{20}}{2 \times 2^{10}} = \frac{2^{21}}{2^{11}} = 2^9 = 512$

Page table size = no of enteries x size of entry

$= 2^9 \times 2B = 1kB$

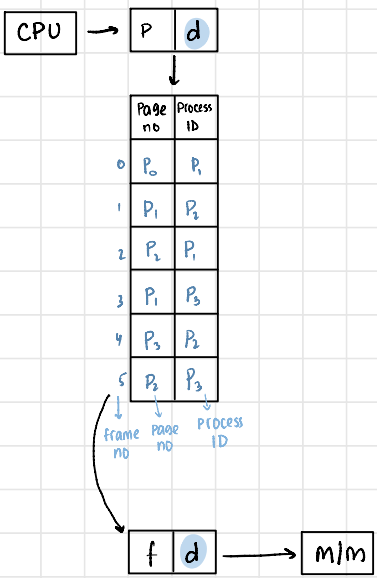
↳ can fit in frame ($4kB < 1kB$)

5.14

Inverted Paging

1. Instead of each and every process has a PT, (as m/m is limited, no of frame is limited)
   all processes have only 1 PT ⇒ Global Page Table
   no of enteries in PT = no of frames

CON
Searching is linear hence searching time high

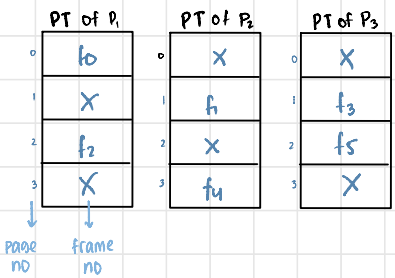


Normal Paging

1. each and every process has a PT
2. PT will be in m/m

. no of enteries in PT = no of pages

CON
memory is limited



5.15

Q) VAS = 32 bits, Page size = 4 KB ($= 2^2 \times 2^{10} = 2^{12}$), System RAM = 128 KB ($2^7 \times 2^{10} = 2^{17}$)

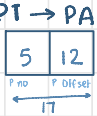
a) what is the ratio of VT and Inverted PT size if, table entry size = 4B

VA → LA

no of VT enteries = 2^{20}

VT size = $2^{20} \times 4$

IPT → PA

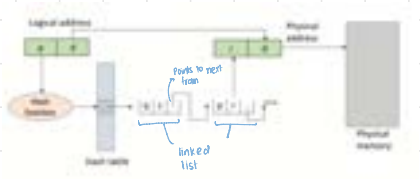


no of IPT enteries = 2^5

IPT size = $2^5 \times 4$

$$\frac{2^{20} \times 4}{2^5 \times 4} = \frac{2^{15}}{1}$$

Hasned Page Table



Virtual Memory

516

Page faults

↳ bring some page of each process in RAM to increase DOM

↳ now some portion of all processes in RAM

↳ If CPU ask for a page that isn't in RAM → Page fault → (to fix) Page fault service time

↳ so Os gets busy in servicing page fault

↳ Performance degrade

↳ brings page from HDD to MM

↳ take a lot of time

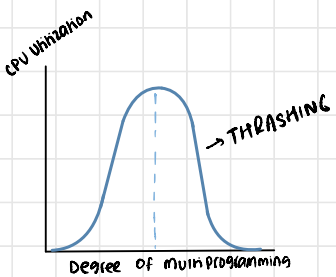
Thrashing

↳ Page faults > Page hit

↳ Page fault service time → OS busy

↳ CPU remains idle for longer

↳ CPU utilization decreases



Degree of multiprogramming

(DOM)

↳ no of processes in RAM increases

CPU utilization increases

If CPU utilization too low

↳ increase degree of multiprogramming

To avoid Thrashing

1. Increase size of M/M → difficult to achive
2. Long term scheduler → don't take DOM to max level, decrease it a bit

5.17

Segmentation

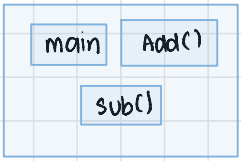
↳ Process divided into parts/segments (not fixed size)

