Today

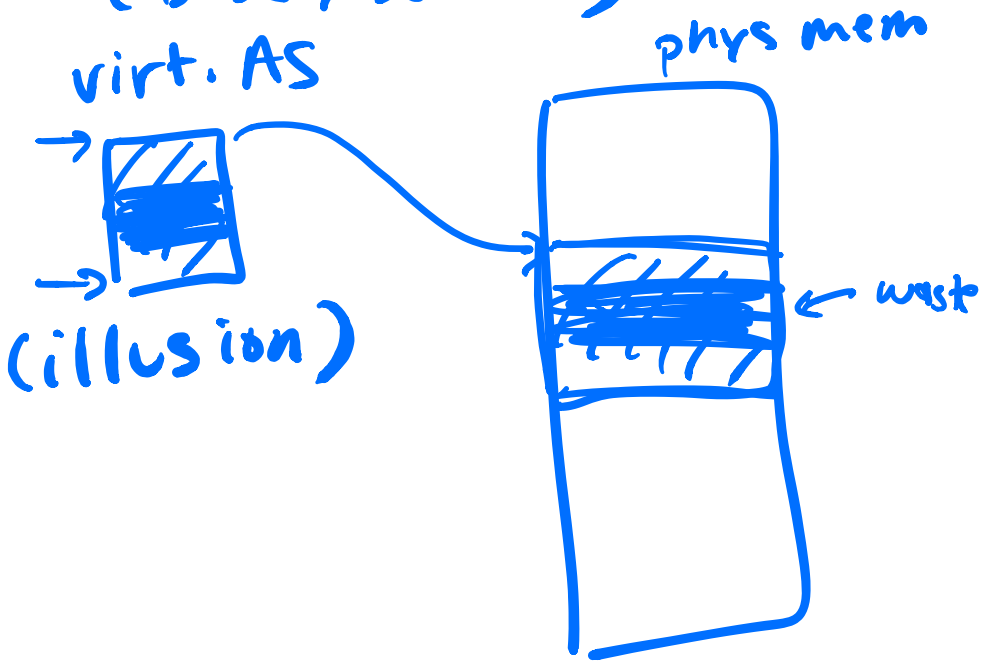
welcome!

