Overview

1. Intro
   - swapping
   - virtual memory support
2. Demand Paging
   - page fault
   - performance
   - page replacement

Swapping

execution time binding

(the whole thing

- a process can be swapped out of memory temporarily to a backing store, and then brought back into memory for continued execution

  fast hard disk : big enough to accomodate copies of all memory images for all users w/ DMA

- swapped when they are in waiting or ready state

- advantages :

- swapping frees up memory for other processes to use
  → thereby increasing degree of multiprogramming
  → and providing memory space for larger processes

- policies for swapping out are based on process priority, size of process, state of process etc
  ↑
  decision is made by mid-term scheduler : the swapper

- major part of swap time is transfer time → ∝ amount of memory swapped. ✓
  ↑
  overhead incurred.

if there's a context switch happening and one of the processes is in the backing store, the overhead incurred during swapping will be high

recap :

long term = job-scheduler = which process to load to memory (new → ready)
short term = CPU-scheduler = which to execute (ready → running)
mid term = swapping

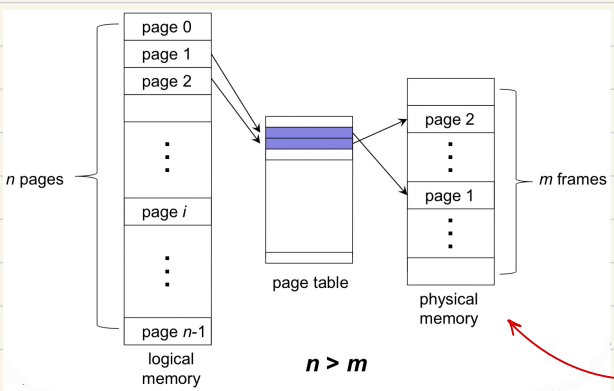
Virtual Memory Support

- so far we assume the whole program is loaded into the memory → limits program size to physical memory size.
  - including all library routines

↑

- to remove this: decouple logical memory size req from size of physical memory space
  - Virtual Memory!

the size of the virtual memory is only limited by the address scheme of a computer



no need to load all if we need to access something that is not there, simply load the page (signal page fault, let OS load)

pages not in the memory can be stored in the backing store

save time as unused pages aren't being swapped in and out of the memory.

implications

① logical address space is much larger than available memory

② only part of program has to be loaded ⇒ greater degree of multiprogramming ⇒ greater CPU use.
not necessarily true

③ needs to allow in and out of pages.

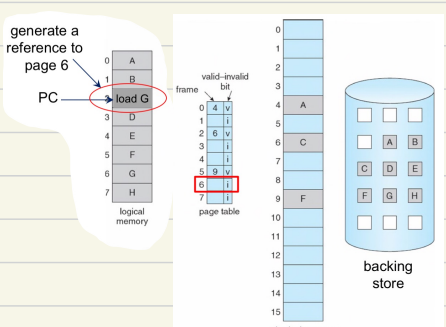
VM uses demand paging and demand segmentation

demand paging : pages are put in memory only when there is demand.

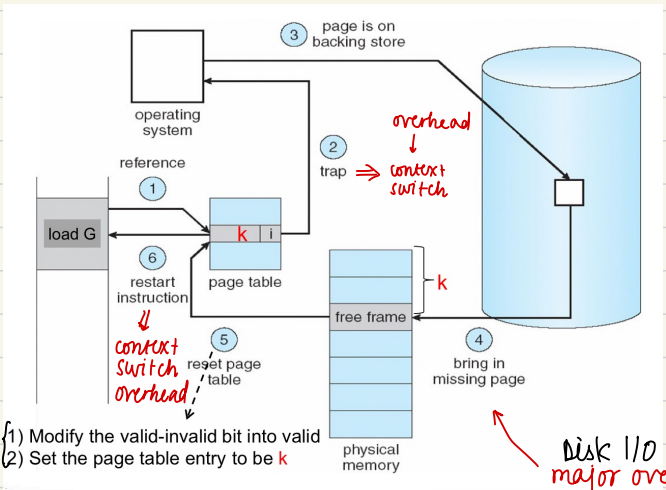
how will os know if page is in memory?