↳ then put in m/m

PRO

↳ works from user point of view

↳ makes segments according to the code



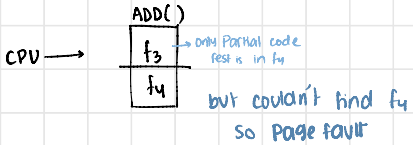
Paging

↳ Process divided into fixed sized pages

↳ then put in m/m

CON

↳ w/o knowing the whats the code we just divide in further pages

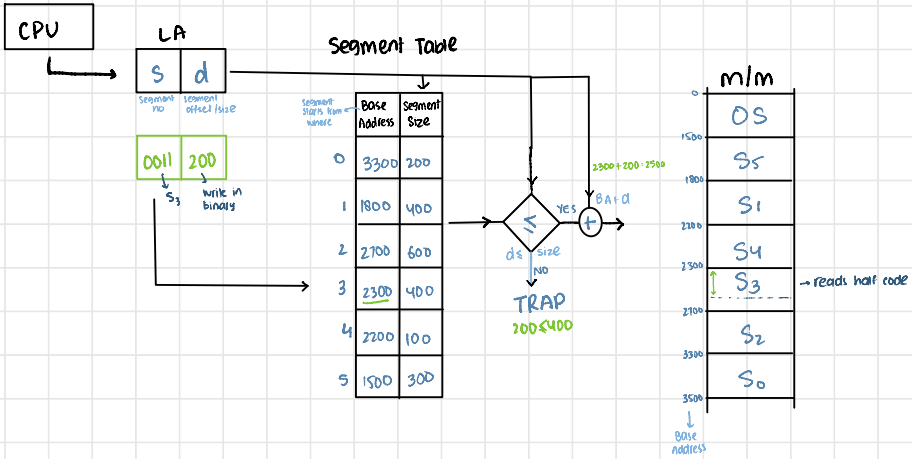


↳ CPU generates logical address

↳ will then asses what code it needs and till where

↳ convert to physical address

LA → MMU → PA



S.10

Virtual memory

↳ we currently use

↳ gives illusion that a process whose size is larger than the size of m/m can be executed

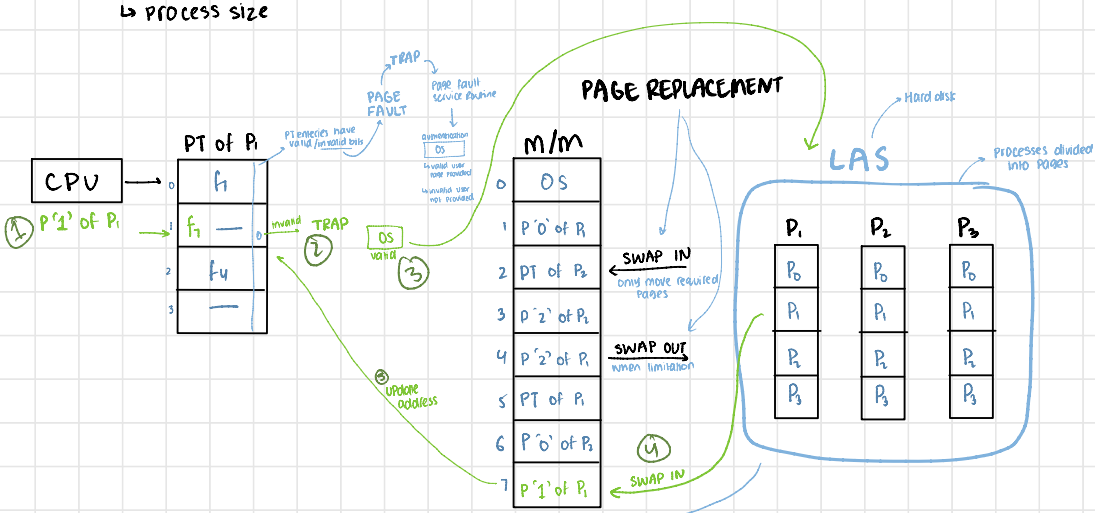
↳ no limitation on

↳ no. of processes

↳ process size

locality of reference

↳ when we bring a page in m/m we bring related pages along with it to



↳ we dont bring all Processes in m/m
↳ only those pages that are currently required
↳ so max parts of processes can be kept in m/m

