Today (3/20) "OS"

=) Bugs (Deadlock) } concurrency

break

=) Midterm Review
(cont. tomorrow)

Concurrency Bugs

Studies: bugs exist (still true)
concurrent bugs?

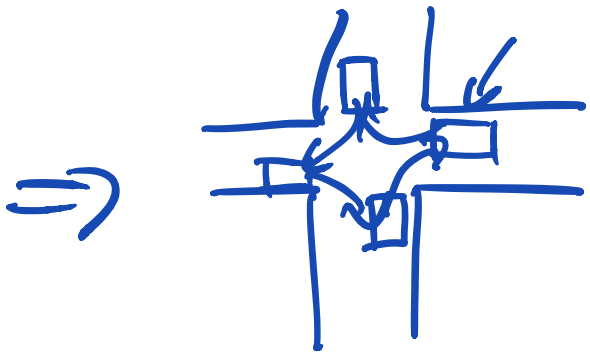
=) atomicity violations
(data races)
need locks

=) ordering violations
(control races)
{need condition variables}

Assume ordering: ?

{ event1 before event2 }

⇒ **deadlock**



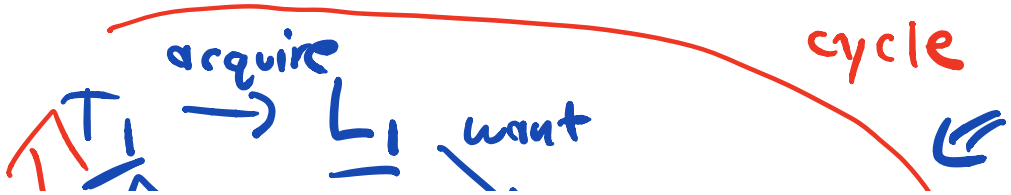
helpful: destroy car (breaking cycle)

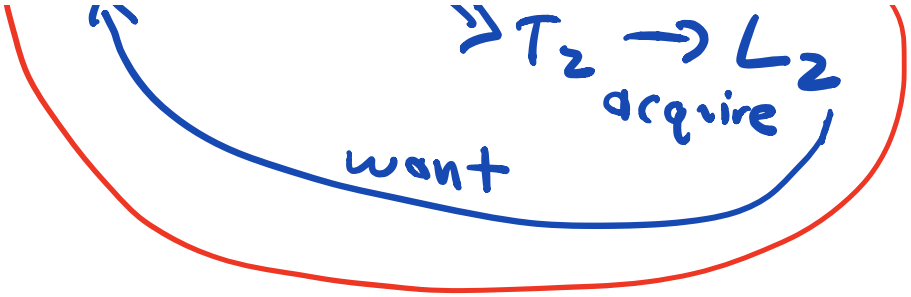
code: locks L_1, L_2

threads T_1, T_2

T_1	T_2
lock(L_1)	Lock(L_2)
lock (L_2)	lock(L_1)

