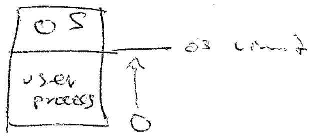
Mem Mgmt

> How to share main memory?
> Adv / Disadv's of Static relocation?
>   dynamic
> H/w interactions

Motivation

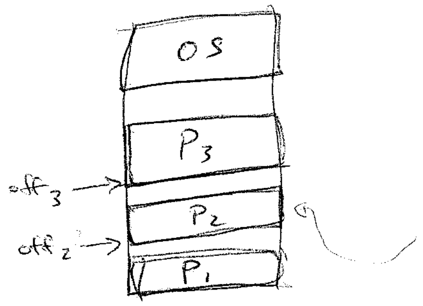
⇒ Simple uniprogrammed env.
early batch, PCs
why bad?
1 process @ a time, protection



⇒ Multiprogramming: goals
Sharing: ≥ 1 process coexist in mem
transparency: to process, looks like large private memory
works even w/ ~~the~~ others running
protection: cannot corrupt OS, other processes
cannot read from others : why?
flexible: allow proc to do what it wants
efficiency: Should not waste too many cycles doing this
should not waste memory either

Static relocation:

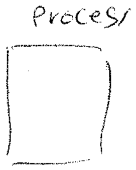
~~Relocation~~ static Relocation:
can run anywhere in memory
modify addresses statically at load time



> compile all as if loaded @ address 0
> loader (which gets process running) goes through and changes addresses @ run time

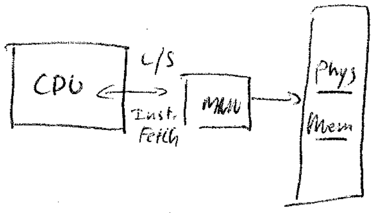
Positives: simple, no h/w required
Negatives: ① fragmentation, ② ~~flexible~~ inflexible: may not be able to increase space while running
③ protection: non-existant

Dynamic Relocation



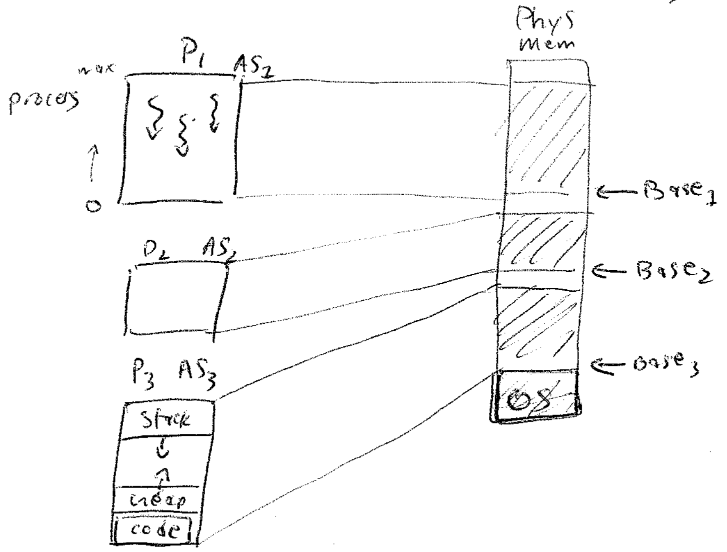
> compiled as if @ 0 => max

> H/W translates logical addresses => physical addresses



> all mem P can address: "address space"

Address Space



H/W needed

Operating modes:

Privileged: in kernel
when trap into OS (systemcall)
=> certain priv. instructions,
=> can access all of mem

User: when process runs
provides logical view of mem

How done? Base + Bounds registers

Base: start location

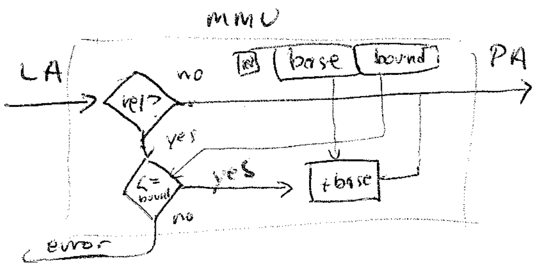
Bounds: max location

Implementation:

Translation: on every mem access (LD/ST, ifetch)

compare logical to bounds reg
if LA > Bounds, error

To get PA: LA + Base



Key to protection:
H/W limits access of process to memory

Context Switch

Add Base + Bounds to PCB

Steps during ctxt switch

- ⇒ privilege mode
- ⇒ save (registers) base + bounds of old
- ⇒ restore (registers) new
- ⇒ change to user mode + jump to new process

Q) what if we forget to change base + bounds?