page table entry page no : frame no , offset , valid
↑
1 in
0 out

when we want a page :
pageTable [page no] . valid = true ? ⇝ return frame no
= false ? ⇝ page fault



page fault



demand paging performance

p = probability of page fault $\quad 0 \le p \le 1.0$

effective access time (EAT)

$$EAT = (1-p) \times \underset{mac}{\text{memory access time}} + p \times \underset{pft}{\text{page fault time}}$$

mac << pft , in practice

page replacement:

memory full, need to swap a page in

page out & page in ⟶ 2 disk I/O ← increases page fault time

↳ of a page not really in memory

steps

1. find victim page + page out
2. change victim's page table entry
3. load new page (page in)
4. update page table.

saves 1 disk I/O overhead. → so if not, can just do steps 2→4 → only if it has been written on / modified. Cause it still resides in disk

↑ "clean" page

↑ "dirty" page

keep track using "modify" or "dirty" bit

page replacement algorithms

- objective : lowest page fault rate
- its a given that ↑ in page frames = ↓ in page fault.
belady's anomaly contradicts that.
man said "actually ... no."

FIFO

- the simplest man, only bread and water vibes

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	4	4	4	5	5	5	5	5	5
F2		2	2	2	1	1	1	1	1	3	3	3
F3			3	3	3	2	2	2	2	2	4	4
	P	P	P	P	P	P	P			P	P	

9 page faults

w/ 4 frames → 10 page faults : belady's anomaly

Optimal Algorithm

- replace the one that will not used for the longest time

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	1	1	1	1	1	1	1	4	4
F2		2	2	2	2	2	2	2	2	2	2	2
F3			3	3	3	3	3	3	3	3	3	3
F4				4	4	4	5	5	5	5	5	5
	P	P	P	P			P			P		

break using FIFO

tie blw 1 & 2

- theoretically the best, practically, as can't keep track
- ∴ used as a bench mark

- optimal algorithm doesn't suffer from belady's anomaly
  ∵ more frames ⇒ more look ahead.

- if (inclusion) → not belady.

inclusion property : pages loaded in n frames is always a subset of pages in n+1 frames

if there's n frames, & no page fault, we will use n+1 frames

LRU

- replace the page that hasn't been used for the longest time

	1	2	3	4	1	2	5	1	2	3	4	5
F1	1	1	1	1	1	1	1	1	1	1	1	5
F2		2	2	2	2	2	2	2	2	2	2	2
F3			3	3	3	3	5	5	5	5	4	4
F4				4	4	4	4	4	4	3	3	3
	P	P	P	P			P			P	P	P

- no belady ∵ inclusion : for LRU, set of pages in the memory, would be the n most recently referenced ones.
  if no of pages is increased, these pages will still be the most recently accessed ones and will still be in the memory

① counter implementation
- → clock incremented for every memory reference.
- → when page is refed, clock value is copied to 'time-of-use' field of the page
- → smallest value is kicked out

② stack implementation
- → doubly linked list
- → when a page is referenced move it to top.
- → bottom has LRU page
- → update is expensive, but search is not needed.

↑ both need too much hardware

↳ they give us one bit → reference bit

hardware does

R = 1 when referenced

LRU approximation

Second Chance

- Kind of fifo, use oldest as candidate
  R = 0 & oldest ⇒ out ⇒ oldest & not used yet (if there's none)
  R = 1 & oldest ⇒ keep, check second oldest

↑ Implemented by Clock Algorithm

second chance
- clock hand points to oldest page
- when page fault, check; if R is set (yes → clear R=0, advance hand)
  (no → evict, advance hand)
- worst case, all set, cycle through whole queue and become FIFO
- best case: LRU

high no of page faults ⇒ low hold of principle of locality

Allocation of Frames

- OS has to decide how many pages to bring to physical memory / how many frames to allocate to a single process

→ smaller no allocated ⇒ greater number of processes, more multiprogramming (degree)

2small → small no of pages ⇒ increased no of page faults

2big → beyond certain number due to principle of optimality, no of frames won't affect rate of page fault

how can I decide the optimal number?

fixed allocation policy

- gives a process a fixed number of frames
- equal / proportional allocation
- page fault → one of those pages must be replaced
- decided at load (process creation) time
- based on type of process or by system manager

variable allocation policy

- allows the number of page frames allocated to a process to be varied over the lifetime of the process
- indicators are no of page faults

