

Welcome back!

MEMORY: TLBS, SMALLER PAGETABLES

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

- Project 3 is due **Monday** → more code than P1 / P2
- Project 1 grades
↳ Check Piazza

AGENDA / LEARNING OUTCOMES

Memory virtualization

What are the challenges with paging ?

How we go about addressing them?

RECAP

PAGING

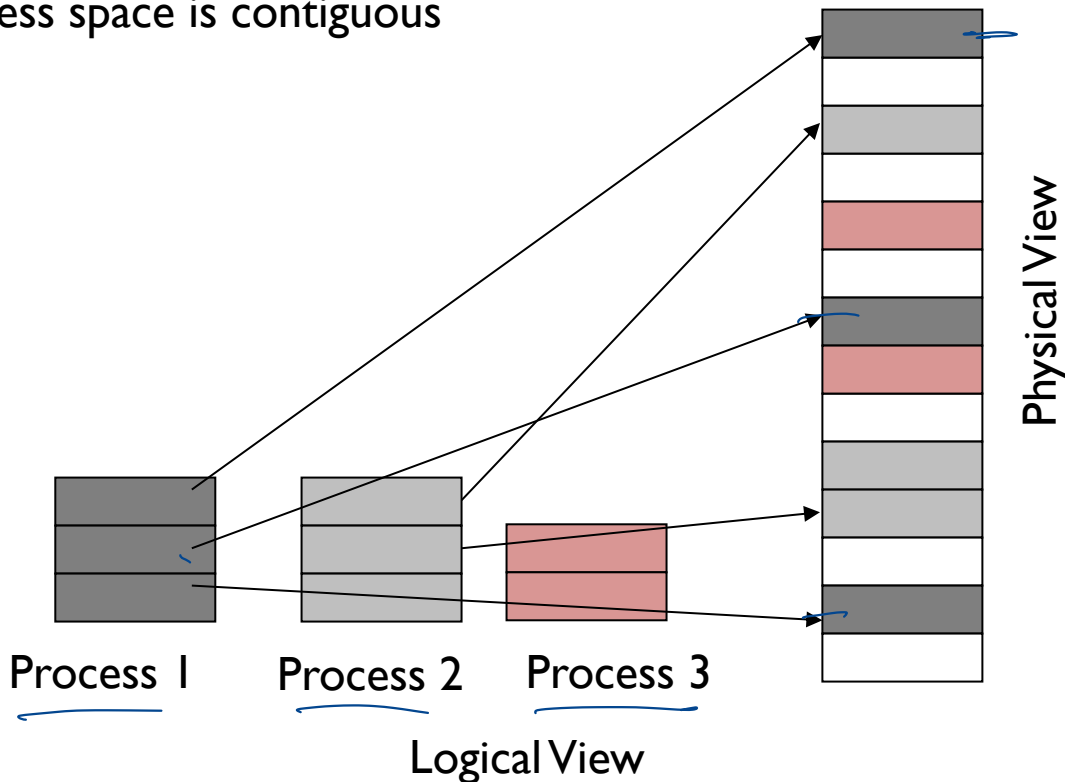
1 GB Phys mem
4 KB Pages

Goal: Eliminate requirement that address space is contiguous

Idea:
Divide address spaces and physical memory into fixed-sized pages

Example page size: 4KB

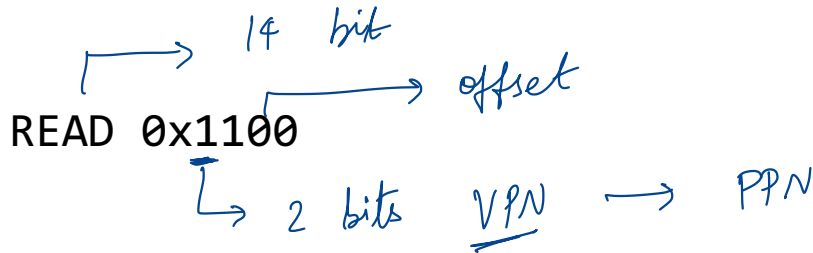
Page table



PAGING TRANSLATION STEPS

For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory $\rightarrow 0x5004$
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory $0x0100$



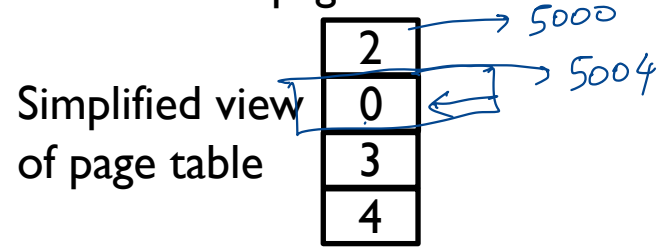
mapping VPN to PFN

14 bit addresses

Assume PT is at phys addr $0x5000$

Assume PTE's are 4 bytes

Assume 4KB pages – 12 bit offset



$$0x1 \quad 100$$
$$\underline{0x0} \quad \underline{100} = 0x0100$$

PROS/CONS OF PAGING

Pros

No external fragmentation

- Any page can be placed in any frame in physical memory

Fast to allocate and free

- Alloc: No searching for suitable free space
- Free: Doesn't have to coalesce with adjacent free space