Demand Paging Steps

User
1. CPU generates
2. Page invalid → Page fault
↳ Trap generated
↳ Page fault service routine

Os
↳ 3. Os authenticates
↳ Os finds page in LAS/Harddisk
4. if frame free swap in the page to m/m
↳ not free uses Page Replacement algorithm
5. Updates address in PT

user
6. CPU given control

Demand paging

PROS

↳ less I/O needed

↳ less memory needed

↳ faster response

↳ more users

(EMAT) effective memory access time = (Probability of Page fault (p)) (Page fault service time (mostly in milliseconds (m sec))) + (no Page fault (1 − p)) (m/m access time (mostly in nano seconds (n sec)))

Page fault service routine

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal, and determine the location of the page in secondary storage.
5. Issue a read from the storage to a free frame:
   a. Wait in a queue until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU core to some other process.
7. Receive an interrupt from the storage I/O subsystem (I/O completed).
8. Save the registers and process state for the other process (if step 6 is executed).
9. Determine that the interrupt was from the secondary storage device.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU core to be allocated to this process again.
12. Restore the registers, process state, and new page table, and then resume the interrupted instruction.

Page fault service routine with page replacement

1. Find the location of the desired page on secondary storage.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c. Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

↳ if PT is normal sized

CPU → LA → PT --frame no→ M/M
(n for PT, as PT is inside m/m; n for M/M)

n → time taken to acess m/m

total time = 2n → ideal condition

↳ if PT too big (divided into multiple levels) then

total time will increase

SOLUTION ?

5.20

Translation Lookaside Buffer (TLB)

↳ to get faster memory use TLB / cache / buffer

as TLB is faster than RAM, so store PT in TLB

> if no page fault
>
> EMAT = HIT(TLB + m/m) + MISS(TLB + PT + m/m)   (PT: same as m/m)

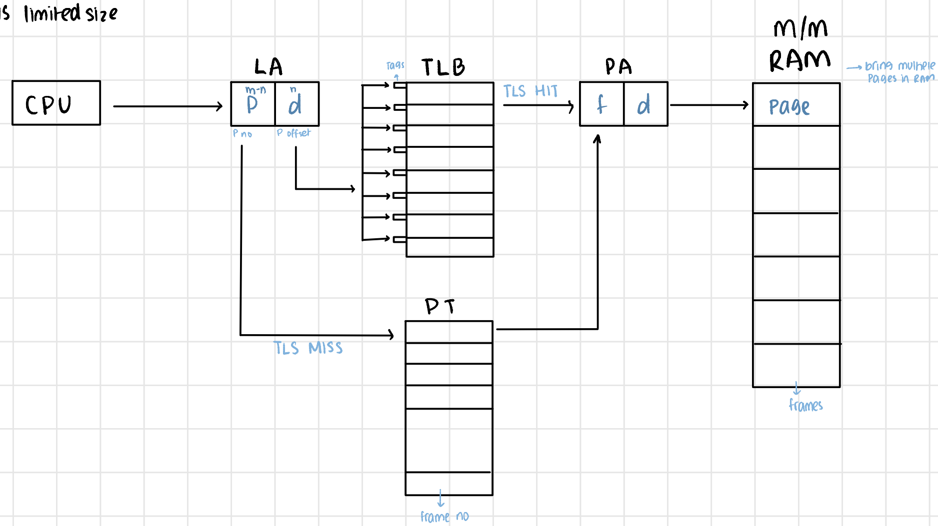
↳ used when Hits > Miss

↳ TLB HIT : found corresponding frame for page   TLB --frame no ✓→ M/M

↳ TLB MISS : then normal paging procedure   TLB --✗→ PT --frame no→ M/M

CON

↳ TLB has limited size



5.21

Q) Paging scheme TLB

TLB acess time : 10 ns , m/m acess time : 50 ns

what is EMAT (in ns) if , Hit = 90% and no page fault

EMAT = HIT(TLB + m/m) + MISS(TLB + PT + m/m)