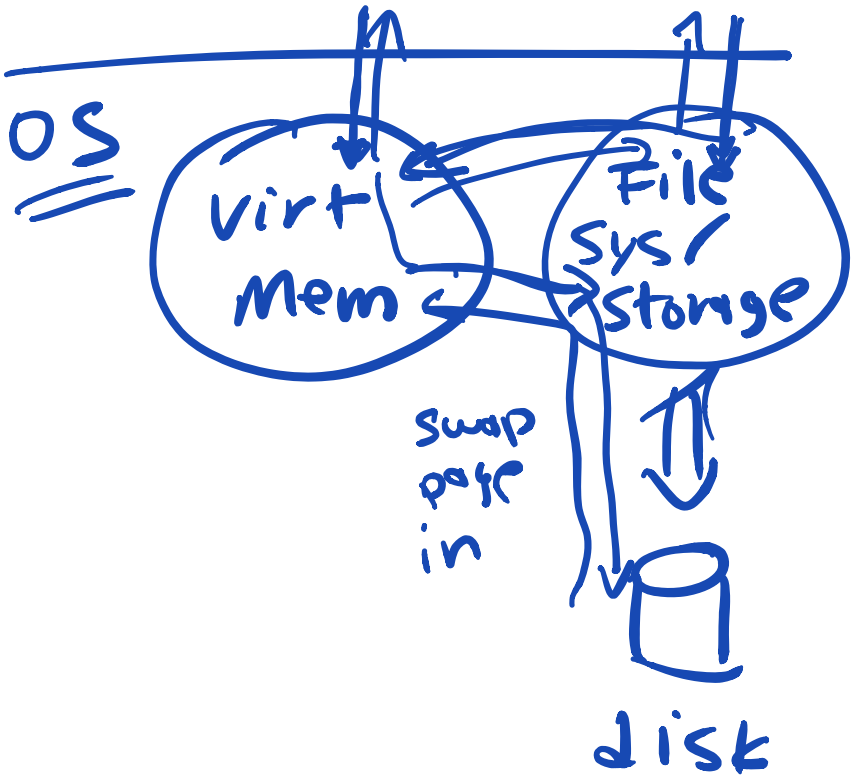
ACQUIRE





Examples: why deadlock occur?

=> Software has complex dependencies



example:

```
void function ( obj_1, obj_2 ) {
    // perf computation
       over both objects
       atomically
  { obj_1.lock();
    obj_2.lock();   } lock by address order
    ⋮
    // release locks here
}
```

T_1

$f(A, B)$

meta → A lock

B lock

→

T_2

$f(B, A)$

meta → B lock

A lock

→

Conditions:

Read: {"Soul of a New Machine"}

=> future: ethics

Conditions: [All 4 must hold for deadlock]

-> [mutual exclusion] => resource: exclusive access
another req -> delay

-> [hold-and-wait] => thread(s) holding, waiting

-> [no preemption] => holding resource, can't be taken away

-> [circular wait] => $T_1, \ldots, T_N$ exist cycle $T_1 \to T_2 \to \ldots \to T_N \to T_1$

Goal: Prevent (by breaking one conditions)

Prevent: circular wait

=> define lock ordering (most common solutions)

Prevent: hold + wait

=) lock acquisition lock
(meta lock)
hold meta lock
while acquiring
=) lock acquisition
atomic

Prevent: needing locks at all
(prevent mut. excl.)

=) "lock free" concurrent
("wait free")

idea: how to build
conc. data structs
w/o locks
(use powerful

low-level instructions

Example:

crit section: (data race)

~~lock~~

⇒ count = count + 1;

~~unlock~~

powerful inst: fetch-and-add

fetch_and_add (count, 1);
(atomic h/w inst.)

Example 2: list insertion

```
void list_insert (int value) {
                  +thread-safe
    node_t *n = malloc(      );
    n->value = value;
    n->next = head;   } needs to be atomic
    head = n;
}
```

use CAS

compare_and_

h/w instruction: swap

```
int CAS(unsigned *addr,
        unsigned expected,
        unsigned value) {
    if (*addr == expected) {
        *addr = value;
        return 1; // success
    }
    return 0; // failure
}

list_insert (int value) {
    node_t *n = malloc (sizeof(node_t));
    n->value = value;
    do {
        node_t * old_head = head;
        n->next = head;
    } while ( CAS (&head, old_head, n) == 0 );
}
```

---

potential problem: livelock (contention)

solution:

if fail: back off

wait (random_time)

try again

exponentially increasing, w/ each failure

Prevent: no preemption

pthread_mutex_lock()

can't preempt

other primitives:

pthread_mutex_trylock()

succeeds: grabs locks

fails: if lock held already

worried about deadlock:

lock(L_1)

lock(L_2)

} concern

top:
lock(L_1)
if (trylock(L_2) == FAIL)
unlock(L_1);
goto top;

← may want to wait

Avoid : scheduling

T_1 : L_1, L_2

T_2 : L_1, L_2

T_3 : L_3

scheduler:
run T_1 to completion,
then run T_2

Detect / Recover :

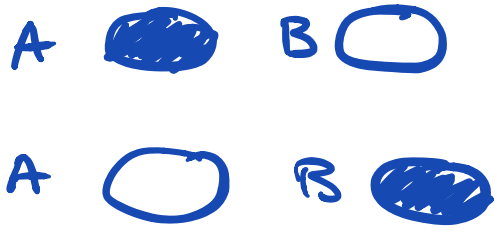
Dist. Databases

Review Topics: (today)

Old Questions (tomorrow)

Midterm:
Location: even → Chem
odd → Psych
student ID is

Format: set "easy grading" 7:15pm → [9:15pm]
=> bring pencil(s)



} mystery: theme

low: 50% !

