DEADLOCKS

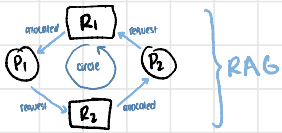
CHP 8

↳ 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

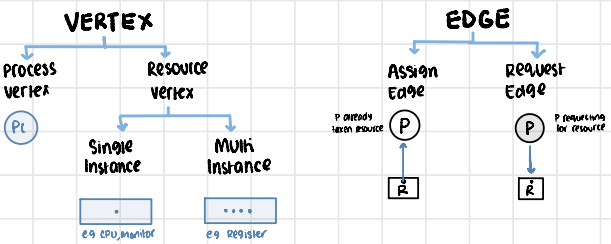
1. Mutual Exclusion — no resource can be used by more than 1 process at a time
2. NO Preemption — a process is holding a resource and second process comes and we preempt (recourse to it, this shouldn't occur
3. Hold and wait — P1 holding R1, waiting for R2; P2 holding R2, waiting for R1
4. Circular wait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



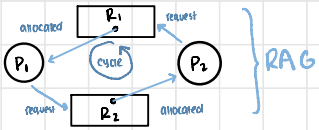
Resource Allocation Graph (RAG)

→ TO DETECT DEADLOCKS

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



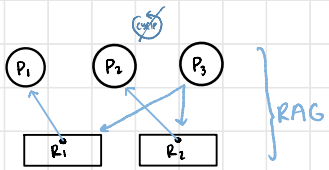
Single Instance RAG



	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
P_2	0	1	0	0
P_3	0	0	1	1
	0	0		

1 → execute as requests no resources
↳ will then terminate and release resources

2 → execute as requests no resources
↳ will then terminate and release resources

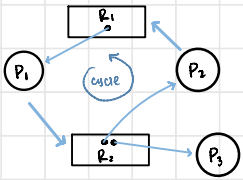
3 → execute as R_1, R_2 both available

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
P_2	0	1	1	0
P_3	0	1	0	0

1 → needs 1 R_2 to execute

2 → as R_2 available

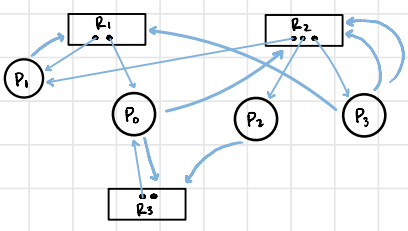
3 → as requests nothing executed first

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

resources freed

as R_2 has 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
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

1 → as R_2 R_3 free

2 → as R_1 free

3 → as it request R_3 which is free

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 solved by CPU scheduling all types are shareable but not for printers

↳ make all resources shareable

2. No Preemption

↳ make preempted using a time quantum

- To attack the no preemption condition:
  - If a process asks for a resource not currently available, block it but also take away all of its other resources
  - Add the preempted resources to the list of resource the blocked process is waiting for

3. Hold and wait

↳ give a process all its demanding before doing anything

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

- Each philosopher tries to get both chopsticks, but if only one is available, put it down and try again later

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 Keyboard

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

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

1. **Selecting a victim** – minimize cost
2. **Rollback** – return to some safe state, restart process for that state
3. **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

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

2 0 0

Current Available

max need − allocation

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 → as max and remaining possible
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				

↳ 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 learns they cannot continue they release their held resource and sleep for 30 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 (just badly), this is a livelock.