$= 0.9(10 + 50) + 0.1(10 + 50 + 50) = 65$ ns

5.22

Page Replacement

↳ in Virtual memory concept

↳ when m/m filled and need to bring new page in m/m

↳ swap in, swap out

HIT ratio = $\frac{\text{no. of HITS}}{\text{no. of references}} \times 100$

miss ratio = $\frac{\text{no. of miss}}{\text{no. of references}} \times 100$

1. FIRST IN FIRST OUT (FIFO)

↳ when page fault occurs, replace page which came in first

Q) Reference string = 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0

f_3			1	1	1	1	0	0	0	3	3	3	3	2	2
f_2		0	0	0	0	3	3	3	2	2	2	2	1	1	1
f_1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0
	*	*	*	*	HIT	*	*	*	*	*	*	HIT	*	*	HIT
	↓ MISS			as no space replace 7 as it came in first											

Page Hit = 3 — Hit ratio = $\frac{3}{15} \times 100 =$

Page fault = 12 — Miss ratio = $\frac{12}{15} \times 100 =$

5.23

Q) Ref → 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

USING 3 FRAMES

f_3			3	3	3	2	2	2	2	2	4	4
f_2		2	2	2	1	1	1	1	1	3	3	3
f_1	1	1	1	4	4	4	5	5	5	5	5	5
	*	*	*	*	*	*	*	HIT	HIT	*	*	HIT

HITS = 3

MISS = 9

USING 4 FRAMES

f_4				4	4	4	4	4	4	3	3	3
f_3			3	3	3	3	3	3	2	2	2	2
f_2		2	2	2	2	2	2	1	1	1	1	5
f_1	1	1	1	1	1	1	5	5	5	5	4	4
	*	*	*	*	HIT	HIT	*	*	*	*	*	*

HITS = 2

MISS = 10

→ **Belady's Anomaly**

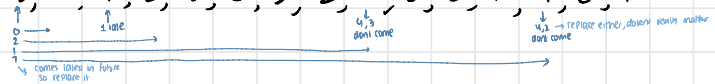
↳ increasing no of frames increased no of page faults

524

2. OPTIMAL PAGE REPLACEMENT

↳ when page fault occurs, replace page whos demand is very late in future

D) Ref → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



f_4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
f_3			1	1	1	1	✗	4	4	4	4	4	4	1	1	1	1	1	1	1
f_2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f_1	7	7	7	7	✗	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	*	*	*	*	HIT	*	HIT	*	HIT	HIT	HIT	HIT	HIT	*	HIT	HIT	HIT	*	HIT	HIT

HIT = 12
MISS = 8

525

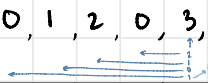
3. LEAST RECENTLY USED (LRU)

→ COUNTING BASED PAGE REPLACEMENT ALGO

↳ when page fault occurs, replace page which is least recently used in the past

speed lesser than FIFO as searching

D) Ref → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



f_4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f_3			1	1	1	1	✗	4	4	4	4	4	4	1	1	1	1	1	1	1
f_2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f_1	7	7	7	7	✗	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
	*	*	*	*	HIT	*	HIT	*	HIT	HIT	HIT	HIT	HIT	*	HIT	HIT	HIT	*	HIT	HIT

HIT = 12
MISS = 8

5.16

4. MOST RECENTLY USED (MRU)

→ COUNTING BASED PAGE REPLACEMENT ALGO

↳ when page fault occurs, replace page which is most recently used in the past

0) Ref → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

(recent ↑)

f_4				2	2	2	2	2	2	3	0	3	2	2	2	0	0	0	0	0
f_3			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f_2		0	0	0	0	3	0	4	4	4	4	4	4	4	4	4	4	4	4	4
f_1	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	*	*	*	*	HIT	*	*	*	HIT	*	*	*	*	HIT	HIT	*	HIT	HIT	HIT	HIT