→ except econ

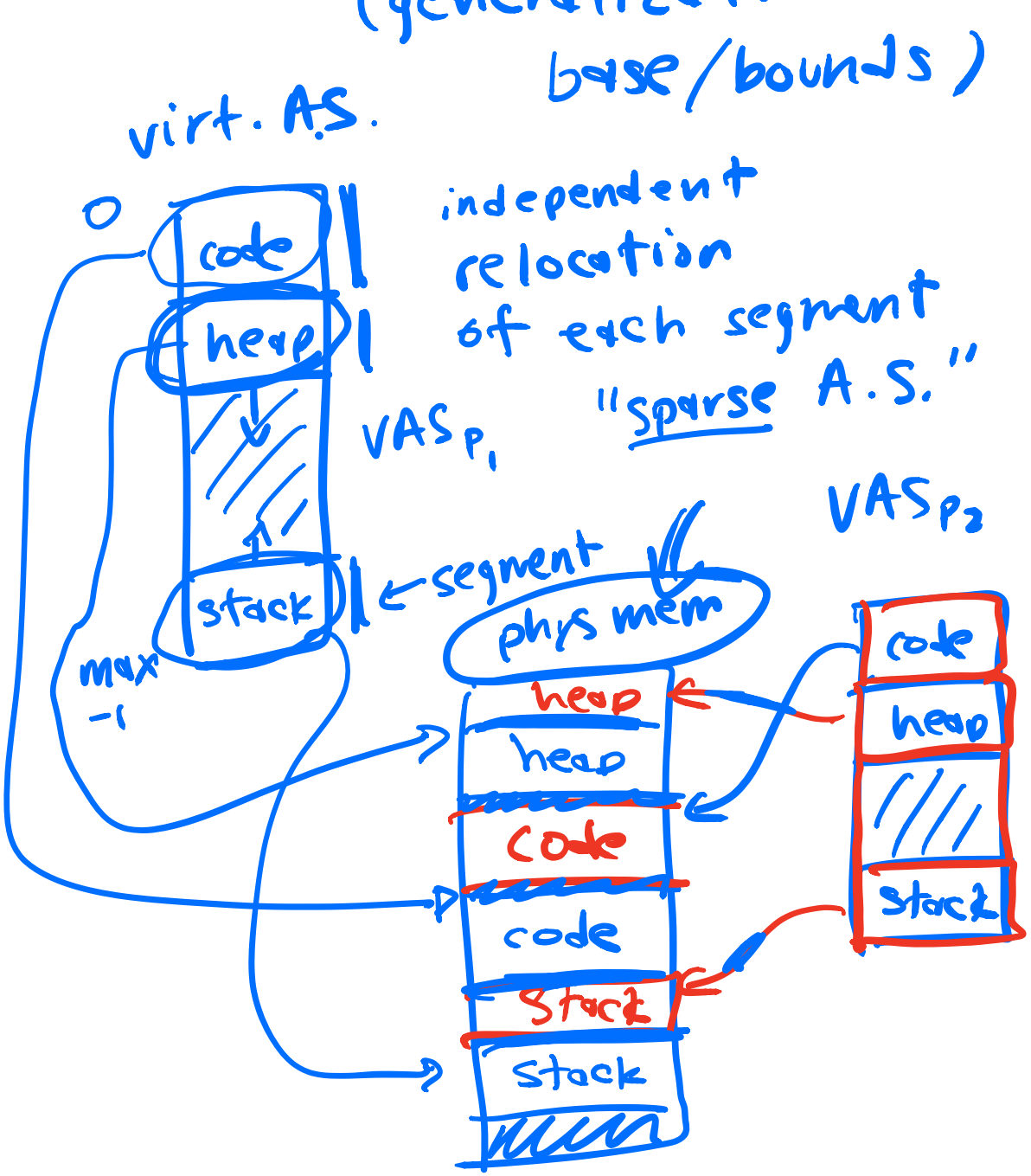
Virtual Memory

=> Mechanisms

→ Dynamic Relocation (base/bounds)



→ Segmentation
(generalization of base/bounds)