Cons

Additional memory reference

- MMU stores only base address of page table

Storage for page tables may be substantial

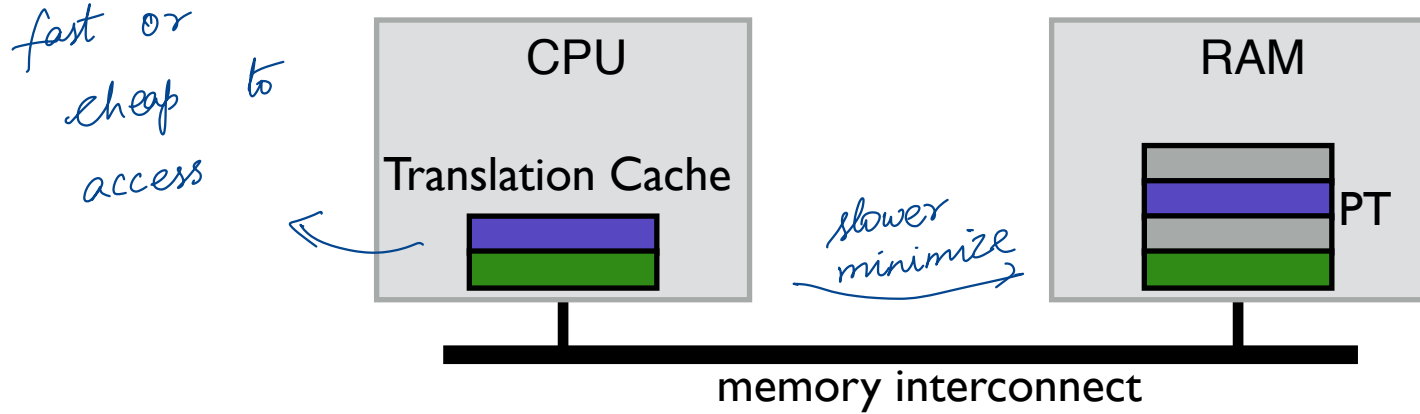
- Simple page table: Requires PTE for all pages in address space
- Entry needed even if page not allocated ?

every translation

↳ access PTE

↳ desired address

STRATEGY: CACHE PAGE TRANSLATIONS



TLB Entry

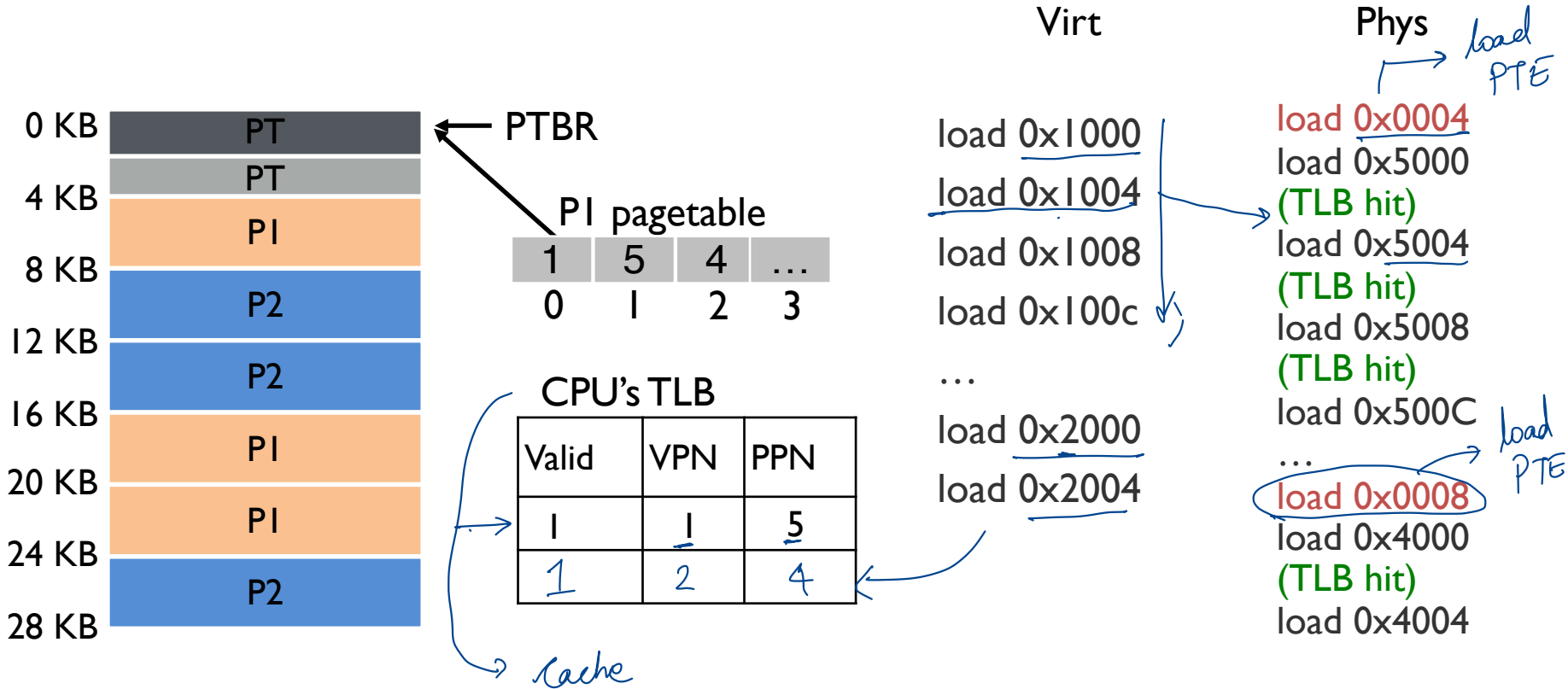
Tag (<u>virtual page number</u>)	<u>Physical page number</u> (page table entry)
------------------------------------	--