Protection

user cannot change base + bounds
user cannot arbitrarily become "privileged"
key: modes provided by H/W

Advantages

- ⇒ supports dynamic relocation
- ⇒ protection
- ⇒ simple : base + bounds
- ⇒ ↓ fast

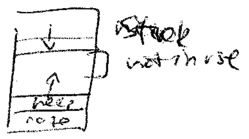
> can swap

Disadvantages

> Allocation of process contiguous in real (phys) mem

fragmentation : can't alloc new process (solve by swapping)

> wasteful must allocate entire regime



> Sharing is hard

running two copies of ~~ns~~ netscape

⇒ could share code

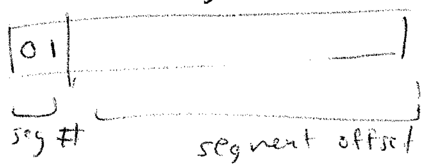
Segmentation and Paging

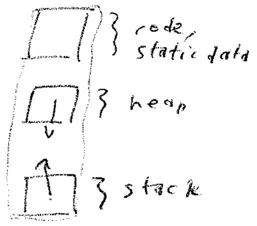
+/- of segmentation
+/- of paging
combine?
speed up?

Segmentation

> Divide AS into "logical" segments
  Each has base/bound
  Per segment · read/write bits, protection (before, all in one)

> How to designate?
  > Part of logical address





  > Implicit by ref type (PC or not)
  > Registers

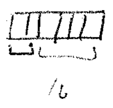
Segment Table

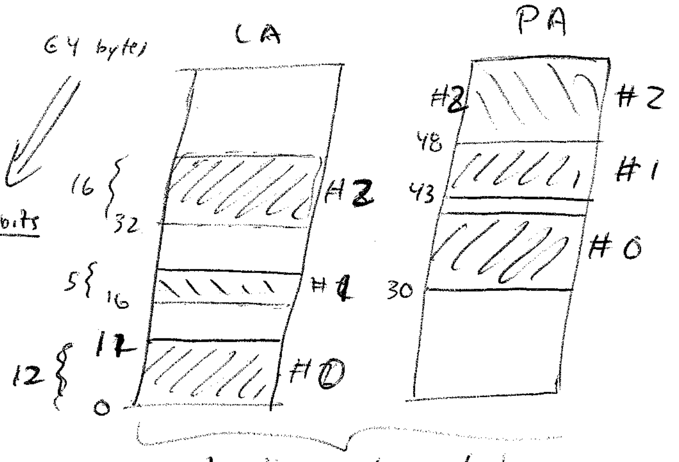
> Need base/bounds ∀ segments in process
  Table : Indirection look up seg info before add/compare

Seg #	Base	Bound	other
0	36	11	
1	30	4	
2	48	15	
3	—	—	

64 bytes

Address : 6 bits





do these have to be same size?

Find PA of
LA: 0 ⇒ 36 #0
18 ⇒ 32 #1
47 ⇒ 63 #2
25 $^{\#1}$⇒ OOB $^{\#1}$

Managing Processes

> Creation
  Find contiguous space per segment
  Fill in base / bounds

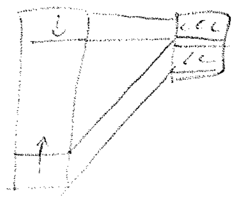
> Mem alloc when no contiguous space
  Compact memory (move segments, update bases)
  (Swap segment to disk?)