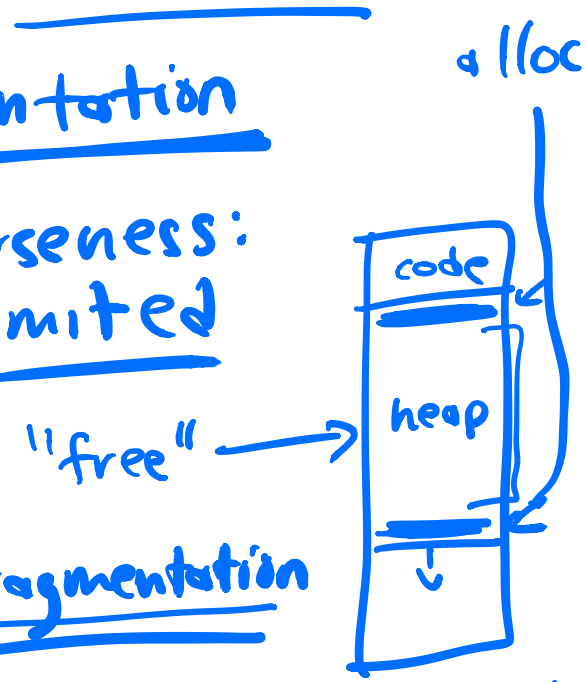
Aside:
2-levels of mem mgmt

→ inside virt addr space
C: malloc/free
+ malloc library
stack: language runtime

→ OS: phys memory

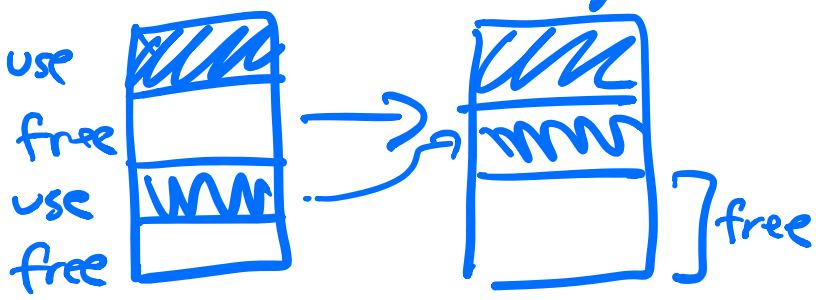
Problems: segmentation

→ support sparseness:
but limited



→ external fragmentation

→ have to reject request
or compact memory



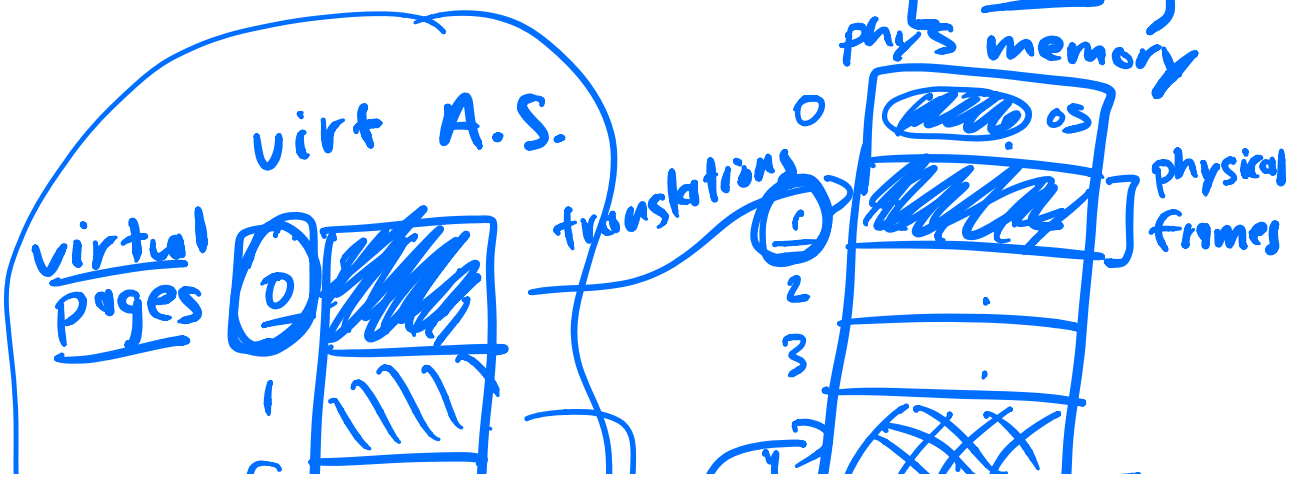
Today: Paging

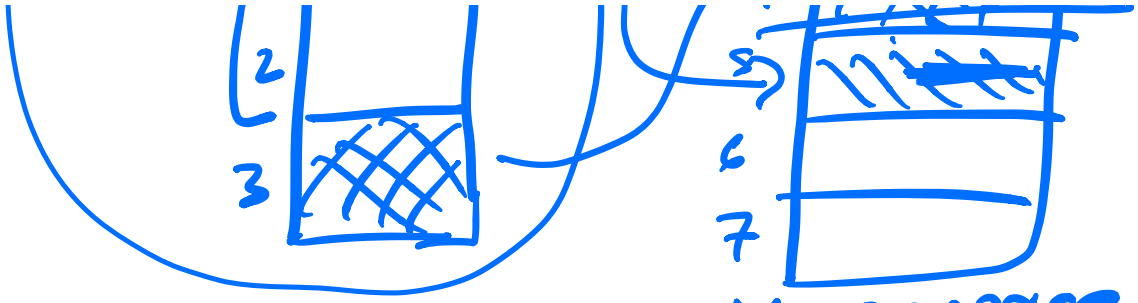
→ Basics
→ Slow
→ Memory hog

} potential problems

Paging:

Divide up
→ Virt. Addr Spaces
→ Phys Memory
into fixed-size units
called pages → typically: 4KB





typical: 32-bit address space
w/ 4 KB pages

→ 2^{20} pages in A.S.?

$2^{32} / 2^{12} \Rightarrow 2^{20}$

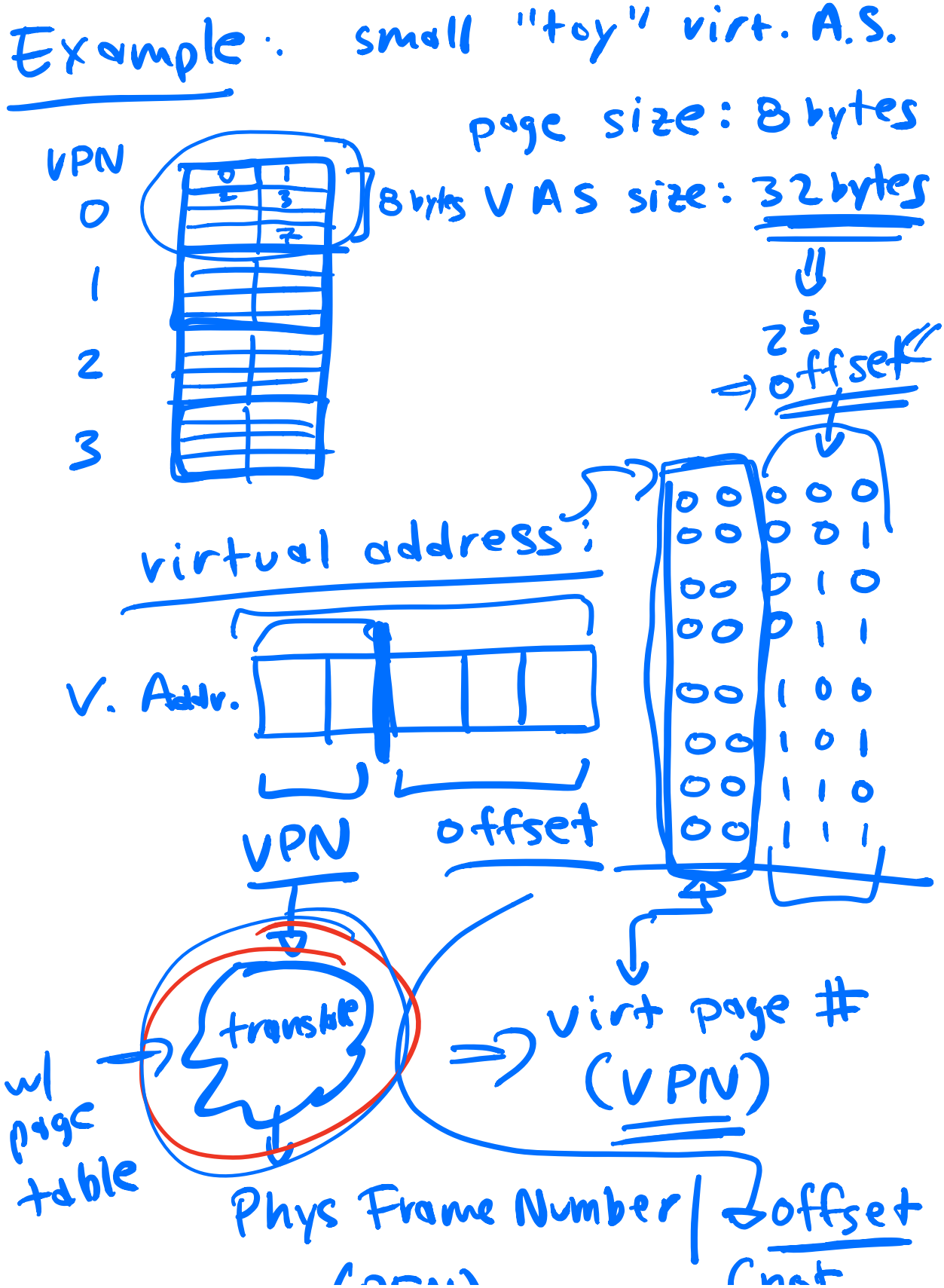
"CS million"