Fully associative

Any given translation can be anywhere in the TLB

Hardware will search the entire TLB in parallel

TLB ACCESSES: SEQUENTIAL EXAMPLE



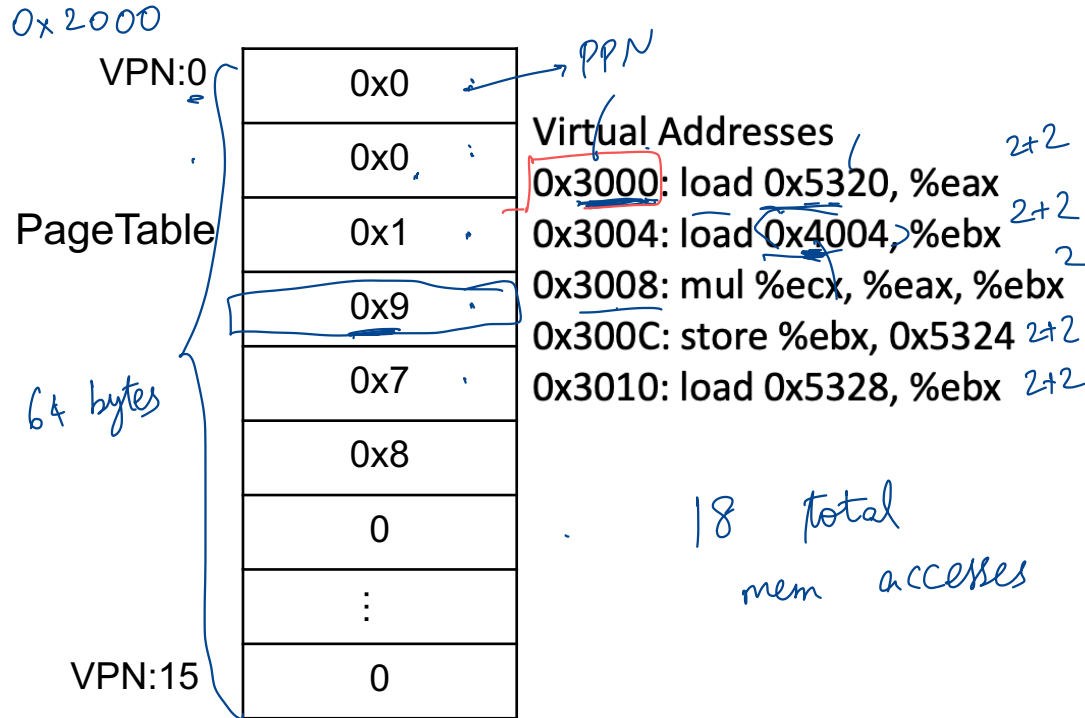
QUIZ 10: TLBS

<https://tinyurl.com/cs537-sp23-quiz10>



Consider a processor with 16-bit address space and 4kB page size. Assume Page Table is at 0x2000 and each PTE is of 4 bytes.

12 bits offset
4 bits VPN



Memory accesses

Fetch 0x3000
 ↳ Translate 0x3000
 $0x2000 + 3 \times 4 = 0x200C \rightarrow$ Ans
 PA is 0x9000

Fetch 0x5320 → 2 mem accesses
 8th mem access = 0x7004

Total

VPN:0

0x0
0x0
0x1
0x9
0x7
0x8
...

PageTable

Virtual Addresses

0x3000: load 0x5320, %eax $2 + 2$

0x3004: load 0x4004, %ebx $1 + 2$

0x3008: mul %ecx, %eax, %ebx 1

0x300C: store %ebx, 0x5324 $1 + 1$

0x3010: load 0x5328, %ebx $1 + 1$

Memory accesses

Total 12

accesses

⇒ TLB saved

6 memory

accesses

TLB

Valid	VPN	PPN
0 1	2 3	8 9
0 1	7 5	23 8
0 1	2 4	5 7
0	3	2
0	1	89

TLB: POLICIES

How to we replace entries in the TLB?