> Ctxt switch
  Segment Table ⇒ PCB

> Exit
  Return to free list

Pros / Cons

+/
Diff. protection / segment
(read-only for code)
Sharing is easier for some segments
Can relocate segment easily
Enables sparse, large address space

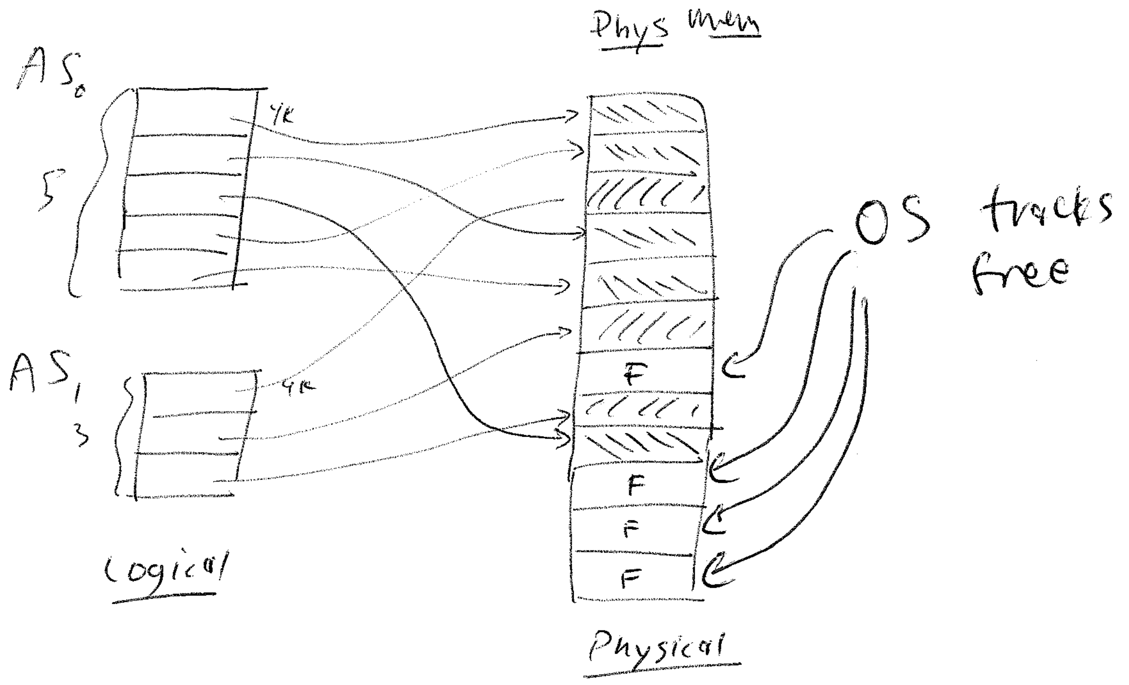


−/
Still have to find contiguous mem
⇒ External Frag: wasted memory

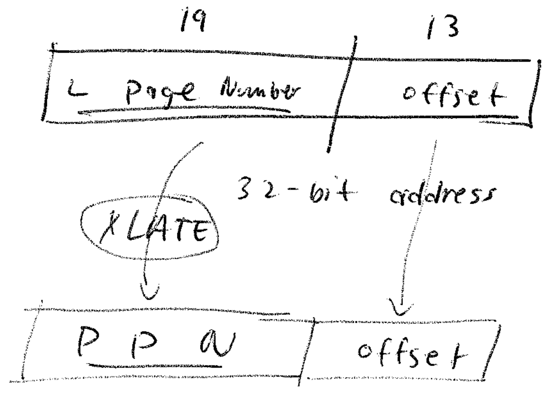
Paging

Divide mem into fixed-sized pages

typical page size: 4K, 8K



Translation



8K page

XLate: Page table

> just a data structure

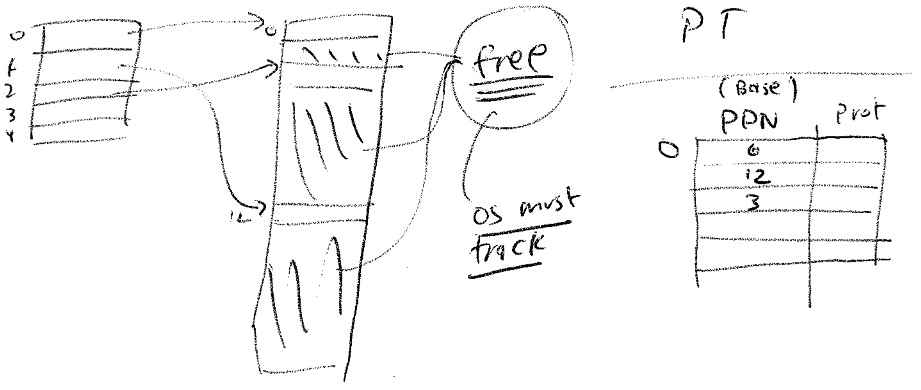
PPN = lookup(PPN);

