MEMORY: TLBS, SMALLER PAGETABLES

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

- Project 2a is due Friday
- Project 1b grades this week

- Midterm makeup emails

AGENDA / LEARNING OUTCOMES

Memory virtualization

- What are the challenges with paging ?
- How we go about addressing them?

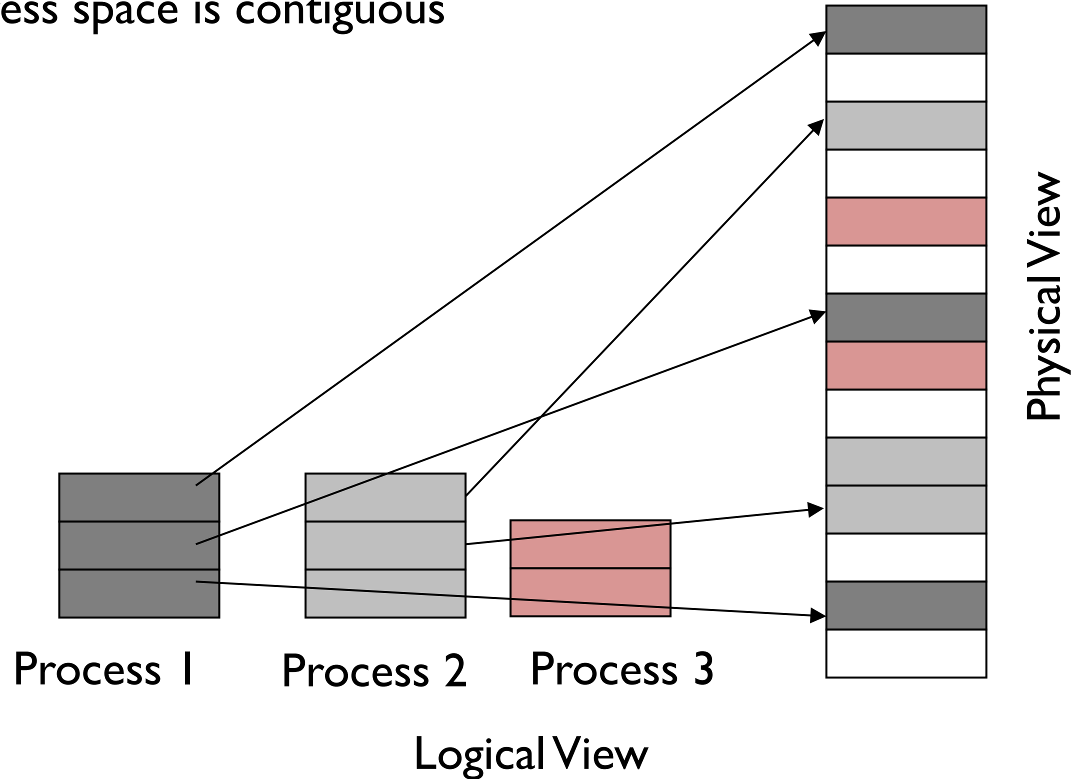
RECAP

PAGING

Goal: Eliminate requirement that address space is contiguous

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

Example page size: 4KB



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
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory

14 bit addresses

Assume PT is at phys addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages – 12 bit offset

Simplified view of page table

2
0
80
99

```
READ 0x1100
```

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 ?

EXAMPLE: ARRAY ITERATOR

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

Assume 'a' starts at 0x1000
Ignore instruction fetches and access to 'i'

What virtual addresses?

load 0x1000

load 0x1004

load 0x1008

What physical addresses?