HIT = 8
MISS = 12

5. Second chance Algorithm

↳ uses reference bit

↳ FIFO

Ref	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6		
			3_0	3_0	3_0	1_0	1_0	1_0	2_0	2_0	2_1	2_1	2_0	2_0	3_0	3_0	3_0	3_0	3_1	3_0		
		2_0	2_0	2_0	2_1	2_0	2_0	6_0	6_0	6_0	6_0	3_0	3_0	6_0	6_0	6_0	1_0	1_0	1_0	6_0		
	1_0	1_0	1_0	4_0	4_0	4_0	5_0	5_0	5_0	1_0	1_0	1_0	7_0	7_0	7_0	2_0	2_0	2_1	2_1	2_0		
					H						H							H	H			

10.4.5.3 Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit (described in Section 10.4.1) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified — best page to replace
2. (0, 1) not recently used but modified — not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean — probably will be used again soon
4. (1, 1) recently used and modified — probably will be used again soon, and the page will be need to be written out to secondary storage before it can be replaced

Enhance Second Chance

- The major difference between ESC algorithm and the simpler SC algorithm is that here we give preference to those pages that have been modified in order to reduce the number of I/Os required.

COPY ON WRITE (COW)

↳ instead share pages

↳ Pages that can be modified COW marked

↳ if a P/C wants to write on that shared page

A copy of that page is made as it is modified

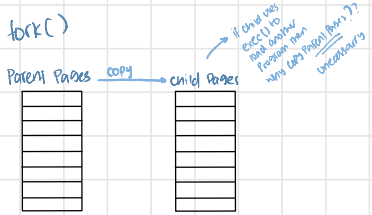
acess to OG page lost

↳ P/C share pages that are unmodified

PRO

↳ efficient Process creation → all pages aren't duplicated only the modified ones are

fork()



MODify/Dirty bit

↳ modify bit with each frame

↳ modify bit = 1

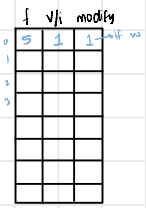
↳ if changes are done

↳ Update the changes in HD/ write back in HD

↳ modify bit = 0

↳ no changes are done

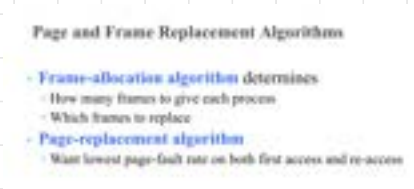
↳ no need to update as copy already in HD



PROS

↳ reduces Page fault time

↳ reduce Page transfers overhead → as only modified Pages are written back



Allocation of Frames

- Each process needs *minimum* number of frames
- *Maximum* of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes.
- Keep some as free frame buffer pool
  - Equal Allocation=Number of frames / number of processes
  - EA= 100/ 5 = 20 frames /process

Propotional Allocation

- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i
$S = \sum s_i$
m = total number of frames
a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$
$s_1 = 10$
$s_2 = 127$
$a_1 = \frac{10}{137} \times 64 \approx 5$
$a_2 = \frac{127}{137} \times 64 \approx 59$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating lgroups
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
- Operating system can given direct access to the disk, getting out of the way of the applications

turn around time = Completion − Arrival time

Waiting time = Total waiting time − ms process executed − Arrival time

↓ Preemptive

Waiting time = turn around time − burst time

↓ non preemptive

AMDAHL'S LAW

↳ speed up in latency of a task execution

$$\text{Speed UP} \leq \frac{1}{S + \frac{S-1}{N}}$$

S: serial, N: no of cores

Page size = frame size (as pages so inside frames)

no of enteries in PT = no of pages

PT size = no of enteries × size of entry (no of frames bits)

word = Byte (unless mentioned)

$2^{3 \text{ bits}} = 32$ (3 bits)