How do we handle context switches? → *what happens to TLB?*

PERFORMANCE OF TLB?

$$5 / 10 = 0.5$$

Page size = 4 kB
Int = 4 bytes
1 page = 1024 ints

Miss rate of TLB: $\frac{\# \text{TLB misses}}{\# \text{TLB lookups}}$


TLB lookups? number of accesses to a = 2048

TLB misses?

= number of unique pages accessed

$$= 2$$

```
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

 2 pages

Miss rate?

$$2 / 2048$$

Would hit rate get better or worse
with smaller pages?

Hit rate?

$$1 - \text{miss rate}$$

WORKLOAD ACCESS PATTERNS

Workload A

```
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

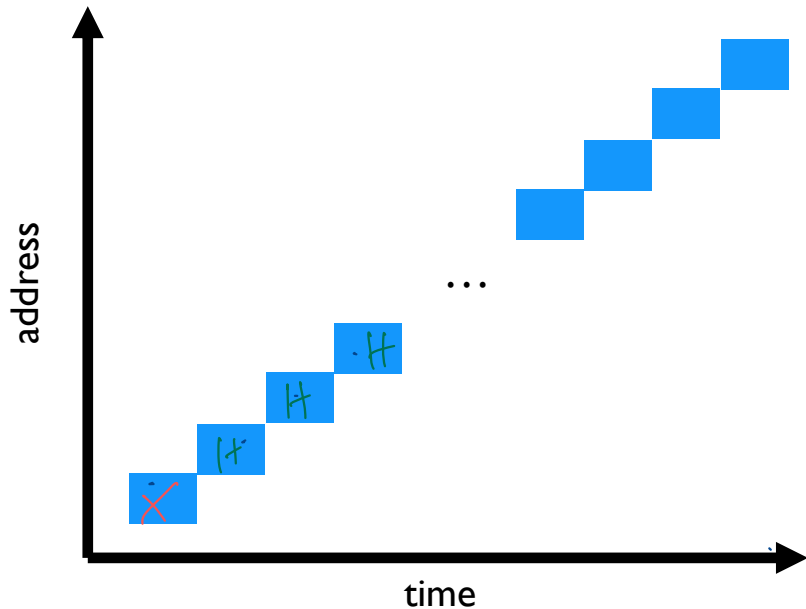
Workload B

```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
```

WORKLOAD ACCESS PATTERNS

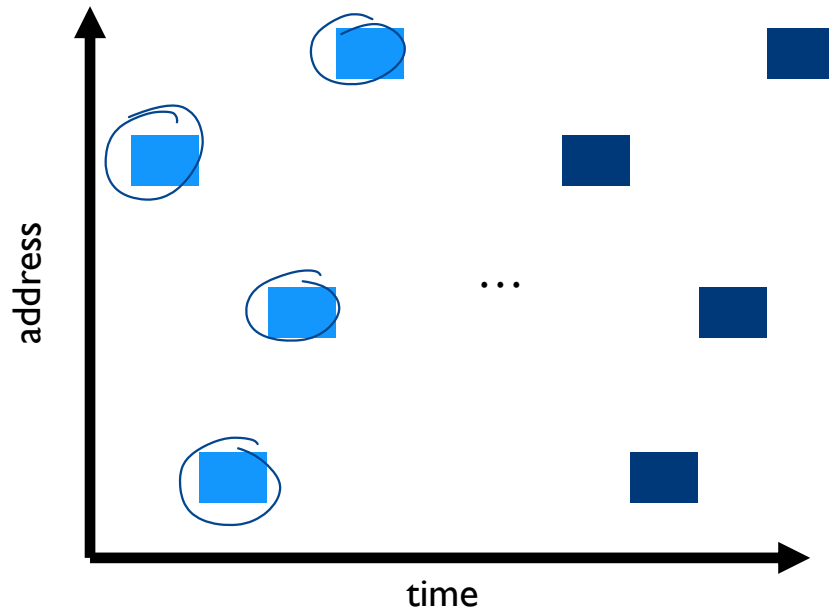
Spatial Locality

Sequential Accesses



Temporal Locality

Repeated Random Accesses



TLB REPLACEMENT POLICIES

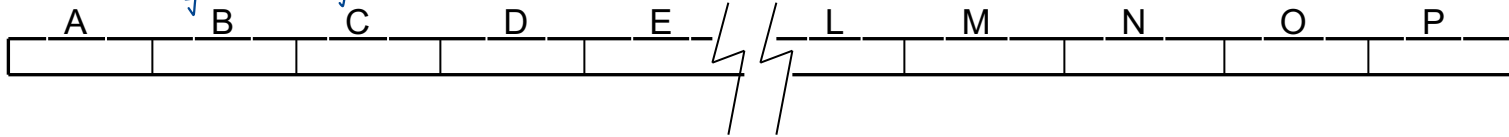
LRU: evict Least-Recently Used TLB slot when needed