[Long]

Strategy:

Shortest Question First

A

↓ ?

→ OS vs H/W: (hardware support) } somewhat

→ Scheduling: Multi-Level Feedback Queue — a lot

→ Page-tables + Address Translation — a lot

→ Multi-level Page Table — a lot, but...

→ TLBs — some

→ Conc + Semaphores — some → very few H/W primitives for locks

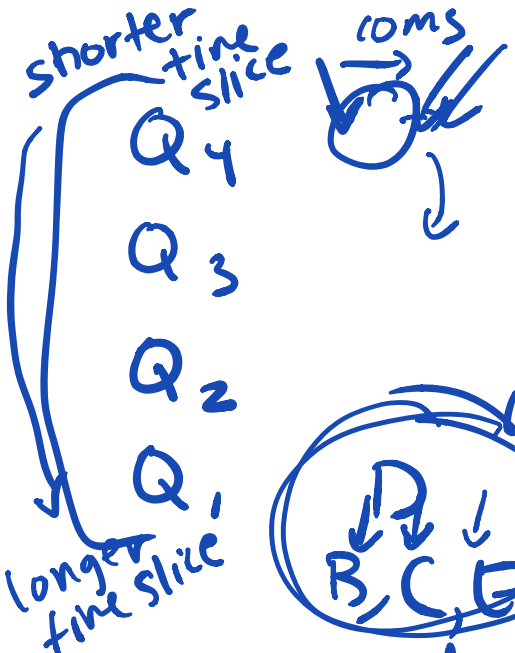
→ Producers / Consumers → C/Us
nobody, but ... few

Sched + MLFQ:

goals: → progress on long running
⇒ no starvation
⇒ want to (interactive) run short jobs quickly
(no knowledge about jobs)

(many) Queues:

- each queue has priority



Job (Process) is on one queues @ a time

→ Highest priority job always runs.
→ if 2 (or more) jobs have

same priority, round robin

→ Enter: at highest priority

→ if uses time quantum @ given priority level, move down one level queue

→ Periodically: boost priority of all jobs (to topmost level)

what defines time slice length?

→ min: timer interrupt every 1 ms

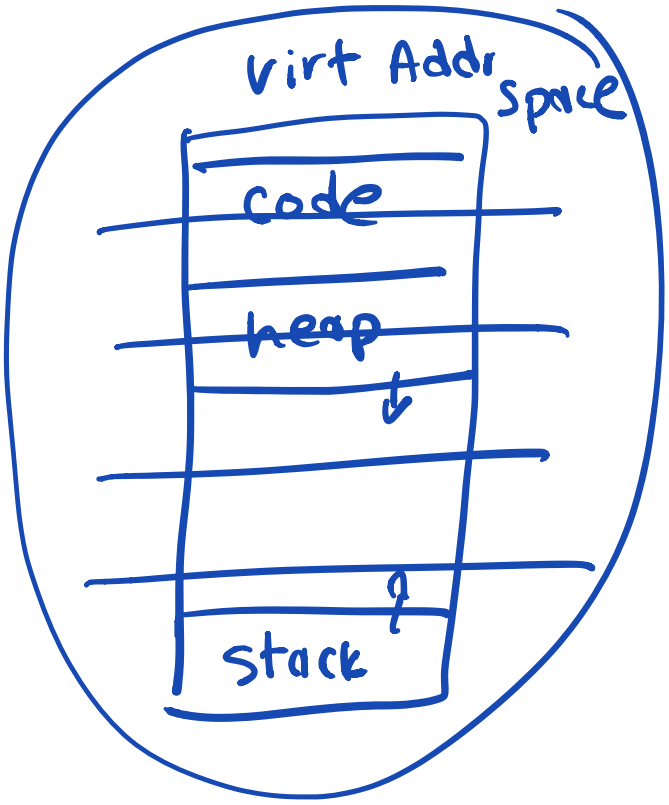
→ for each queue: define time slice is a multiple of timer