Memory Management

CHP 9

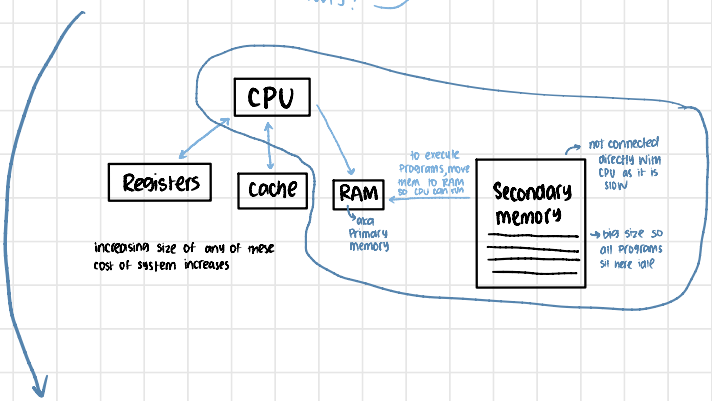
5.1

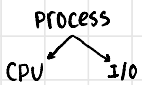
HOW DEGREE OF MULILPROGRAMING RELATES TO MEMORY MANAGEMENT

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

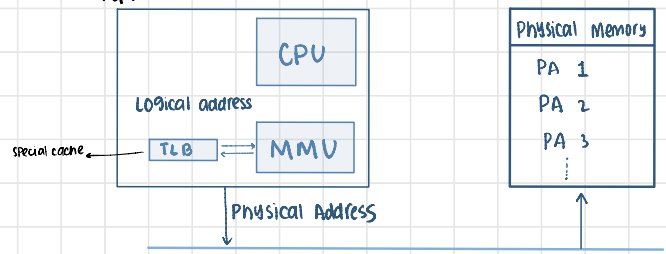
* base register aka relocation register

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

Memory management unit (MMU)

↳ converts LA into PA

↳ Page Table



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

Memory Management Techniques

NO SPAN
CONSECUTIVE

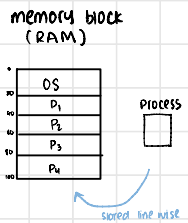
CONTINOUS

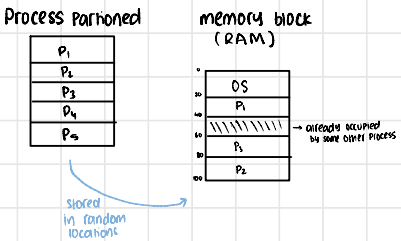
- Static
  - fixed partition
  - variable partition
- Dynamic

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

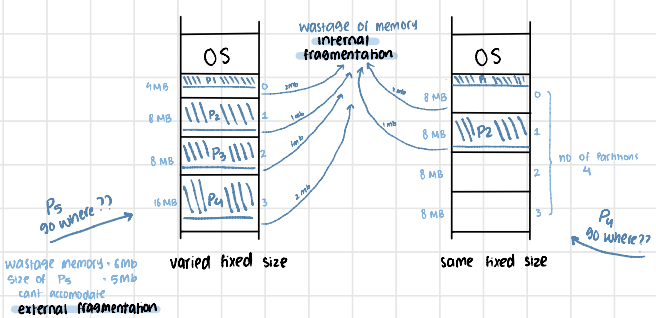
5.3

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 can't have process greater than 16MB as it max size
3. Limitation on Degree of Multiprogramming → e.g can't 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

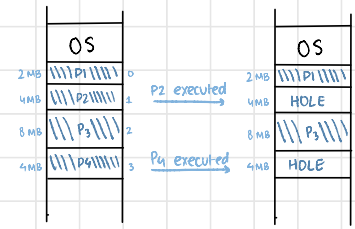
LIMITATIONS

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 cant fit in it

SLOW → as searches entire list

Worst-Fit

allocate the largest hole that is big enough

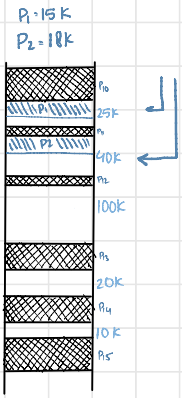
PROS

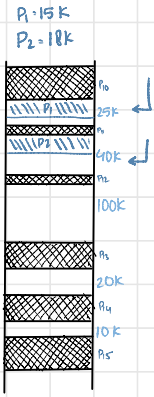
creates big holes → so further processes can fit in it

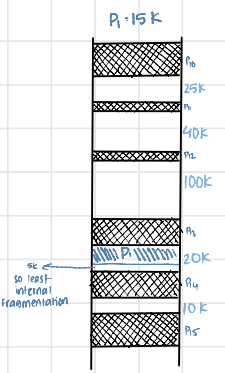
CONS

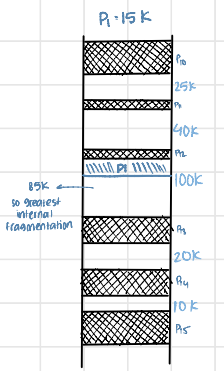
great internal fragmentation

SLOW → as searches entire list









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

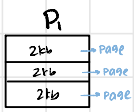
↳ bring some of those pages into M/M (which is divided into frames)