$2^{10} \Rightarrow 1$ KB

$2^{30} \Rightarrow 1$ GB

1) a lot of information (per process)

⇒ generally stored in memory (*slow)


Example: small "toy" virt. A.S.
page size: 8 bytes
VAS size: 32 bytes
2^5
offset
VPN
0
1
2
3
8 bytes
virtual address:
V. Addr.
VPN
offset
translate
w/ page table
virt page # (VPN)
Phys Frame Number
offset

(PFN)

translated)

Translation Information:

=) store this in mem somewhere

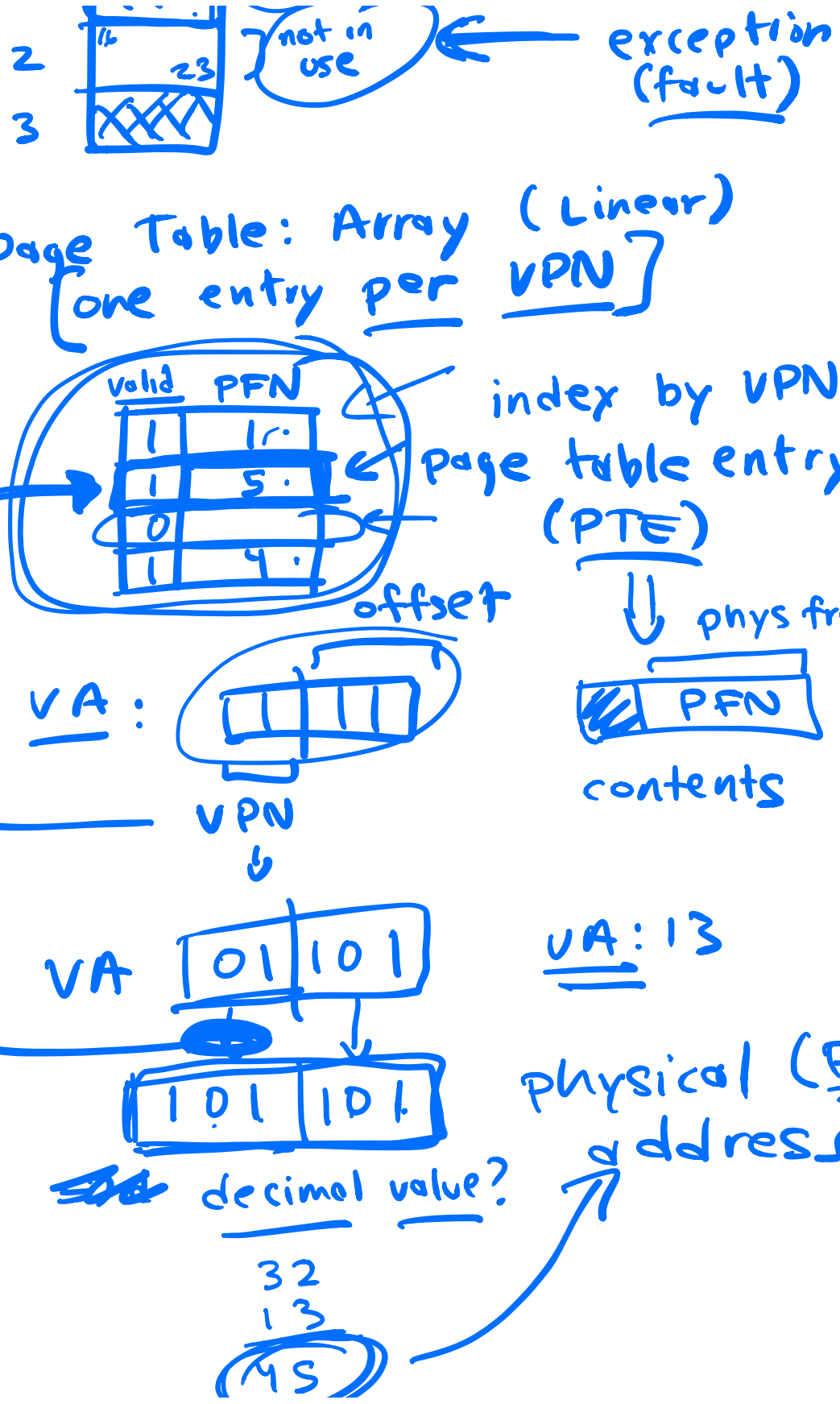
=) call data structure that stores this info a

