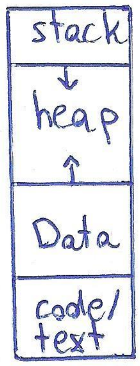
Memory Management

"way to allocate portions of memory at a program's request then freeing it for reuse"



local variables
(function call -> return)

dynamic allocation
(malloc -> free)

global variables
(start -> finish)

• PROCESS LOADING/ALLOCATION:

- each process has its own address space (abstract memory for processes to live in)
- A program being compiled can't know where it will be allocated. That is decided during execution and may change due to process swapping

• SWAPPING: storing the memory content of a process on the disk in a reserved area known as swap area.

- Used when loading a high priority process into a full memory.

• MEMORY ALLOCATION:

- contiguous (old OS, no longer in use)
- non-contiguous

☆ CONTIGUOUS: [address space must be all in one location]

sharing date between 2 processes is impossible

- Each process has a Base Register and Limit Register

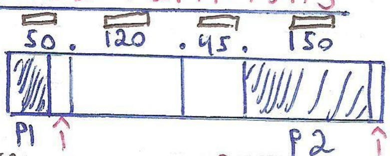
Base Register: holds the "Relocation" start address

Limit Register: holds the "Protection" end address

Note: virtual address: 0 -> size-1
physical address: x -> x+size-1
} handled by MMU (Memory Management Unit)

process $\xrightarrow{\text{virtual}}$ MMU $\xrightarrow{\text{physical}}$ physical memory

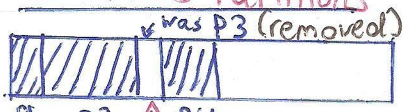
FIXED Partitions



each partition has a queue for waiting processes

Internal fragmentation

DYNAMIC Partitions



external fragmentation

Both suffer from fragmentation that can be fixed with Compaction

~ • try to fit multiple processes in one region to free up others

both costly: cpu / time

~ • re-locate processes and move all free space to the right

Contiguous ALGORITHMS:

FIRST FIT

start from begining and use the first hole that fits.

NEXT FIT

start off from last place we inserted and look for a place to fit.
end => start over

BEST FIT

search entire memory to find best spot

WORST FIT

search memory for worst/most wasteful spot

Management of free & used space in memory:

Bitmaps: - divide memory into allocation units (smaller unit size => longer bitmap)
- each unit has a (0/1 => free/used) bit in the map.
- to get a k-unit sized process into the memory, we need k consecutive 0 bits => hard after a lot of swapping.

Linked Lists: Keep track of processes (P) and holes (H)

[P | 0 | 5 |] → [h | 5 | 3 |] → [P | 8 | 7 |] → [P | 15 | 4 |] → [h | 19 | 30]

start, size

Note: sometimes we use separate lists for P and H.

Note: Process grows => must re-locate if it doesn't fit anymore.

★ NON-CONTIGUOUS: - solves problem of external fragmentation
- minimizes problem of internal fragmentation

Pagination: • virtual address (page#, offset) space: set of addresses a process can access
it is made up of pages of fixed size. (usually 4kB)

ex: 64 kB memory and 4kB pages
=> 64/4 = 16 pages.
find page/location of address 4120:
- Method 1: $\underbrace{(4120)_{10} / 4KB}_{\text{page\#}}$, $\underbrace{(4120)_{10} \% 4KB}_{\text{offset}}$ (page size) → (page#, offset) (1, 24)
- Method 2: ($\underbrace{0010}_{\text{page\#}}$ $\underbrace{0110\ 0011\ 0011}_{\text{offset}}$)

Note: usually calculate offset from page size ($4kB = 2^{12}$: 12 bits) and the rest are for page #.

• physical address: set of frames.
frame count = page count

Note: compiled programs go into the swap area. pages are loaded in when needed. processes go on multiple pages

MMU manages physical/virtual conversion using page tables with one entry per page

each row has (4B):

p#	presence	protection (rwx)	referenced	clean	f#

presence: if this page exists in physical memory (if true then f# ≠ null)

page not loaded in memory (presence = 0) => page fault

Each process has it's own table, that's also stored in the memory (it's location is in a special register) => get table → get frame → get data from frame

32bit architecture => 2^{20} entries per table => 4 MB per table/process

By the Principle of Locality, 90% of accesses are for only 10% of addresses => no need to save entire table

Solutions: - Multi-Level Page Table
- Inverted Page Table

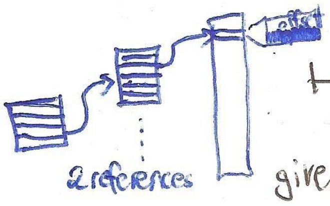
• **Multi-Level-Page Table:** page table whose rows point to other tables

=> don't load entire table only parts we need.

entries per table = page size / Row size

ex) 32-bit: 4KB (4096) page size, Row is 32 bits (4B) by default } => $4096/32 = 1024 = 2^{10}$ entries

=> 10 bits



PT1	PT2	offset
10bits	10bits	12bits (2^{12} to hold 4kB pages)

given $(00403004)_{16} = (4206596)_{10} = (4MB \times 1) + (4kB \times 3) + 4$

PT1 row 1, PT2 row 3, offset

64-bit: 10 10 10 10 10 12

6 references to memory per mapping

By the principle of locality, we speed up paging by putting the mostly hit pages in the TLB (Translation Lookaside Buffer) [hardware] that maps virtual to physical inside MMU