↳ PT will be in M/M

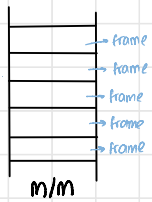
↳ when CPU demands a page/process

↳ search PT to find page frame

Process divided before hand in secondry memory

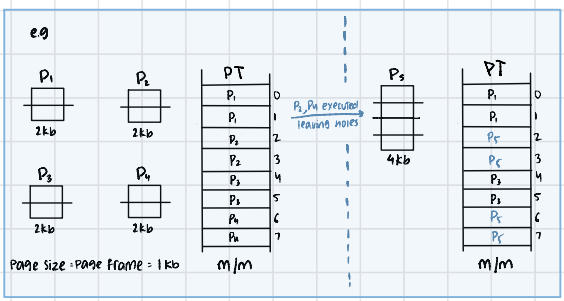


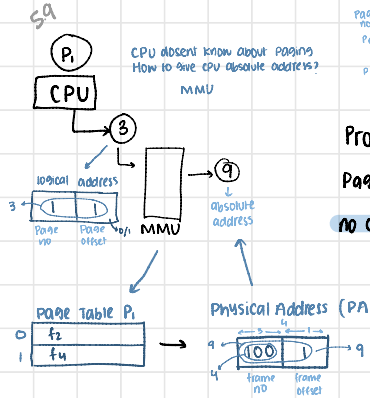
main memory divided before hand

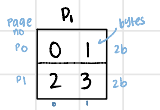


Page size = frame size

as pages go inside frame so perfect fit



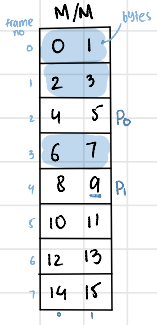




Process size = 4b

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

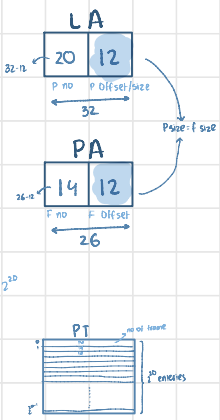
Page size = 4kB = $2^2 \times 2^{10} = 2^{12}$