[page table]

Simple (to begin)

=) array (one per process) => linear page table

=) Contents (of page table)


2
3
23
not in use
exception (fault)
Page Table: Array (Linear)
[one entry per VPN]
valid
PFN
1
1
0
1
index by VPN
page table entry (PTE)
01
offset
VA:
phys frame #
PFN
contents
VPN
VA
01
101
VA: 13
101
101
decimal value?
physical (PA) address
32
13
45

⇒ How does translation occur?

⇒ [Limited Direct Execution]

h/w → efficiency

⇒ h/w does most of the work

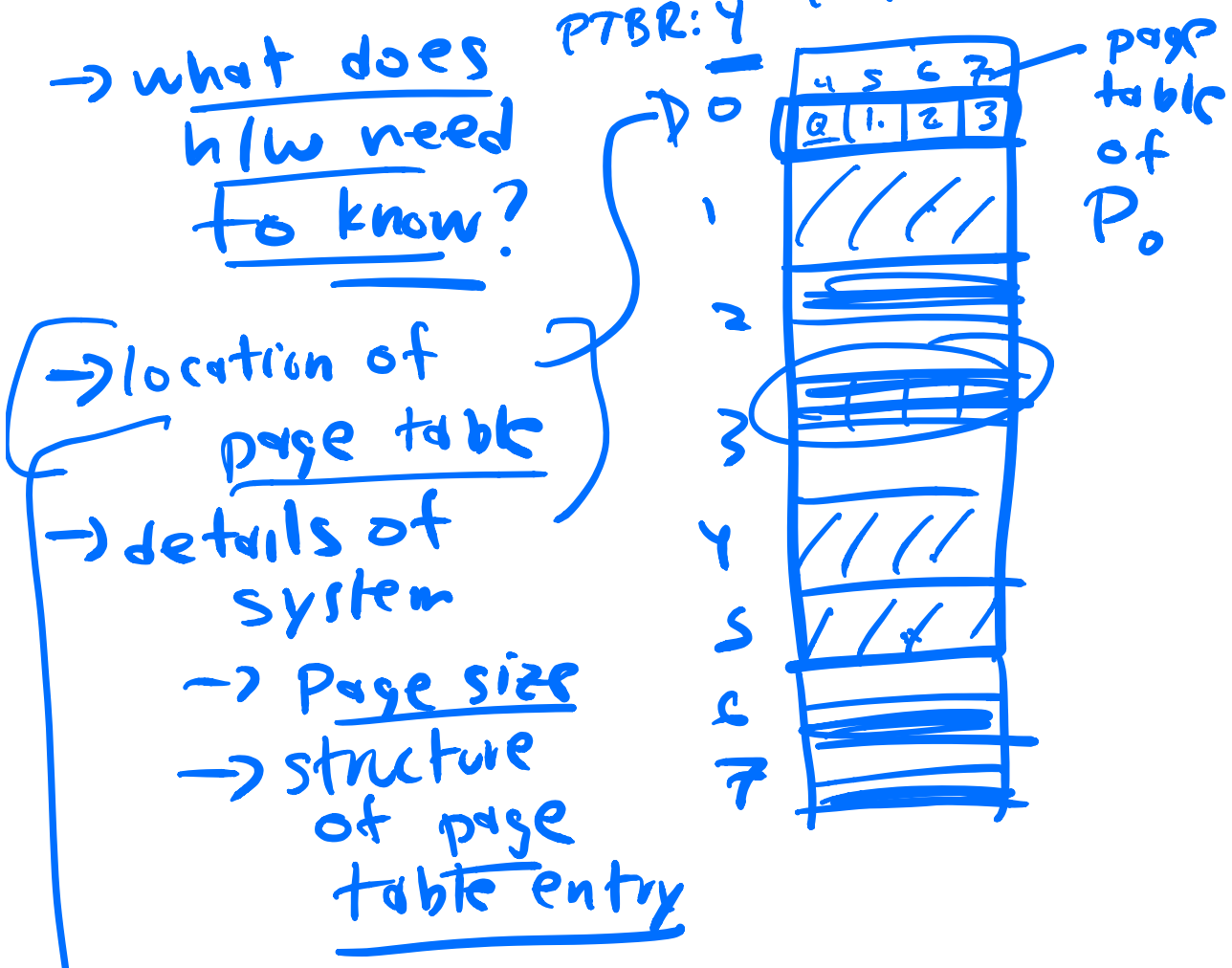
→ what does h/w need to know?