interrupt

Virtual Memory

Addr translation,
TLBs,

Page tables,
↓ linear (array)
↓ multi-level



running program:
process
private

int x; => address of x
loads, stores

x = x + 1; → instructions:

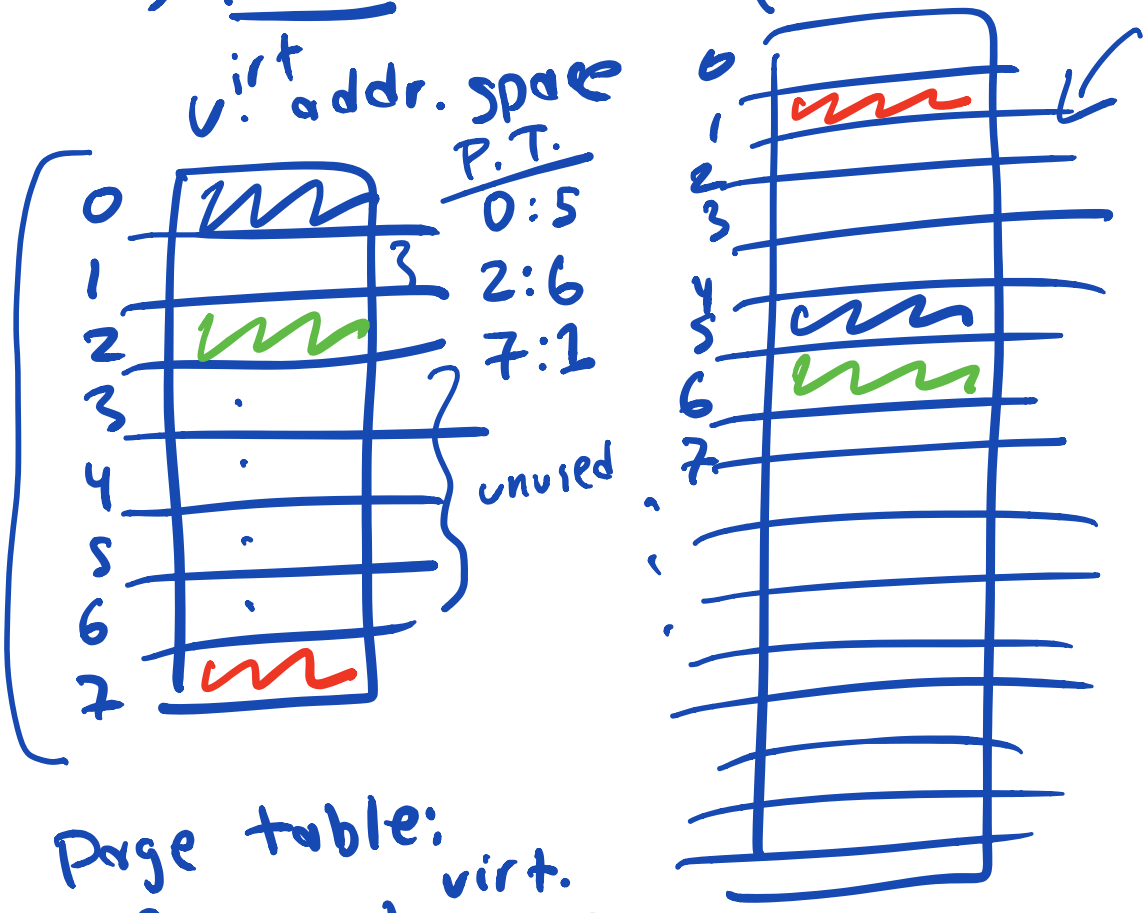
fetch

hardware support:

1) base/bounds

2) segmentation (3 base/bounds)

3) paging:

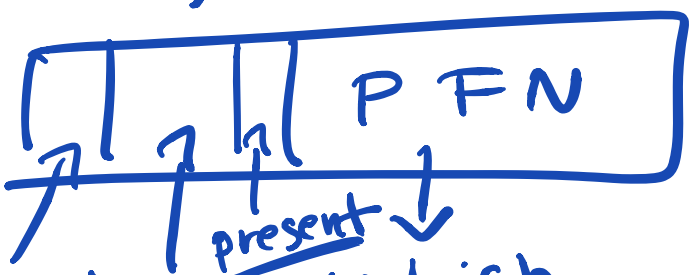


Page table: virt. for each page => where is this page in phys memory?

OR

not valid

Page Table Entry



(read-only, read-write)

1) Size of page tables => too big

2) Speed of paging / translation

linear (array):
one entry per virt page

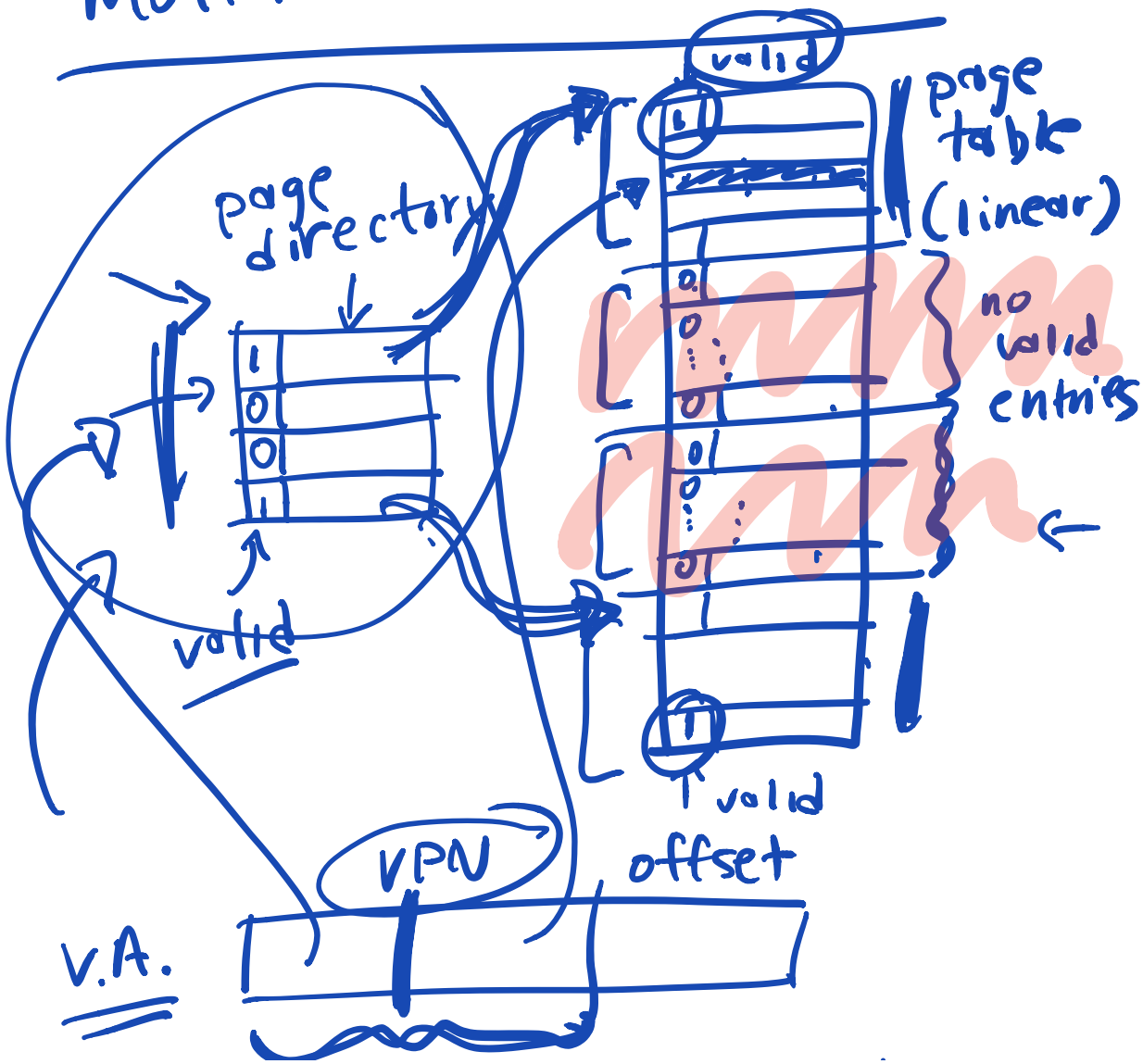
32-bit addr space
4KB page