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

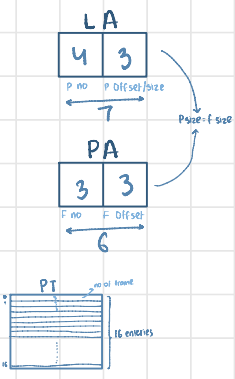
Page size = 8 words = 2^3 = 3 bits

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

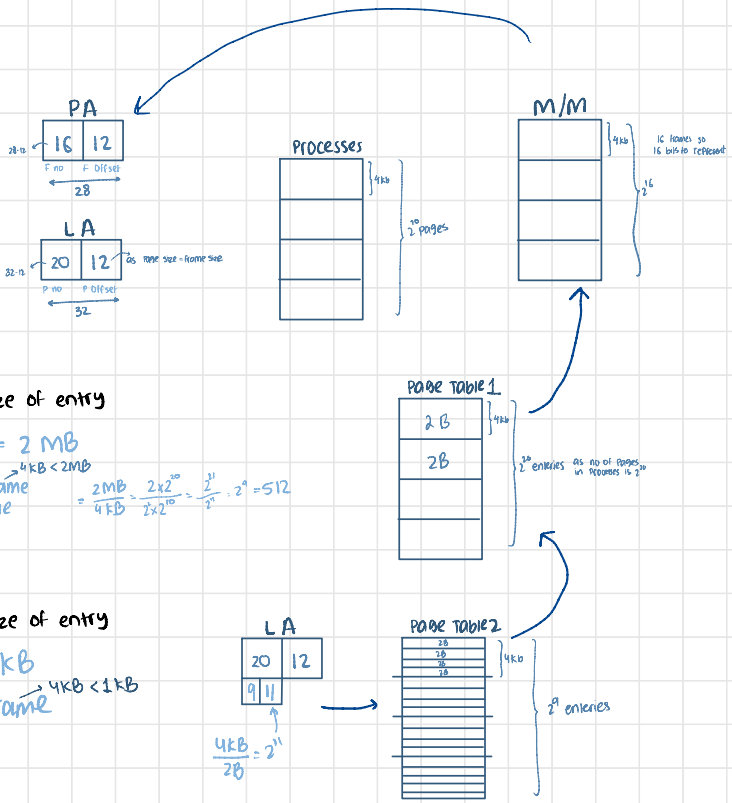
↳ so divide page table to smaller pages

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

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

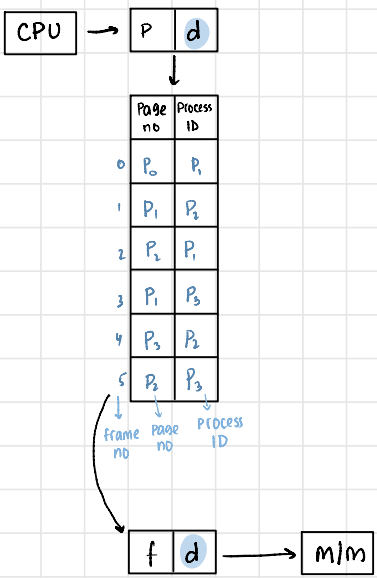
↳ can fit in frame ($4kB < 1kB$)

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

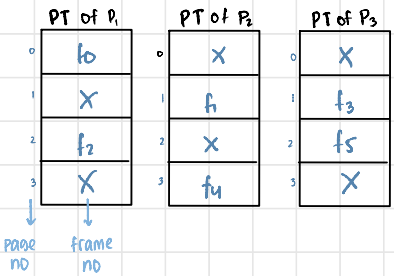


Normal Paging

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

20	12
P no	P. Offset

32

no of VT enteries = 2^{20}

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

IPT → PA

5	12
P no	P. Offset

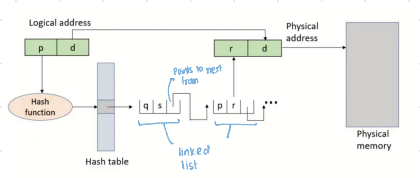
17

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

↳ brings page from HDD to MM

↳ take a lot of time

↳ So Os gets busy in servicing page fault

↳ Performance degrade

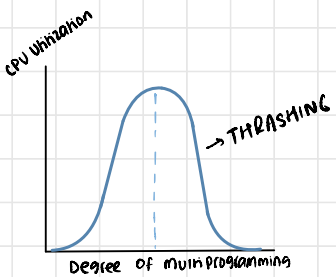
Thrashing

↳ 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

Segmentation

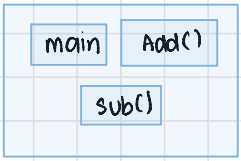
↳ Process divided into parts/segments (not fixed size)