→ location of page table

→ details of system

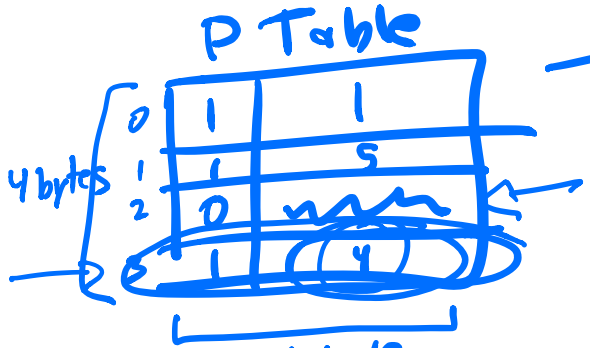
→ Page size

→ structure of page table entry



↳ per CPU register: hold phys. address of page table of currently running process

page table base register (PTBR)



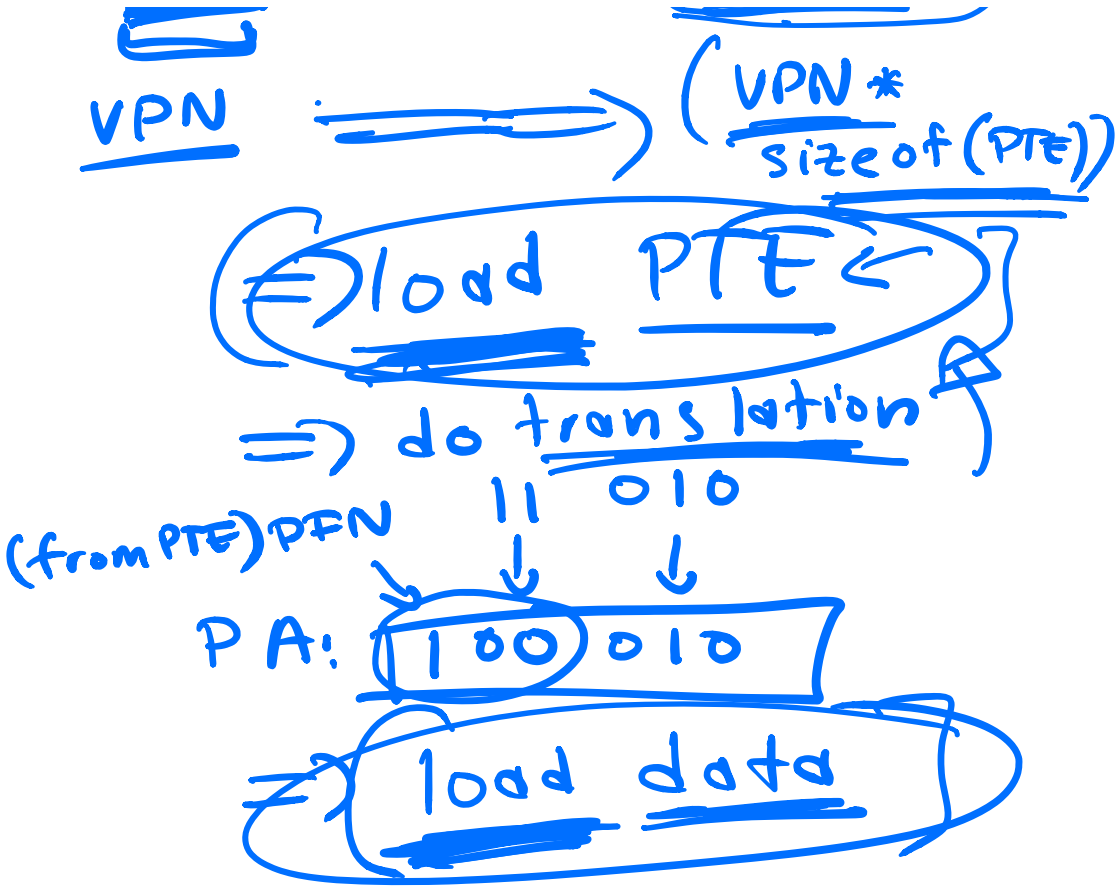
→ in memory: @ address ⇒ 4 (Phys. Addr)

set PTBR: 4 ← changes upon context switch

VA: need to translate:

Phys Addr of desired PTE

VA: 11 | 010 PA: PTBR +



PTBR details:

-> restricted (not all can update)

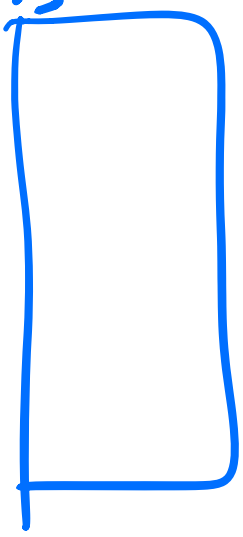
-> process table:
save, PTBR
restore

Problems:

→ Too slow (extra mem refs)

→ Too big

V.A.S



32-bit, 4KB page

⇒ ~1M entries

× 4 Bytes

⇒ 4 MB page table

⇒ 1000 procs

⇒ 4 GB

C code: ⇒ how machine works

```
int sum=0;
int i; // loop variable
for (i=0; i<2048; i++) //inc i
    sum += a[i]; // Summing it up!
```

4KB pages:

a[...]; how many pages are referenced?