=> always check TLB before accessing page tables

• **Inverted Page Table:** one global page table for all processes (entry count = frame count)

ex) 64-bit: 4KB (4096) page size, Row is 16B by default, take RAM as 512 MB } $512MB = 2^{29}B$ ∴ page table $= 2^{29-12} = 2^{17}$ entries

∴ $2^{17} \times 16 = 2MB$ total page table size

small page table but long search times (might iterate all 2^{17} entries)

small fix: Linear Inverted page tables: one entry per physical frame

=> no need to store physical p# as it is the index. [iteration time still too long though]

PAGE REPLACEMENT ALGORITHMS:

Note: They can be local (choose victim from process's pages) or global (from all pages in memory)

★ **Random Algorithm:** choose the victim randomly. $w = \{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5\}$

★ **FIFO Algorithm:** remove the page that spent the longest time in memory.

w	f1	f2	f3	fault
1	1			y
2	2	1		y
3	3	2	1	y
4	4	3	2	y
1	1	4	3	y
2	2	1	4	y
5	5	2	1	y
1	5	2	1	N
2	5	2	1	N
3	3	5	2	y
4	4	3	5	y
5	4	3	5	N

Faults: 9

Faults on 4 frames: 10

★ **LRU (Least Recently Used) Algorithm:** referencing a page sets it to the front of the list.

w	f1	f2	f3	faults
1	1			y
2	2	1		y
3	3	2	1	y
4	4	3	2	y
1	1	4	3	y
2	2	1	4	y
5	5	2	1	y
1	1	5	2	N
2	2	1	5	N
3	3	2	1	y
4	4	3	2	y
5	5	4	3	y

10

8

★ **Second Chance Algorithm:** FIFO + "R" bit. replacing page if R = 1 => set R=0 and keep looking (left)

w	f1	f2	f3	faults
1	1			y
2	2	1		y
3	3	2	1	y
4	4	3	2	y
1	1	4	3	y
2	2	1	4	y
5	5	2	1	y
1	5	2	1	N
2	5	2	1	N
3	3	5	2	y
4	4	3	5	y
5	4	3	5	N

9

10

★ **NRU (Not Recently Used) Algorithm:** pages go into 4 classes:

C1: R=0 M=0

C2: R=0 M=1

C3: R=1 M=0

C4: R=1 M=1

priority; R: ref., M: mod.

• page ref => R=1

• cycle => all R=0

• fault => random from C1

★ **Optimal Algorithm:** remove the page that will be referenced as late as possible in the future

⚠ can't implement, only for comparisons

Belady Anomaly: more memory doesn't mean less faults (sometimes more).

Segmentation: user's view of memory becomes different than the actual physical memory. Where the user sees a program as a single partition it can be spread out on multiple segments.

- Virtual address space is now a set of segments of ≠ sizes each representing a different part (prog, library, stack, variables, etc...)

- Address:

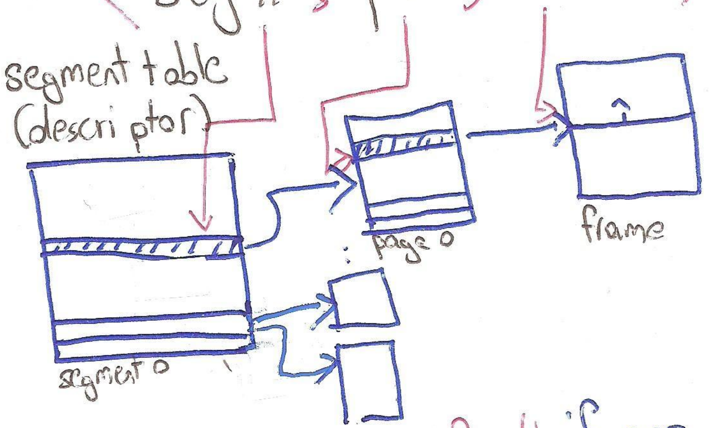
⟨seg #, offset⟩

each process is assigned a table of segments:

segment #	base address in memory	size (bytes)	other info

In reality, we use a combination: SEGMENTATION with PAGINATION:

⟨seg #, p #, offset⟩

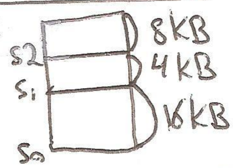


segmentation fault or page fault if one isn't already loaded in memory

ex) given: 11 | 9 | 12 — s # | p # | offset ⇒
- pg size = 2^{12} since offset is on 12 bits
- max segment size is $2^9 \times 2^{12} = 2^{21}$

---

ex) given: pg size = 4kB
physical memory = 64kB



at time t_1 given in memory:

(S0, P1) : f2 (S1, P1) : f9
(S0, P2) : f0 (S2, P0) : f12

CPU generated: $\boxed{8212}$

We can deduce:
- 8212 < 16kB ⇒ S0
- 8212 / 4kB = 2 ⇒ P2 ⇒ f0
- 8212 % 4kB = 20 offset ⇒ ⟨0, 2, 20⟩

Comparison:

paging	segmentation
made to get a larger linear address space without expanding physical memory	made to aid sharing & protection of data by breaking up the data
doesn't separate procedures & data	✓
automatic / done by system	dev. must be aware of memory limitations
X	accomodates tables with changing sizes