> One "page table" per process
Base addr (PPN) + protection bits

> How many entries in table? 2^{19} entries

BIG

Example



Advantages

+) Fast to allocate / free

alloc/ keep free list, grab first
(no best fit needed => all fit!) why?

free/ add page to free list
(no sort needed)

> Easy to swap to disk

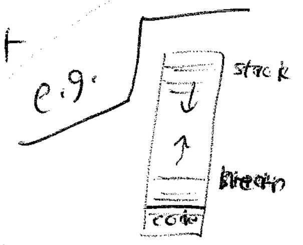
page size ~ block size
just move needed pages to disk

Disadvantages

+) > Every load => 2 loads, every PC fetch => 2 fetches

> can't keep entire page table in ~~memory~~ special registers!
(must keep in main memory)
MMU: base addr of table

> Huge! Lots of mem required
Simple: entry for all pages, used or not
Better: Base/bounds BUT => e.g.



> Fragmentation: internal (not a big deal) (waste)
Larger page => fewer entries => more internal frag

Combine Paging + Segmentation

Structure : segment is a logical unit of AS

code, data, stack : vary in size, large

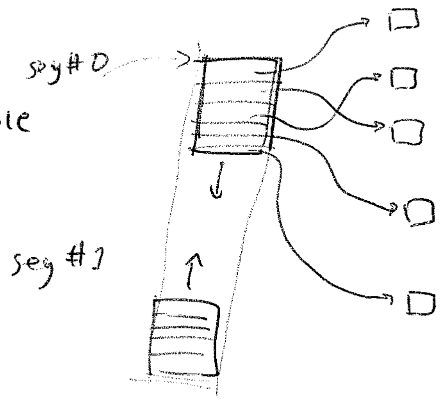
each seg. is made up of many pages

2-level lookup

Page-table ~~/~~ per segment

Base (real addr) + bound (size) per table

(use to check for valid page)



LA: 3 components

Seg#	Page#	Page offset

Plus

+) Segments: Sparse addr space

no wasted page table space now

+) paging

external fragment problem gone

easy to grow

+) Both : flexible sharing (code)

seg / page / both

Disadv/

- internal frag. inc
  (~~just~~ more segs ⇒ more pages)
- overhead
  tables in main memory
  1 or 2 extra loads / ref] TLB
- Large Page Tables
  don't want to allocate PT contiguously] page page tables

> TLB
> Inverted PT
> Paging Page Tables

Advantages of Paging

Fast allocation, freeing
no search, list ops easy
Easy to move back and forth from disk

Problems

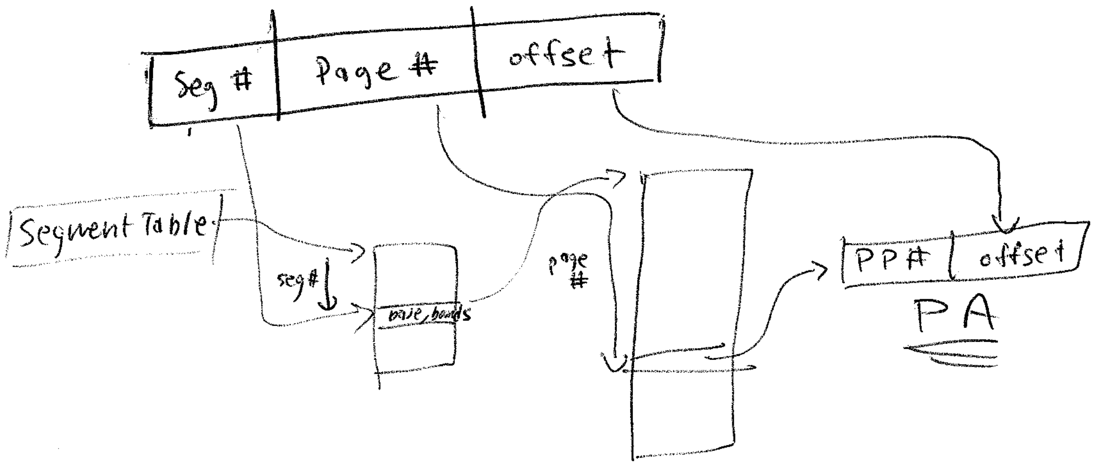
Page table is too big
⇒ must keep in main memory, not MMU
(MMU might hold info about where processes page table is)
} example

⇒ one entry per page
base/bounds may help, but not w/ stack, heap

⇒ internal fragmentation
(how much waste?)