B = 8 bits → so no need to convert

$K = 2^{10 \text{ bits}}$

$M = 2^{20 \text{ bits}}$

$G = 2^{30 \text{ bits}}$

If no page fault

$$EMAT = HIT(TLB + m/m) + MISS(TLB + PT + m/m)$$

(PT: same as m/m)

(EMAT)

$$\text{effective memory acess time} = (P)(\text{Page fault service time}) + (1-P)(m/m \text{ acess time})$$

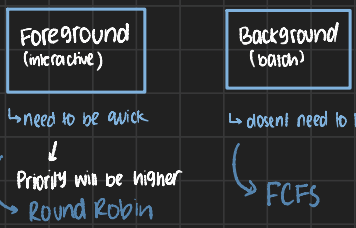
P: probability of page fault; Page fault service time: mostly in milliseconds (m sec); (1−P): no page fault; m/m acess time: mostly in nano seconds (n sec)

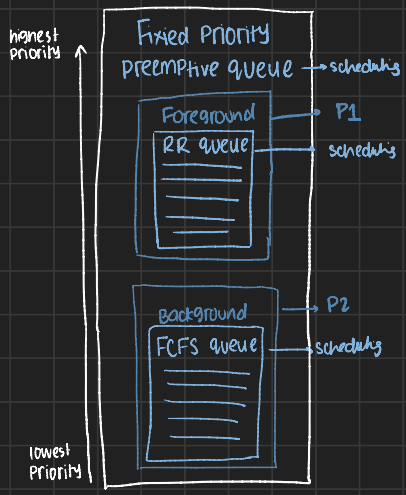
NO TALKING CLASS SAY BAM!R

Multilevel Queue Scheduling

Scheduling algorithms for situations in which processes are classified in different groups

E.g





- Partions ready queue into several seperate queues
- scheduling takes place within and among these queues
- Each process permamenlay assigned to 1 queue
  based on size, priority, type
- Each queue has its own scheduling algorithm (FCFS, SJF, RR...)

Multilevel Feedback Queue Scheduling

- allows processes to move between queues
- seperate processes according to their CPU bursts
- If a process uses too much CPU time
  then moved to lower priority queue (So everybody gets a chance)
- I/O and interactive processes are in higher priority queues
  as they need quick responses
- A process in lower priority queue for too long
  may be moved to higher priority queue (prevents starvation)

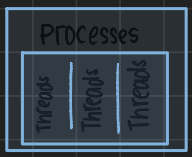
Parameters

- no. of queues
- scheduling algo for each queue
- method to upgrade process to higher priority
- method to demote process to higher priority
- method to determine queue when process demoted/upgraded

Process

↳ A program in execution

↳ can have 1 or more threads



Threads

↳ unit of execution within a process

↳ It compromises of

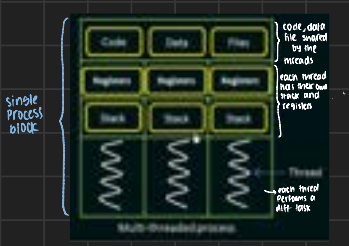
1. Thread ID
2. Program counter
3. Register set
4. Stack

↳ shares code section, data section, os resources with other threads of same processes

S.No.	Process	Thread
1.	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2.	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3.	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4.	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5.	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6.	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

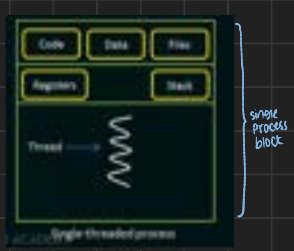
Multi Thread

↳ Performs multiple tasks at a time



Single Thread

↳ Performs one task at a time



MULTI-THREADING BENEFITS

↳ Responsivness: may continue execution if process is blocked/performing lengthy operation

dedicated threads for handling user events

↳ Resource Sharing: threads share memory and resources of a process → allows diff threads activity within same address space

→ easier than shared memory/message passing

↳ Economy: cheaper than process creation as threads share resources

allocating memory/resources is costly

thread switching has lower overhead than context switching

↳ Scalibility: utilization of multiple cores for parallel execution

increases concurrency