↳ then put in m/m

PRO

↳ works from user point of view

↳ makes segments according to the code



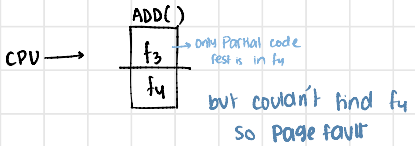
Paging

↳ 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

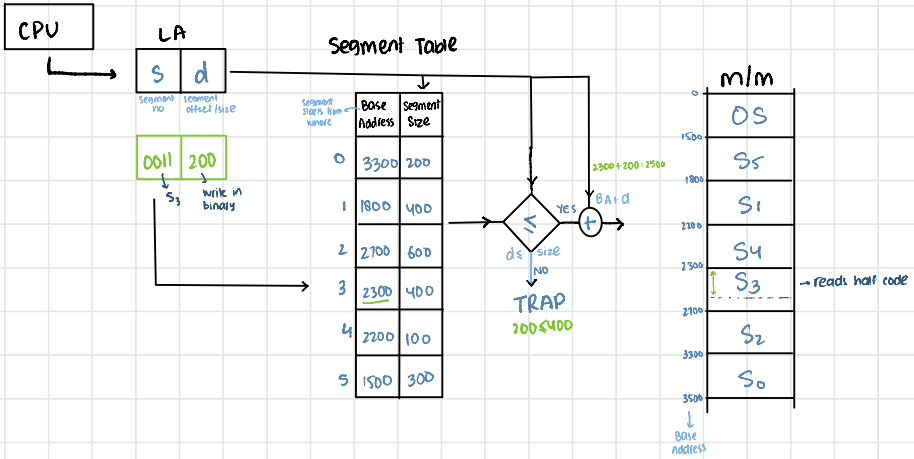


↳ CPU generates logical address

↳ will then asses what code it needs and hill where

↳ convert to physical address

LA → MMU → PA



5.10

Virtual memory

↳ we currently use

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

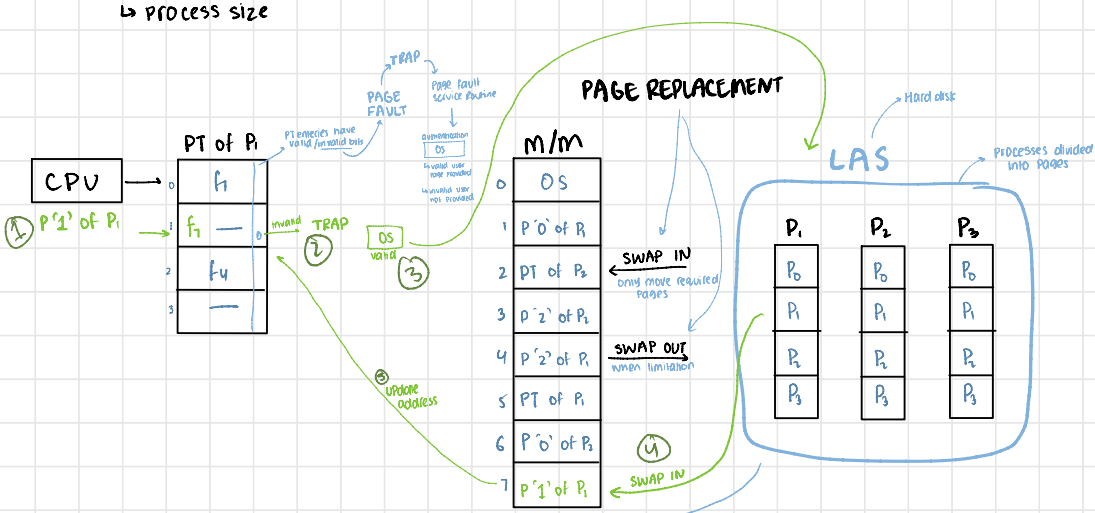
↳ no limitation on

↳ no. of processes

↳ process size

locality of reference

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



Demand Paging steps

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

User

   ↳ 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

Os

6. CPU given control

user

↳ 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

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 → time taken to access M/M

total time = 2n → ideal condition

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

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) (same as M/M)

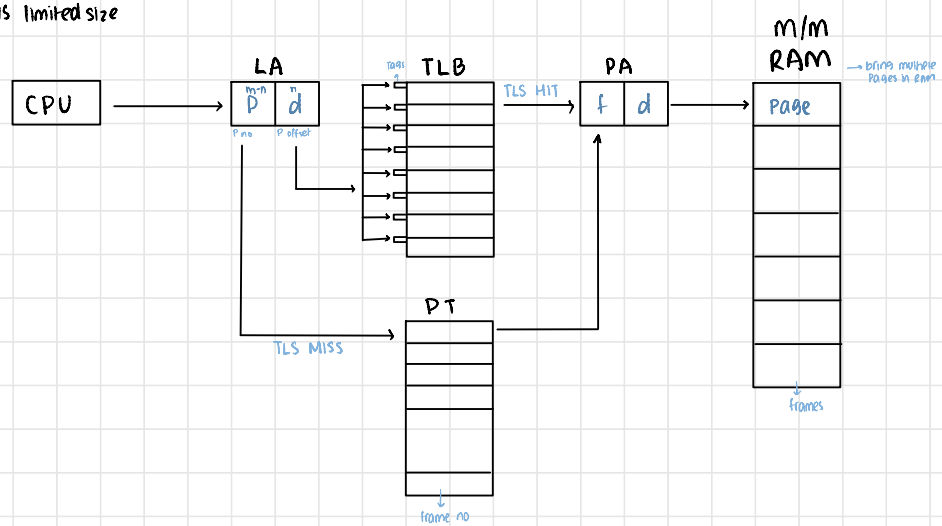
↳ used when Hits > Miss

↳ 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 access time: 10 ns, M/M access 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
	* ↓ MISS	*	*	* ↓ as no space replace 7 as it came in first	HIT	*	*	*	*	*	*	HIT	*	*	HIT