Solution: Combine paging w/ segmentation
Segments ∝ logical units (code, data, stack)
Segment consists of a bunch of pages

2-levels of mapping
Page-table per segment
Base (real address) + bound (size) for each PT



Advantages

<u>Segmentation</u> : <u>sparse</u> address spaces

<u>Paging</u> : no worry of <u>external</u> fragmentation
easy to grow w/o reshuffling

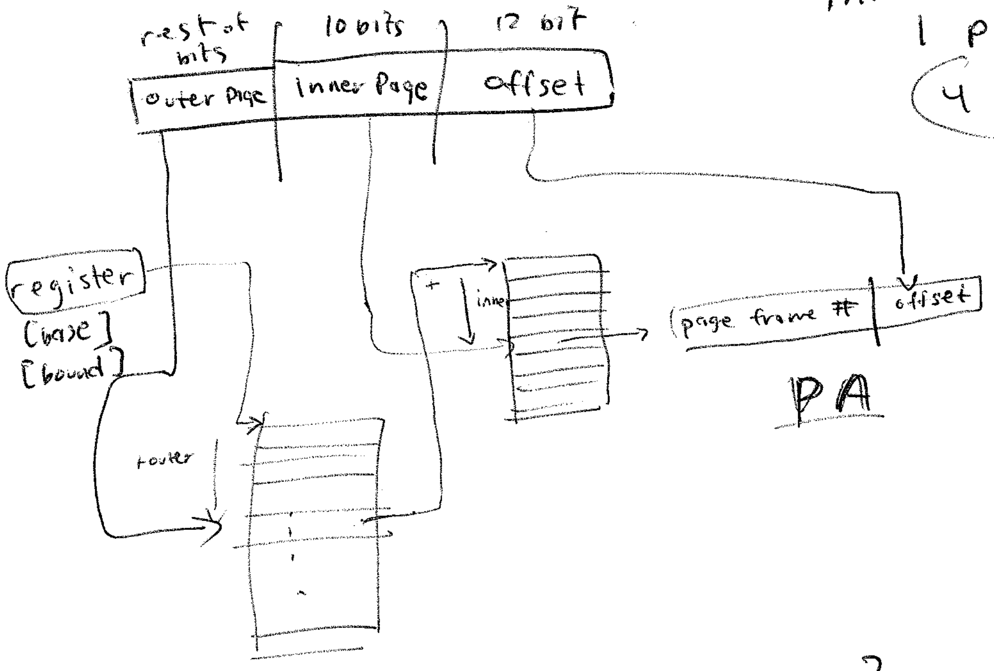
<u>Both</u> : can share page / segment (page table)
between processes

Minus

- slight increase in waste due to <u>internal</u> ~~fragmentation~~ fragmentation
- mem. <u>access overhead</u>: 1 or 2 references per <u>access</u>
- size overhead <u>large</u> page tables still a problem

Multi-level page tables

Before: entire PT for segment must be <u>contiguous</u> in <u>physical</u> memory



inner-level:
1 PAGE / <u>PT</u>
(4 bytes / entry)

$2^{10} \cdot 2^{2} \Rightarrow 2^{12}$

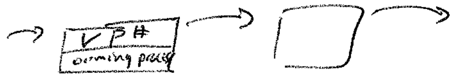
$\Rightarrow$ how many bits for outer page?

Inverted Page Tables

Idea: why have so much of VA defined by page table?
instead, have table that corresponds to size
of physical memory

Translation: $f(VP\#) \Rightarrow PP\#$
can be any data structure

Could keep lists



why bad?

efficient lookup: hash on VP#

bad: sharing is difficult: one VA per PA (typically)

Slow: TLB

need: speed up accesses w/ H/W support
use H/W cache of translations

VPN#	offset

↓

in TLB → PP# + offset ⇒ PA

not?

↓

do full translation
update TLB
→ who does this?
H/W or S/W

Size: usually small
(32, 64 entries)

Policy: fully associative