↑ seems nice. requires OS to assess behavior of running processes. causes software overhead + is dependent on hardware mechanisms provided by processor platform

activated by page fault w/ no free page frames

Global Replacement	Local Replacement
process selects a replacement from the set of all frames one process can take a frame from another provided it is unlocked	process selects a frame from its own allocated set ↑ may hinder other processes by not making available its less used pages/frames

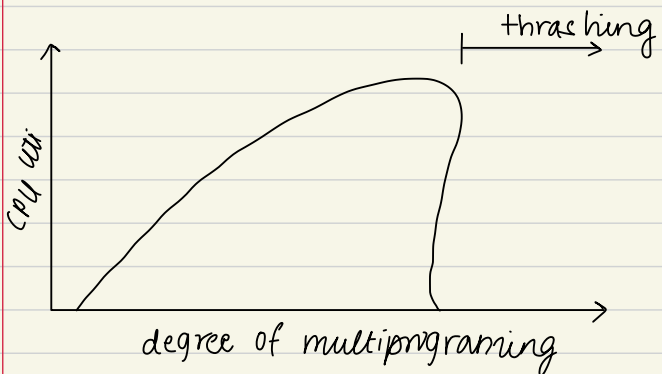
	local replacement	global replacement
fixed allocation	can	nah
variable allocation	from allocated which can change	from all, varies

a process is busy bringing pages in and out rather than executing instructions.

↖ Thrashing

- if a process doesn't have "enough" pages, the page fault rate is very high
  ↘ low CPU utilization

- improve CPU utilization ⇒ increase the degree of multiprogramming



all active pages are loaded, it needs one more so fault, replace, fault, replace → continues.

soln: OS tries to guess which pieces are likely to be used in the near future based on recent history

① WS model
② Page fault frequency.

Working-Set Model

main concept.

- Locality of reference : processes tend to refer to pages in a localized manner.

loop → ① temporal locality : time (recent)

array element access — sequential → ② spatial locality : space (physically)

```
while(i)          ← temporal
  sum += a[i--]   ← spatial
```

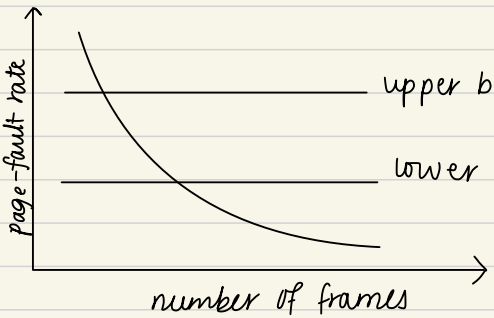
- $\Delta \equiv$ working set window = no of page references.
- $\max\{n(WS)\} = 10$ — repeated not in set
- max fault : when WS is totally new ↖ happens when the process travels to another locality
- $n(WS) = WSS_i \rightarrow$ Process P_i
- $D = \Sigma WSS_i$ = total demand for frames for all processes

m = total # of frames

$D > m \Rightarrow$ thrashing

if $D > m$ ⟶ swap out one of the processes ↘ so $\Delta \downarrow$

Page Fault Frequency Scheme



increase number of frames

decrease number of frames

- establish a page-fault rate : if we go above or below → then increase or decrease

if increase but no available frames → swap.

True or False

True or False

Logical address needs to be translated to physical address during execution if address binding is done in load time. F
execution

Memory compaction is used to reduce internal fragmentation. F
external

Consider a paging system with page table stored in memory. Using translation look-aside buffer (TLB) will always reduce effective memory access time. F
depends on hit ratio

Using inverted page table increases address translation time. T
cause pid

If segmentation is used for memory allocation, the logical address space of a process will remain contiguous in physical memory. F

Similar to paging, memory allocation using segmentation suffers from internal fragmentation. F

True or False

Context switch contributes the most to the overhead of handling page faults. F
(1) → disk I/O (2)

When there is a page fault and memory is full, the victim page identified by the page replacement algorithm always needs to be copied to the backing store. F
only if written / dirty

As the number of available page frames increases, the number of page faults will always decrease no matter which page replacement algorithm is used. F
fifo

Using demand paging for virtual memory support, CPU utilization can be always improved by increasing the degree of multi-programming F

Variable frame allocation implies a global policy must be used for page replacement. F can do local also.

Thrashing can be detected by evaluating the level of CPU utilization as compared to the level of multiprogramming. T