Fixed size TLB

VPN	PPN	Last used
3	20	
⋮		
⋮		
⋮		

timestamp

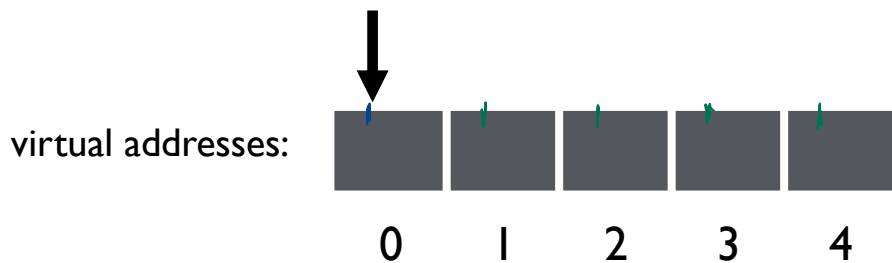
Find oldest timestamp & replace that entry



LRU TROUBLES

size of TLB = 4

0x001
0x101
0x201
0x301
0x401
0x002
0x102
0x202
⋮



Valid	Virt	Phys
0	4 1 0	?
0	1 1	?
0	1 2	?
0	1 3	?

Workload repeatedly accesses same offset (0x01) across 5 pages (strided access), but only 4 TLB entries

What will TLB contents be over time?

How will TLB perform?

TLB REPLACEMENT POLICIES

LRU: evict Least-Recently Used TLB slot when needed

Random: Evict randomly chosen entry

Sometimes random is better than a “smart” policy!

CONTEXT SWITCHES

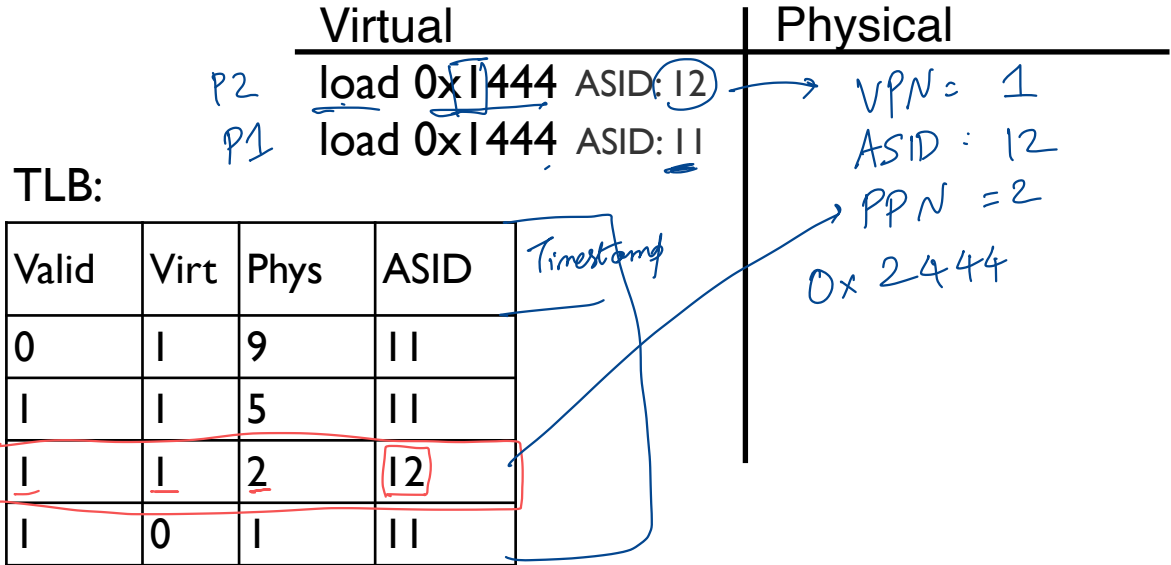
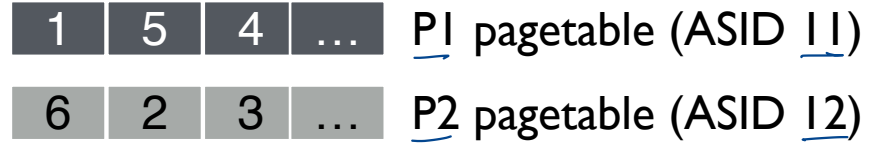
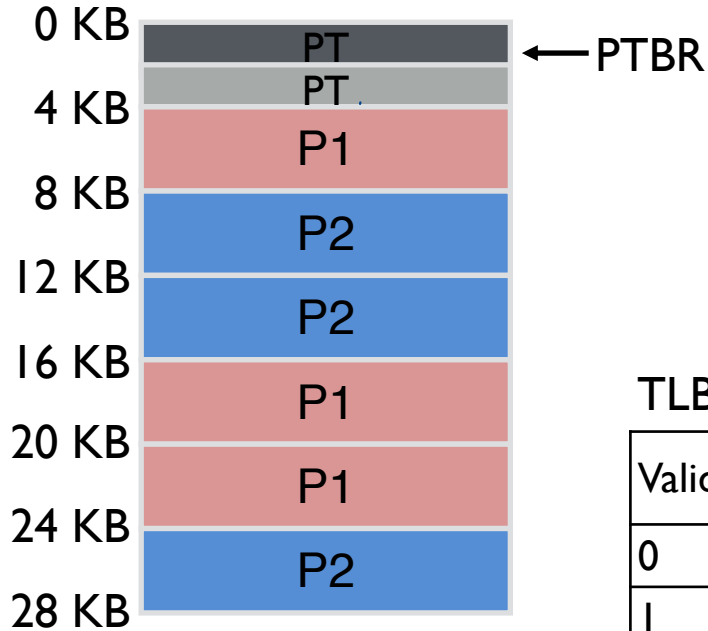
What happens if a process uses cached TLB entries from another process?