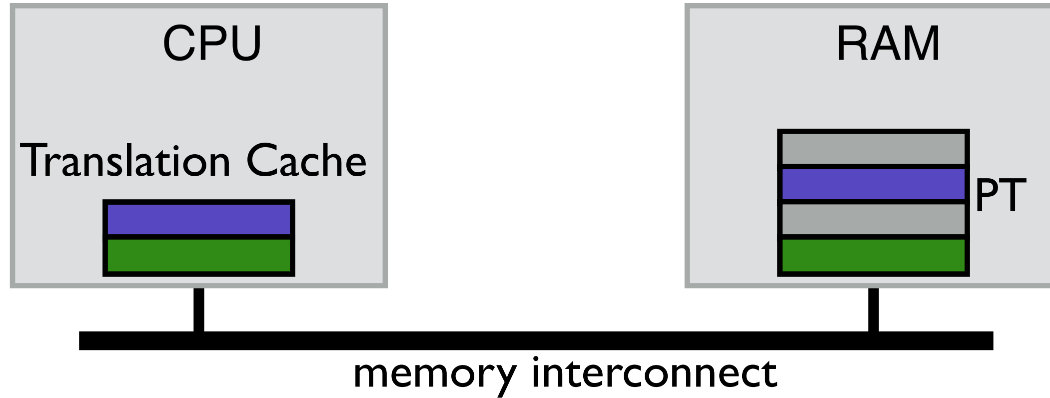
load 0x0004
load 0x5000
load 0x0004
load 0x5004
load 0x0004
load 0x5008

What can you infer?

PTBR: 0x0000; PTE 4 bytes each

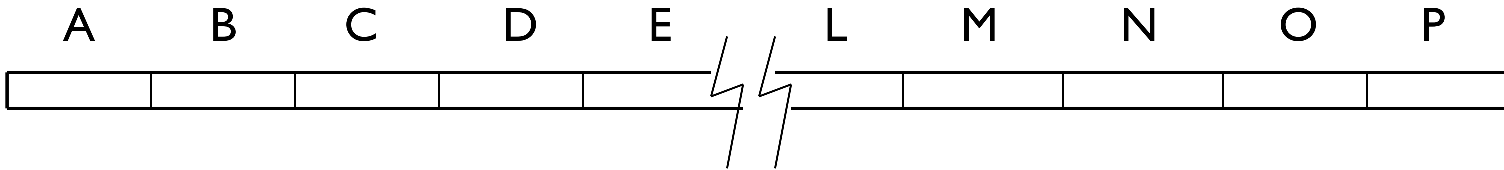
VPN 1 → PPN 5

STRATEGY: CACHE PAGE TRANSLATIONS



TLB ORGANIZATION

TLB Entry



Fully associative

Any given translation can be anywhere in the TLB
Hardware will search the entire TLB in parallel

ARRAY ITERATOR (W/ TLB)

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

Assume ‘a’ starts at 0x1000
Ignore instruction fetches
and access to ‘i’