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**

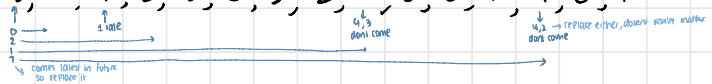
↳ increasing no of frames increased no of page faults

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

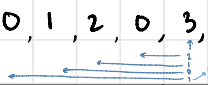
3. LEAST RECENTLY USED (LRU)

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

→ COUNTING BASED PAGE REPLACEMENT ALGO

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 ↑)

f4				2	2	2	2	2	2	2	2	2	2	2	0	0	0	0	0
f3			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f2		0	0	0	0	3	0	4	4	4	4	4	4	4	4	4	4	4	4
f1	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
			3₀	3₀	3₀	1₀	1₀	1₀	2₀	2₀	2₁	2₁	2₀	2₀	3₀	3₀	3₀	3₀	3₁	3₀
		2₀	2₀	2₀	2₁	2₀	2₀	6₀	6₀	6₀	6₀	3₀	3₀	6₀	6₀	6₀	1₀	1₀	1₀	6₀
	1₀	1₀	1₀	4₀	4₀	4₀	5₀	5₀	5₀	1₀	1₀	1₀	7₀	7₀	7₀	2₀	2₀	2₁	2₁	2₀
					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

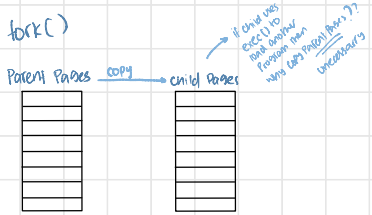
A copy of that page is made as it is modified

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



MODify/Dirty bit

↳ modify bit with each frame

↳ modify bit = 1

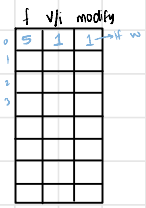
↳ if changes are done

↳ 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

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

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 *Pi* 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 reference again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

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; (S-1): parallel; 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 x 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)

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

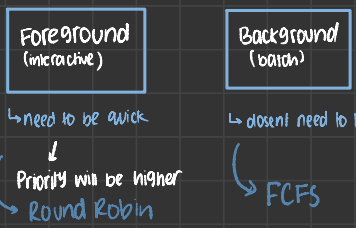
P: probability of page fault; Page fault service time: mostly in milliseconds (m sec); (1−P): no page fault; m/m access time: mostly in nano seconds (n sec)

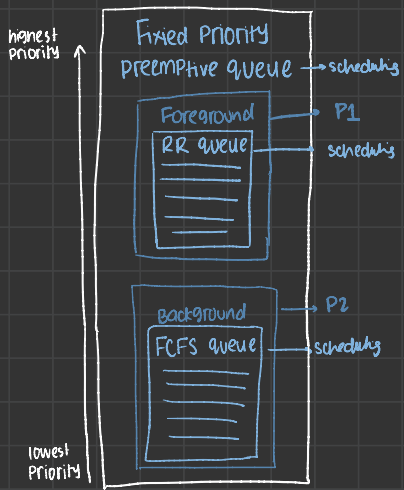
NO TALKING CLASS SAH BAHIR

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 permamentaly 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)

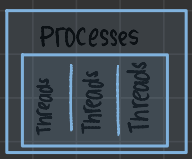
Parameters

- 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

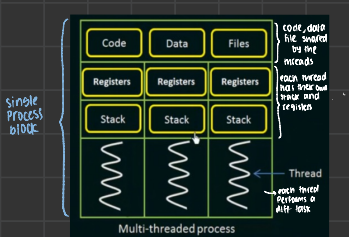
1. Thread ID
2. Program counter
3. Register set
4. Stack

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

S.N.	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.

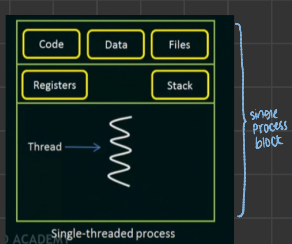
Multi Thread

↳ Performs multiple tasks at a time



Single Thread

↳ Performs one task at a time



MULTI-THREADING BENEFITS

↳ Responsiveness: 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

↳ Scalability: utilization of multiple cores for parallel execution

increases concurrency