1. Flush TLB on each switch
Costly → lose all recently cached translations

*safe solution
because no stale TLB entries
from previous process*

2. Track which entries are for which process
 - Address Space Identifier
 - Tag each TLB entry with an 8-bit ASID

TLB EXAMPLE WITH ASID



TLB PERFORMANCE

Context switches are expensive

Even with ASID, other processes “pollute” TLB

Architectures can have multiple TLBs

- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 TLB for “super pages”

stack / heap access

code sections

HW AND OS ROLES

If H/W handles TLB Miss

CPU must know where pagetables are

- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW “walks” the pagetable and fills TLB

→ OS is not involved.

HW knows layout of
page table

If OS handles TLB Miss:

“Software-managed TLB”

- CPU traps into OS upon TLB miss.
- OS interprets pagetables as it chooses
- Modify TLB entries with privileged instruction

invokes a handler

for TLB miss.

flexibility

Costly

TLB SUMMARY

Pages are great, but accessing page tables for every memory access is slow

Cache recent page translations → TLB

- MMU performs TLB lookup on every memory access

TLB performance depends strongly on workload



replacement policy

- Sequential workloads perform well
- Workloads with temporal locality can perform well

In different systems, hardware or OS handles TLB misses

TLBs increase cost of context switches

- Flush TLB on every context switch
- Add ASID to every TLB entry

QUIZ 11: MORE TLBS

<https://tinyurl.com/cs537-sp23-quiz11>



1. What problem(s) can be solved by using ASIDs ?

TLBs need to be flushed across context switches

2. For a hardware-managed TLB miss, which of the following statements are true?

HW knows where page tables
OS plays no role in TLB miss

3. For a software-managed TLB miss, which of the following statements are true?

HW raises exception on a TLB miss
OS moves entries in and out

DISADVANTAGES OF PAGING

Additional memory reference to page table → Very inefficient

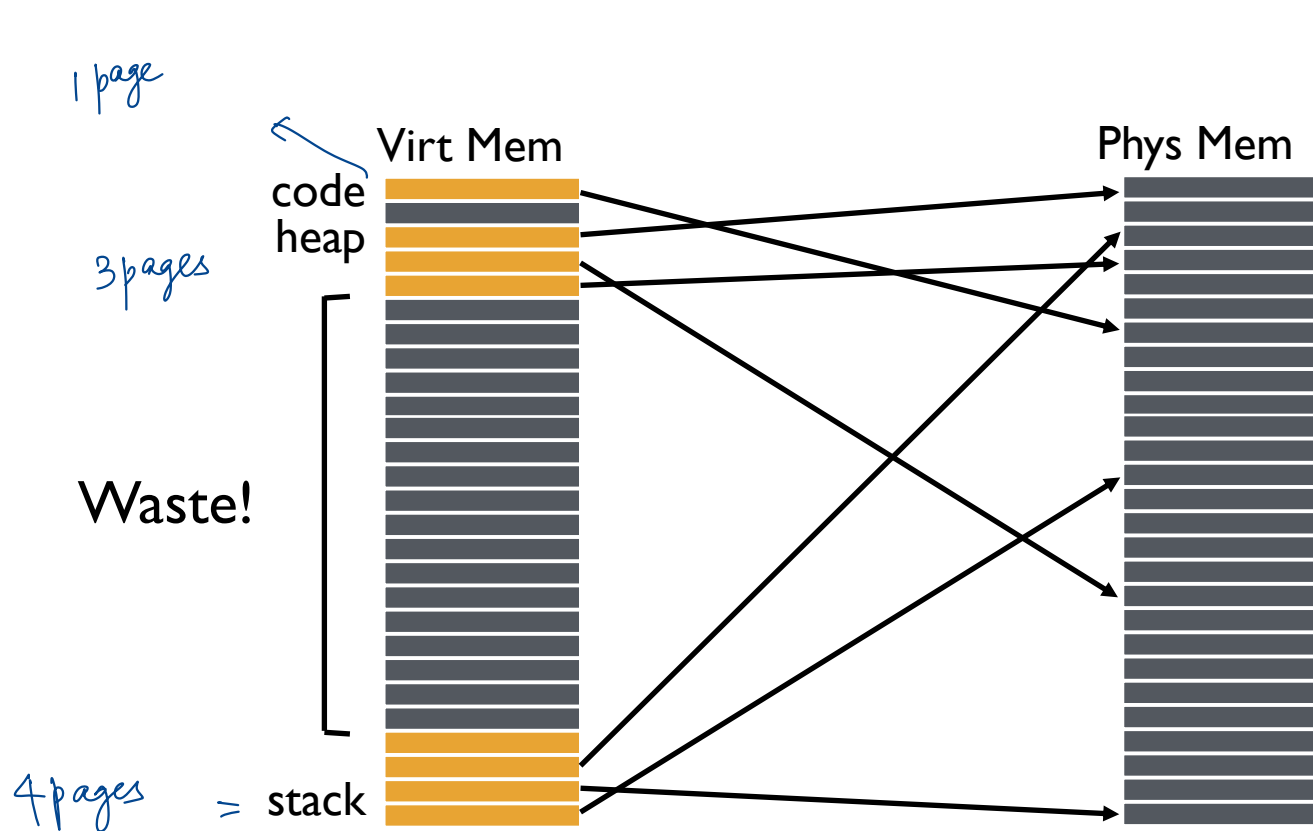
- Page table must be stored in memory
- MMU stores only base address of page table