Assume following virtual address stream:
load 0x1000

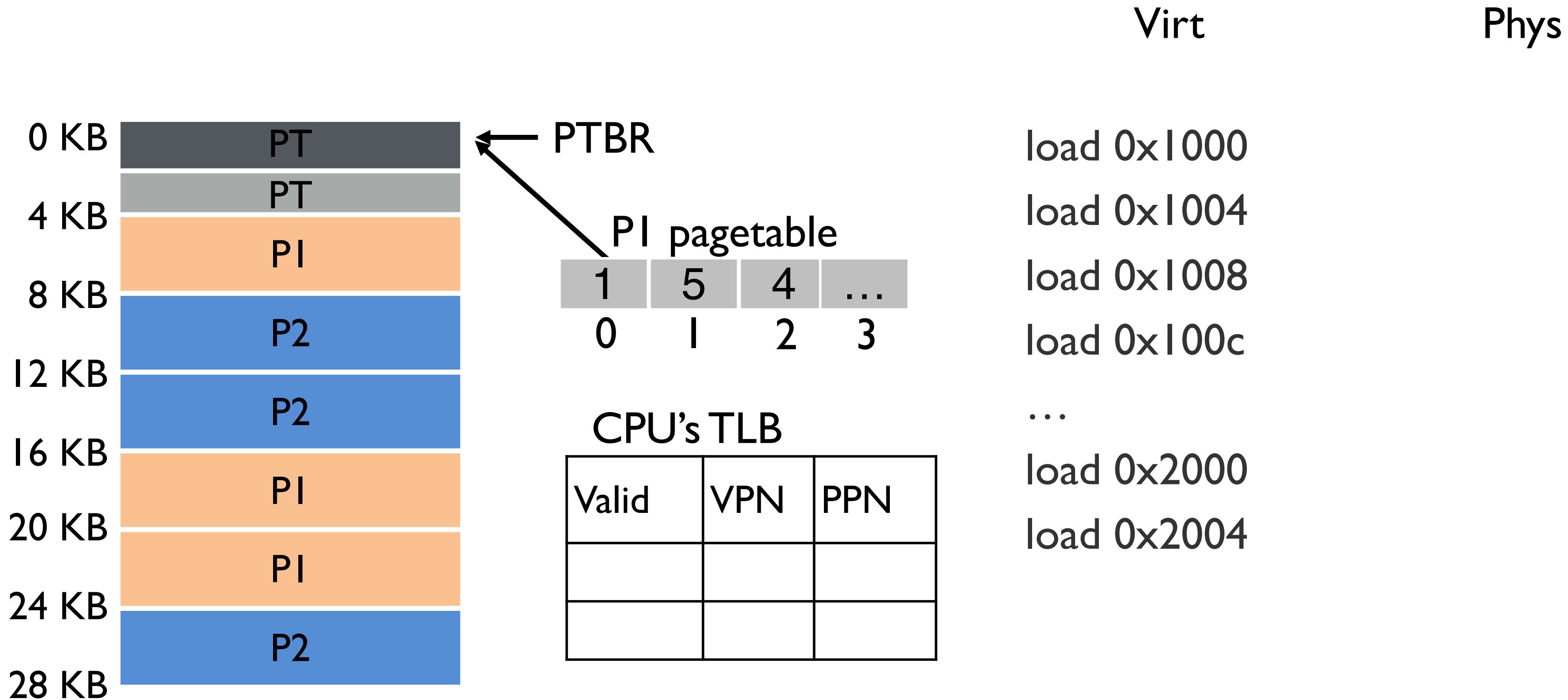
load 0x1004

load 0x1008

load 0x100C
…

What will TLB behavior look like?

TLB ACCESSES: SEQUENTIAL EXAMPLE



Virt

Phys

load 0x1000

load 0x1004

load 0x1008

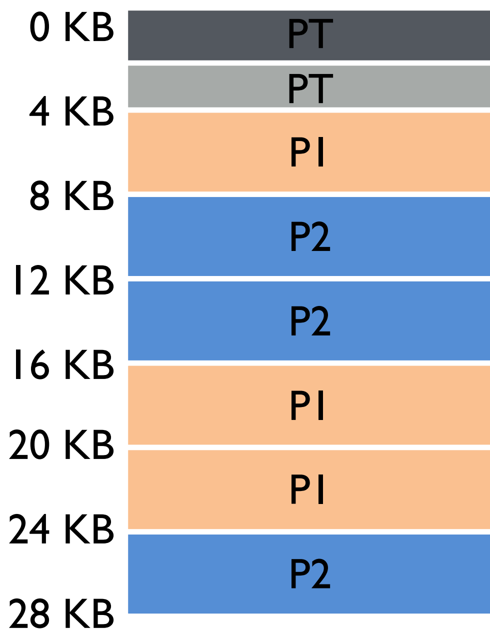
load 0x100c

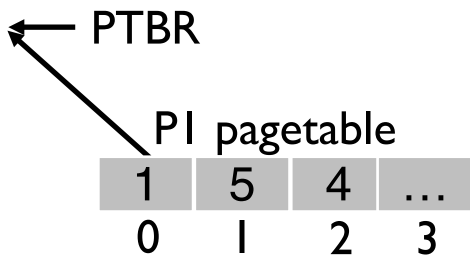
...

load 0x2000

load 0x2004

TLB ACCESSES: SEQUENTIAL EXAMPLE





CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virt

load 0x1000
load 0x1004
load 0x1008
load 0x100c
...
load 0x2000
load 0x2004

Phys

load 0x0004
load 0x5000
(TLB hit)
load 0x5004
(TLB hit)
load 0x5008
(TLB hit)
load 0x500C
...
load 0x0008
load 0x4000
(TLB hit)
load 0x4004

QUIZ 10: TLBS

https://tinyurl.com/cs537-sp20-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.