=> [No discussion]

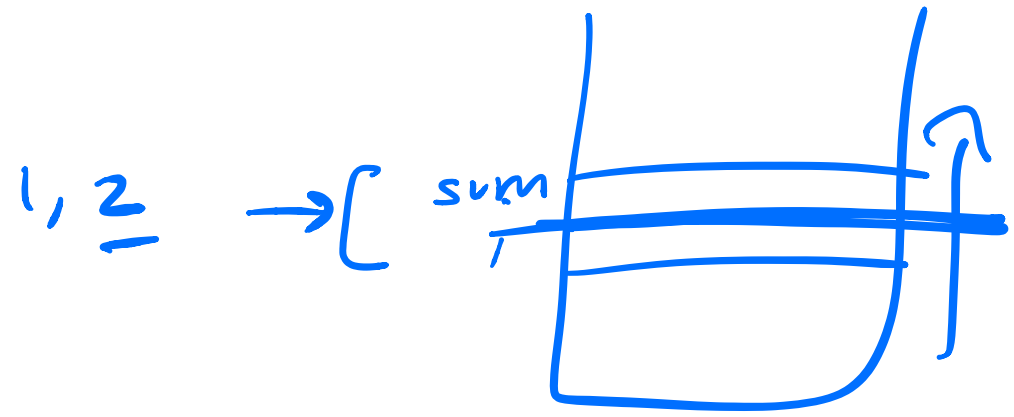
a[2,3]

Array : 8KB

4KB

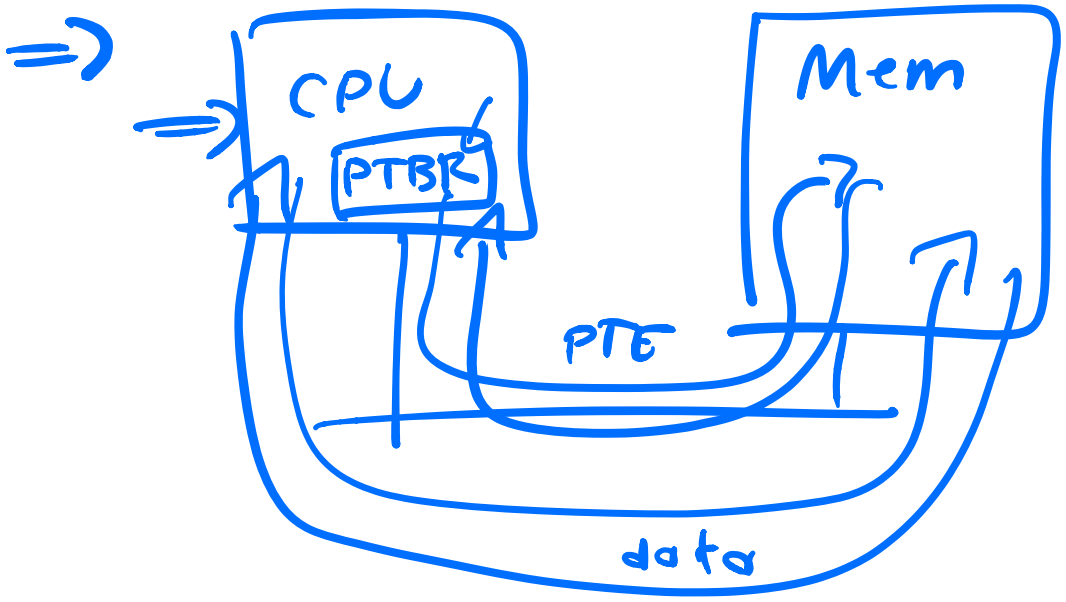
aligned?