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
Entry needed even if page not allocated ?

SMALLER PAGE TABLES

WHY ARE PAGE TABLES SO LARGE?



linear Page
Tables =
number of virtual
pages
× PTE size
= 1M virtual
pages
× 4 bytes
= 4MB page table

MANY INVALID PT ENTRIES

	PFN	valid	prot
	10		r-x
	-	0	-
	23		rw-
	-	0	-
	-	0	-
	-	0	-
	-	0	-
	...many more invalid...		
	-	0	-
	-	0	-
	-	0	-
	-	0	-
	28		rw-
	4		rw-

how to avoid storing these?

Can we use a better data structure?

AVOID SIMPLE LINEAR PAGE TABLES?

Use more complex page tables, instead of just big array

Any data structure is possible with software-managed TLB

- Hardware looks for vpn in TLB on every memory access
- If TLB does not contain vpn, TLB miss
 - Trap into OS and let OS find vpn->ppn translation
 - OS notifies TLB of vpn->ppn for future accesses

OTHER APPROACHES

1. Multi-level Pagetables → *HW friendly*
 - Page the page tables
 - Page the pagetables of page tables...
2. Inverted Pagetables → *Software managed TLBs*

MULTILEVEL PAGE TABLES

Goal: Allow page table to be allocated non-contiguously

Idea: Page the page tables

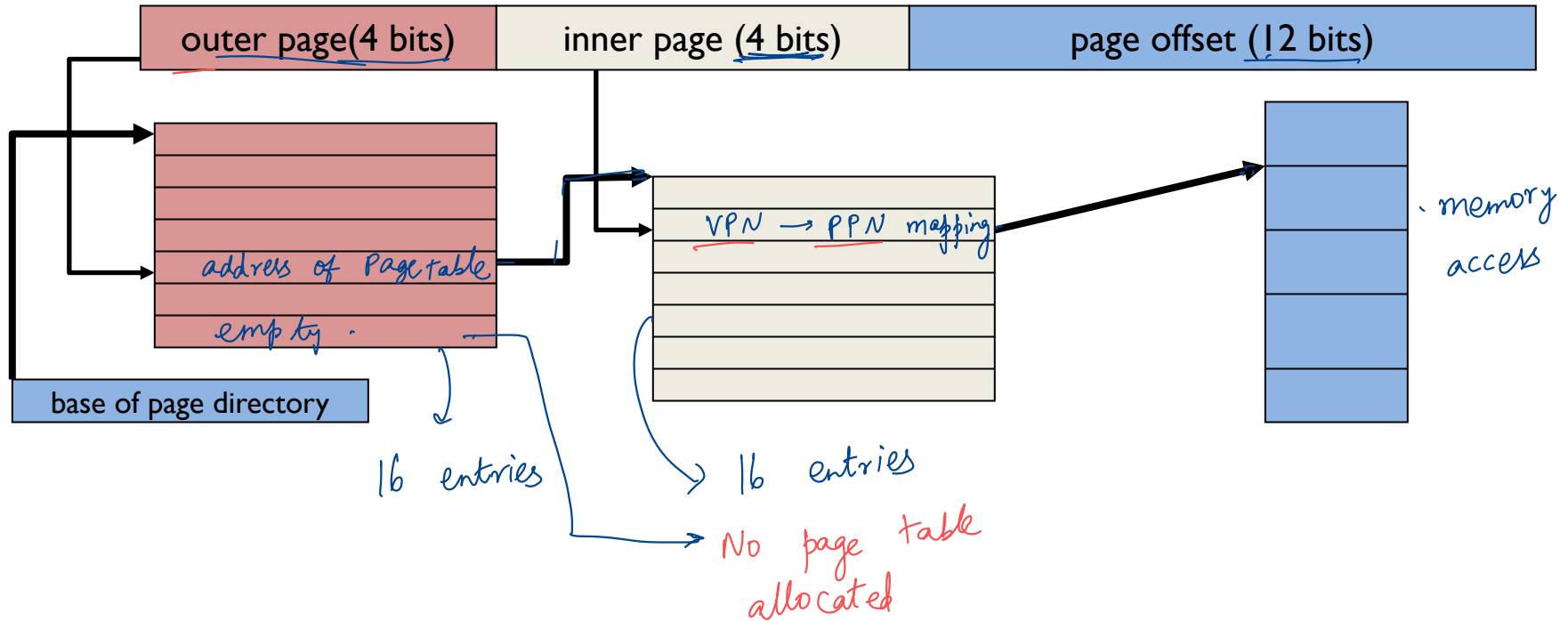
- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)