PageTable

VPN	
VPN:0	0x0
	0x0
	0x1
	0x9
	0x7
	0x8
	0
	⋮
VPN:15	0

Virtual Addresses

```
0x3000: load 0x5320, %eax
0x3004: load 0x4004, %ebx
0x3008: mul %ecx, %eax, %ebx
0x300C: store %ebx, 0x5324
0x3010: load 0x5328, %ebx
```

Memory accesses

Total

VPN:0

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

PageTable

Virtual Addresses

0x3000: load 0x5320, %eax
0x3004: load 0x4004, %ebx
0x3008: mul %ecx, %eax, %ebx
0x300C: store %ebx, 0x5324
0x3010: load 0x5328, %ebx

TLB

Valid	VPN	PPN
0	2	6
0	7	23
0	2	5
0	3	2
0	1	89

Memory accesses

TLB: POLICIES

How to we replace entries in the TLB?

How do we handle context switches?

PERFORMANCE OF TLB?

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

Would hit rate get better or worse with smaller pages?

Miss rate of TLB: # TLB misses / # TLB lookups

TLB lookups? number of accesses to a =

TLB misses?
= number of unique pages accessed

Miss rate?

Hit rate?

WORKLOAD ACCESS PATTERNS

Workload A

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

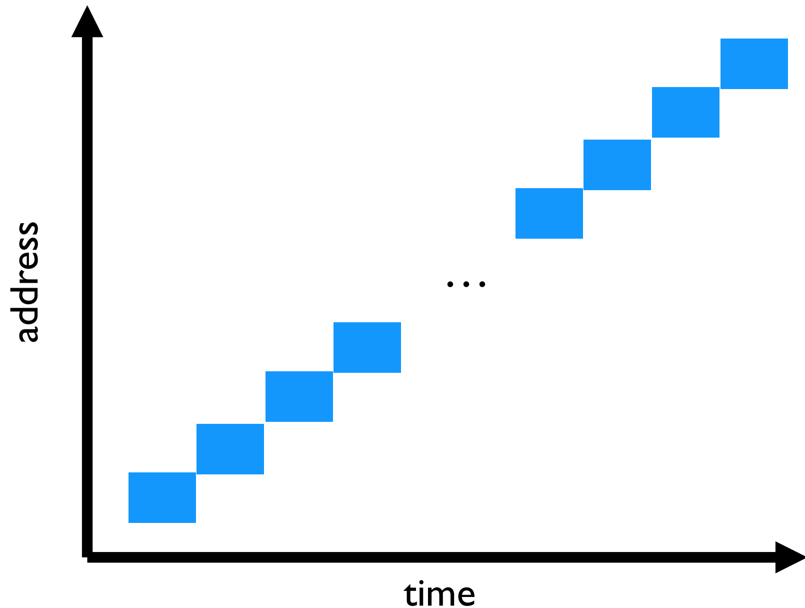
Workload B

```
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