i, sum : ? in memory



code:
5 instructions 1, 2 pages

Too slow : extra mem reference

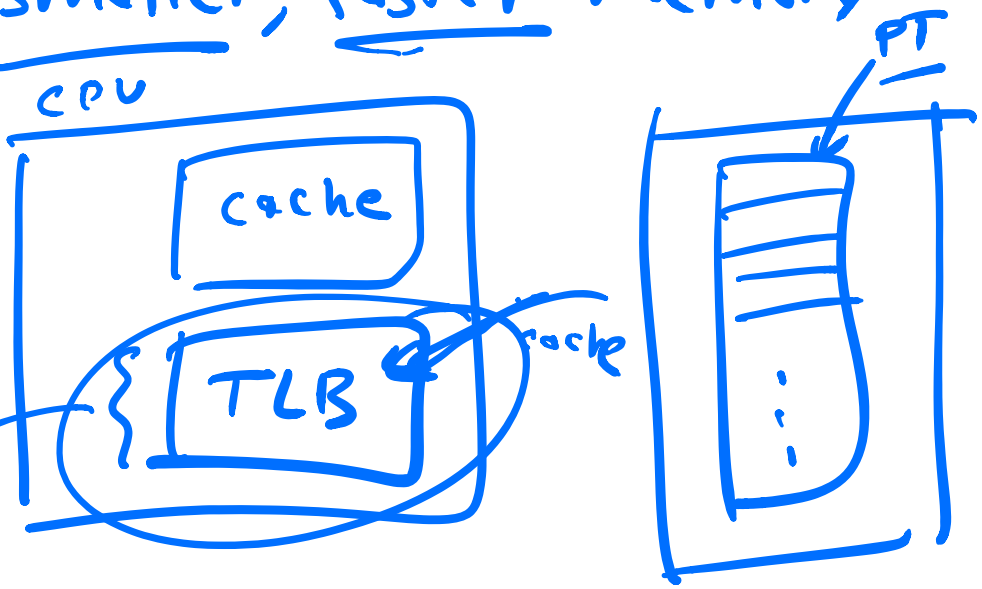


Cache : in CPU

=> Translation Lookaside Buffer (TLB)

=> "Address Translation Cache"

smaller, faster memory



[~10s, 100s of entries]

Translation:

VA: | VPN | offset |

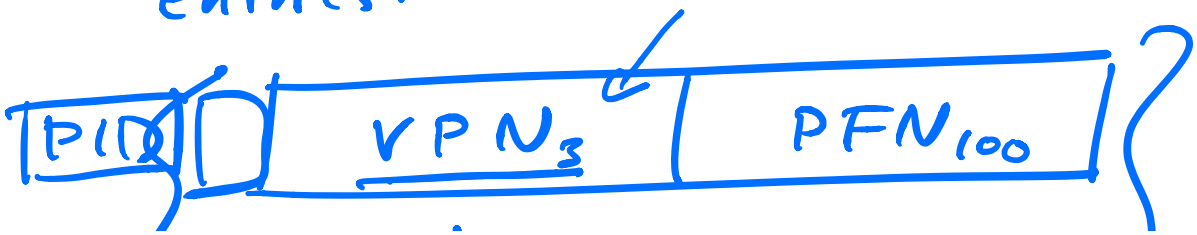
CPU: extract VPN from VA
lookup VPN in TLB
→ "hit": translation in cache
(good: fast)
PFN: => form PA
+ do mem ref

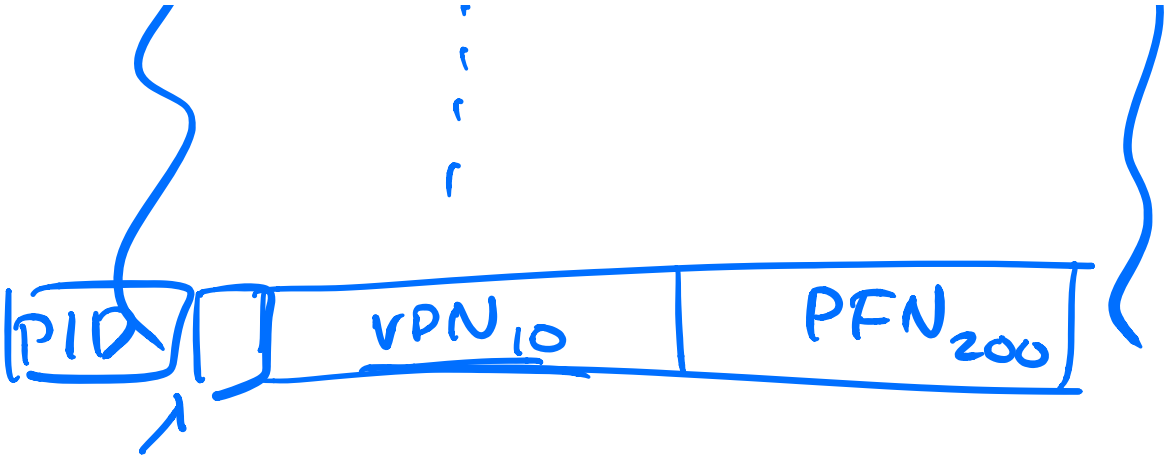
caches:
locality
→ spatial
→ temporal

→ "miss":
fetch PTE from memory (slow)
→ update TLB w/ contents
→ retry instruction (mem ref)

TLB: contents

entries:





valid bit : is this entry in TLB in use? (valid translation?)

context switch:

$P_1 \rightarrow P_2$

⇒ what to do?

flush TLB contents

(set all entries to not valid)

Different way to structure mem mgmt

support in H/W:

"so far H/W managed TLB"

new: s/w

"OS managed TLB"

idea:

TLB hit: still all h/w

TLB miss:

TLB miss exception

→ OS exception handler

⇒ consult page table + find translation

⇒ update TLB

priv.

⇒ ret from trap

=) ret from trap

→ H/W: retry inst (hit)

Freedom in OS:

=) h/w doesn't know details of page table (no PTBR)

=) H/W simpler