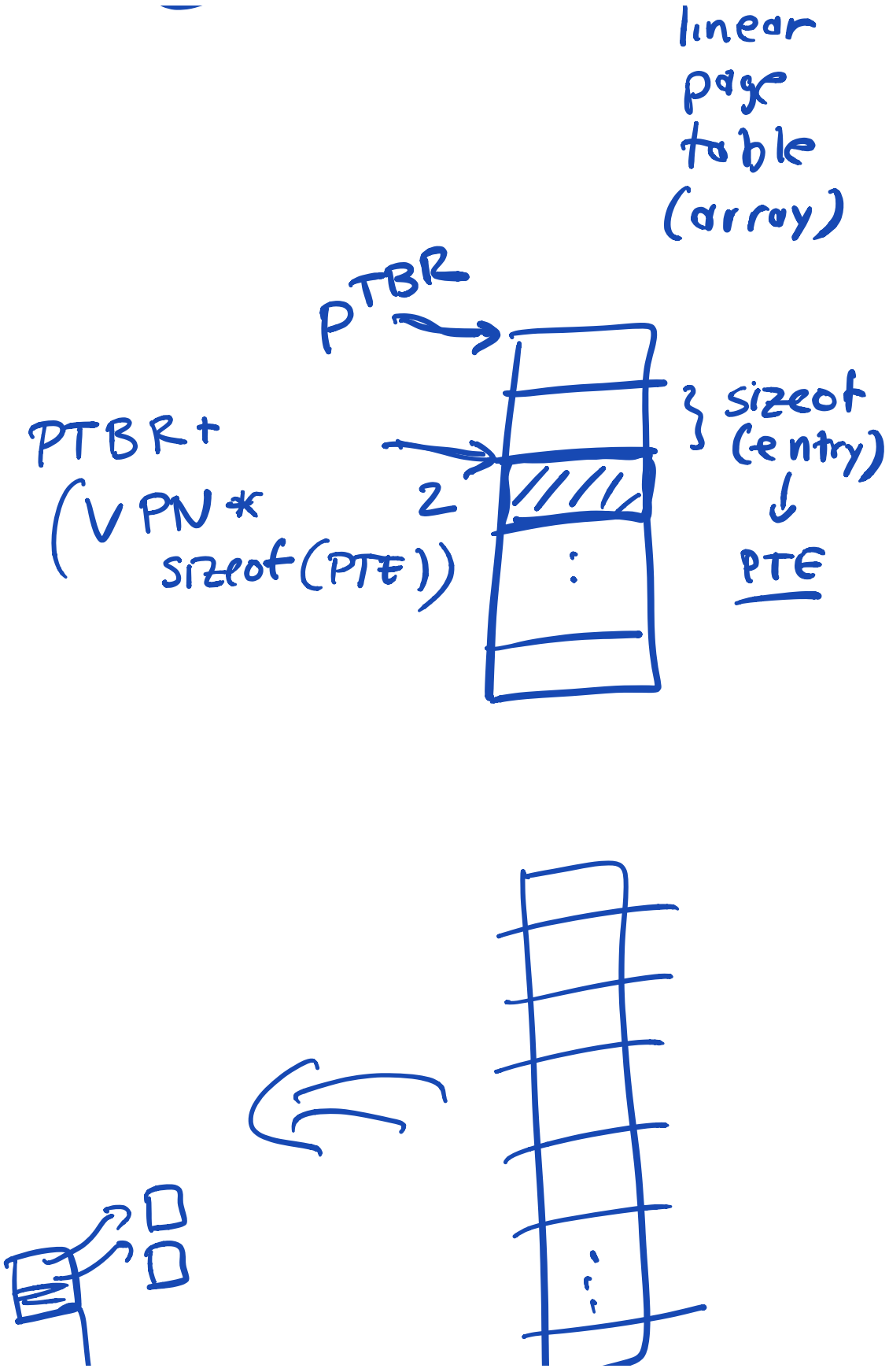
VPN 32-bit

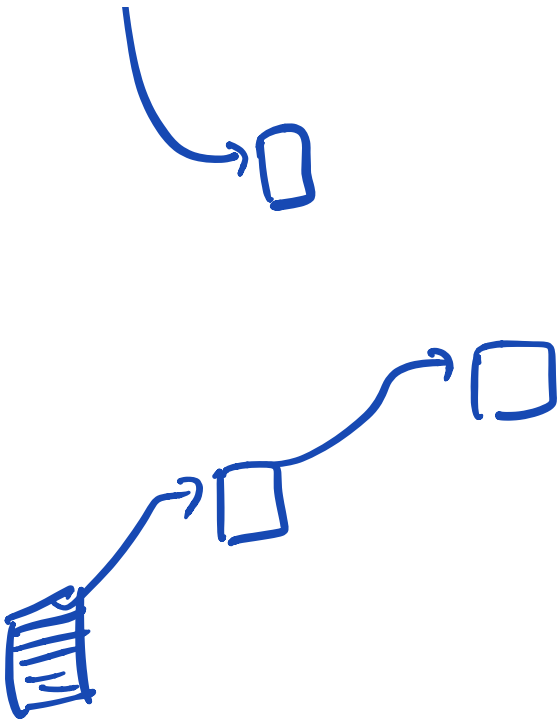
2^{20} virt pages

~ 1 Million entries

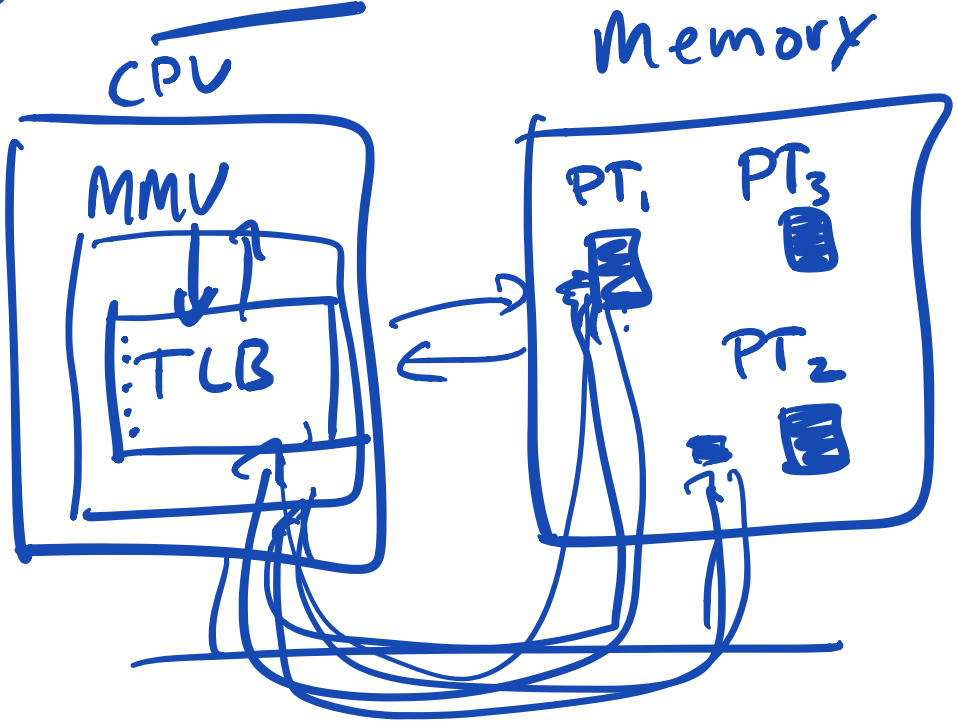
Multi-level page table:




linear
page
table
(array)
PTBR
PTBR+
(VPN*
sizeof(PTE))
2
sizeof
(entry)
PTE



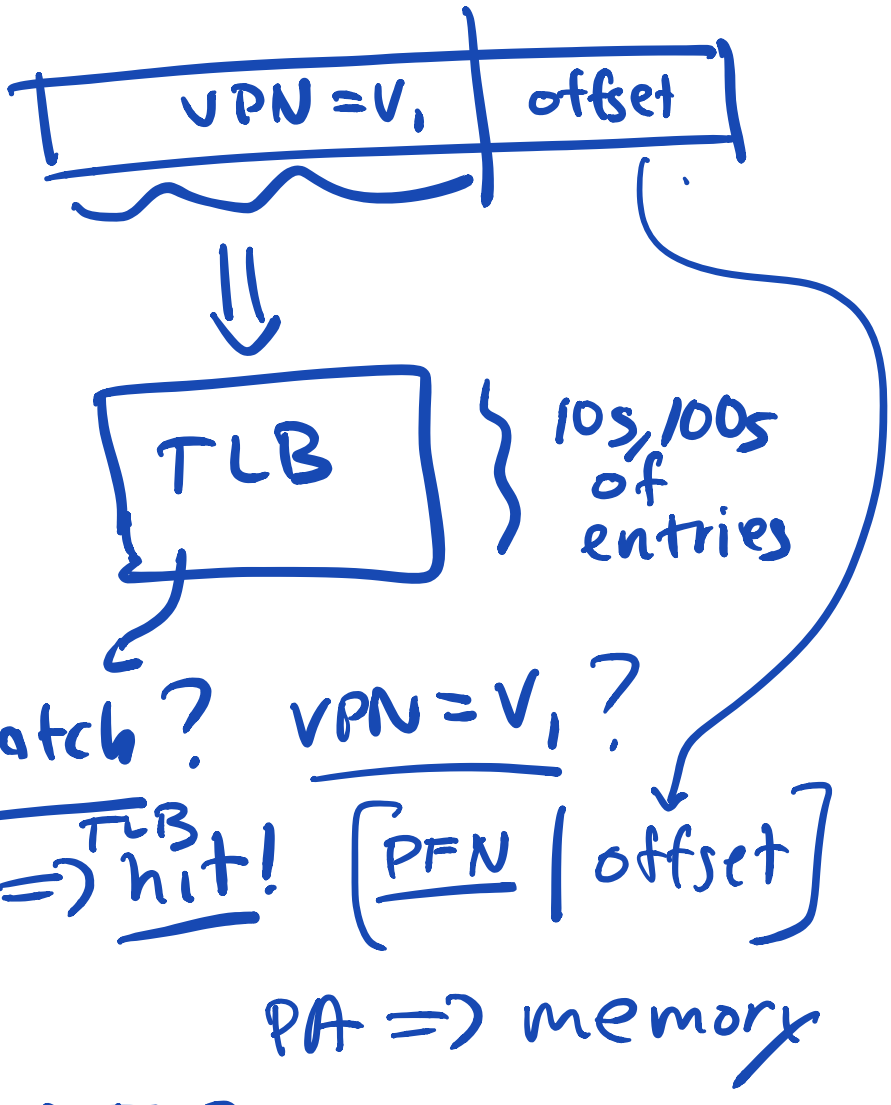
TLBs: Speed!



w/o TLB: 2 mem refs for every mem ref

TLB: "Address Translation Cache"

⇒ more complexity



⇒ TLB miss:

H/W based:

H/W look up entry in Page Table (pointed to by PTBR)

(CR3)

update TLB

(retry)