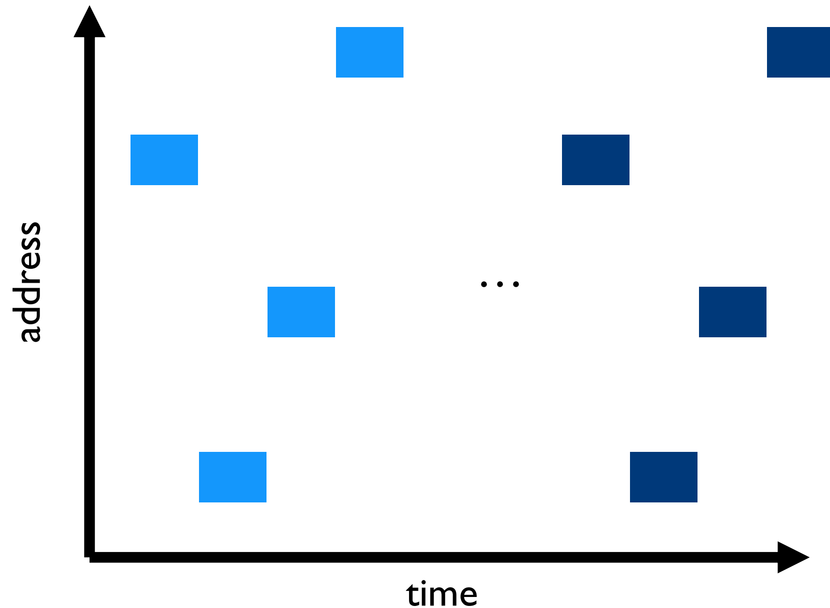
Spatial Locality

Sequential Accesses



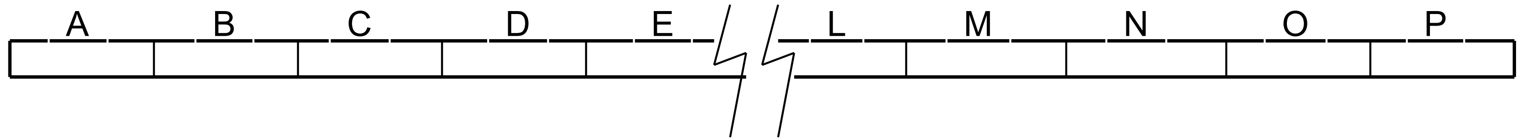
Temporal Locality

Repeated Random Accesses

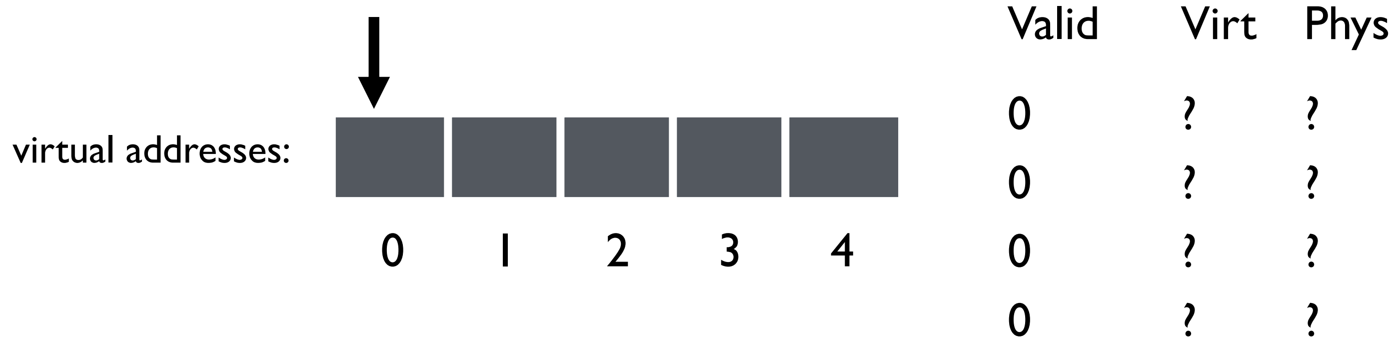


TLB REPLACEMENT POLICIES

LRU: evict Least-Recently Used TLB slot when needed



LRU TROUBLES



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

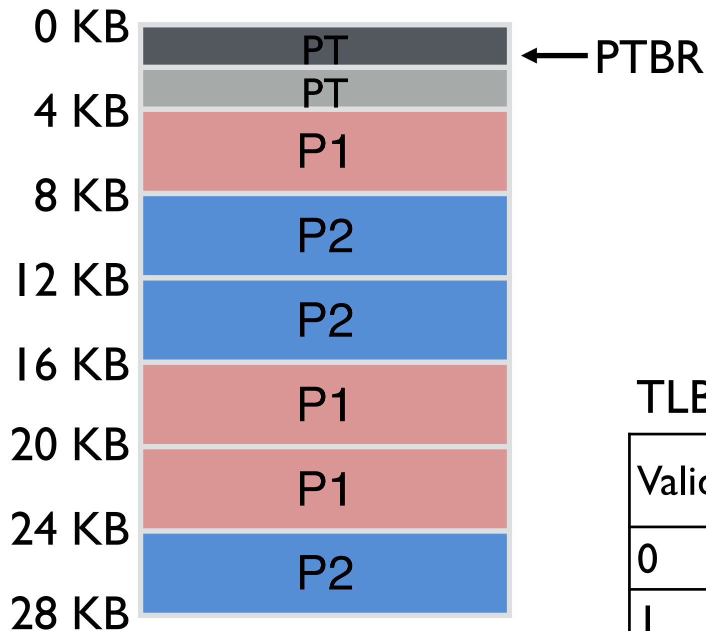
What happens if a process uses cached TLB entries from another process?

1. Flush TLB on each switch

   Costly → lose all recently cached translations

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

TLB EXAMPLE WITH ASID



Virtual	Physical
load 0x1444 ASID: 12	
load 0x1444 ASID: 11	

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

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”

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

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

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

- 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-sp20-quiz11

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