only allocate
page tables when pages are used

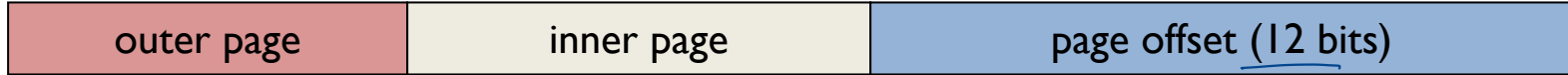
MULTILEVEL PAGE TABLES

20-bit address:



ADDRESS FORMAT FOR MULTILEVEL PAGING

30-bit address:



How should logical address be structured? How many bits for each paging level?

Goal?

– Each inner page table fits within a page

– PTE size * number PTE = page size

Assume PTE size = 4 bytes

Page size = 2^{12} bytes = 4KB

→ # bits for selecting inner page = 10

PTE = 4 bytes
page size = 4 KB
Inner page table = 1024 entries

Remaining bits for outer page:

– $30 - \underline{12} - \underline{10} = \underline{8}$ bits

MULTILEVEL TRANSLATION EXAMPLE

page directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

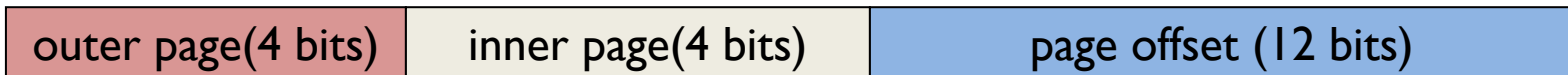
PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

translate 0x01ABC

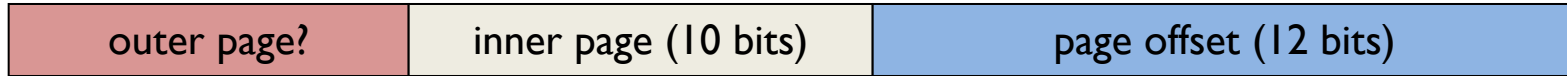
20-bit address:



PROBLEM WITH 2 LEVELS?

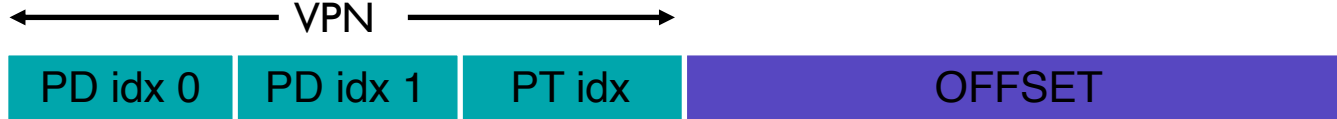
Problem: page directories (outer level) may not fit in a page

64-bit address:



Solution:

- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.



How large is virtual address space with 4 KB pages, 4 byte PTEs,
(each page table fits in page)

4KB / 4 bytes → 1K entries per level

1 level:

2 levels:

3 levels:

FULL SYSTEM WITH TLBS

On TLB miss: lookups with more levels more expensive

Assume 3-level page table

Assume 256-byte pages

Assume 16-bit addresses

Assume ASID of current process is 211

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl \$0x3, %edi

(c) 0x0519: movl %edi, 0xFF10

INVERTED PAGE TABLE

Only store entries for virtual pages w/ valid physical mappings

Naïve approach:

Search through data structure $\langle \text{ppn}, \text{vpn} + \text{asid} \rangle$ to find match

Too much time to search entire table

Better:

Find possible matches entries by hashing $\text{vpn} + \text{asid}$

Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB

QUIZ 12

<https://tinyurl.com/cs537-sp23-quiz12>



Consider a virtual address space of 16KB with 64-byte pages.

1. How many bits will we have in our virtual address for this address space?
2. What is the total number of entries in the Linear Page Table for such an address space?
3. Consider a two-level page table now with a page directory. How many bits will be used to select the inner page assuming PTE size = 4 bytes?

QUIZ12

page directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

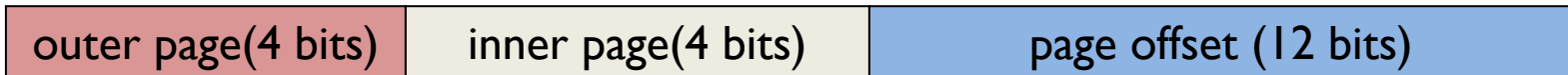
PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

translate 0xFEED0

20-bit address:



SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure

- Inverted page tables (hashing)

If Hardware handles TLB miss, page tables must follow specific format

- Multi-level page tables used in x86 architecture
- Each page table fits within a page

NEXT STEPS

Project 3: In progress

Next class: Swapping!