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

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

DISADVANTAGES OF PAGING

Additional memory reference to page table → Very inefficient

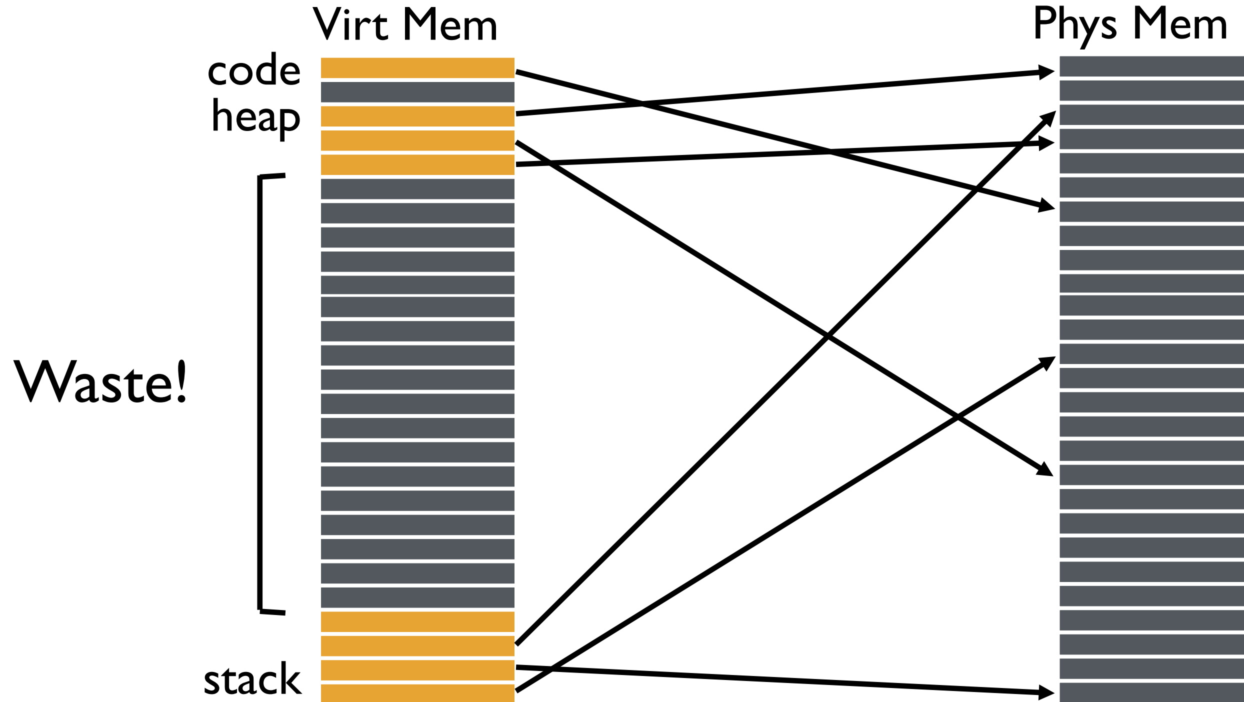
- Page table must be stored in memory
- MMU stores only base address of page table

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?



MANY INVALID PT ENTRIES

PFN	valid	prot
10	1	r-x
-	0	-
23	1	rw-
-	0	-
-	0	-
-	0	-
-	0	-
…many more invalid…		
-	0	-
-	0	-
-	0	-
-	0	-
28	1	rw-
4	1	rw-

how to avoid storing these?

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. Segmented Pagetables (Today)
2. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
3. Inverted Pagetables

VALID PTES ARE CONTIGUOUS

PFN	valid	prot
10	1	r-x
-	0	-
23	1	rw-
-	0	-
-	0	-
-	0	-
-	0	-
…many more invalid…		
-	0	-
-	0	-
-	0	-
-	0	-
28	1	rw-
4	1	rw-

how to avoid storing these?

Note "hole" in addr space: valids vs. invalids are clustered

How did OS avoid allocating holes in phys memory?

Segmentation

COMBINE PAGING AND SEGMENTATION

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages

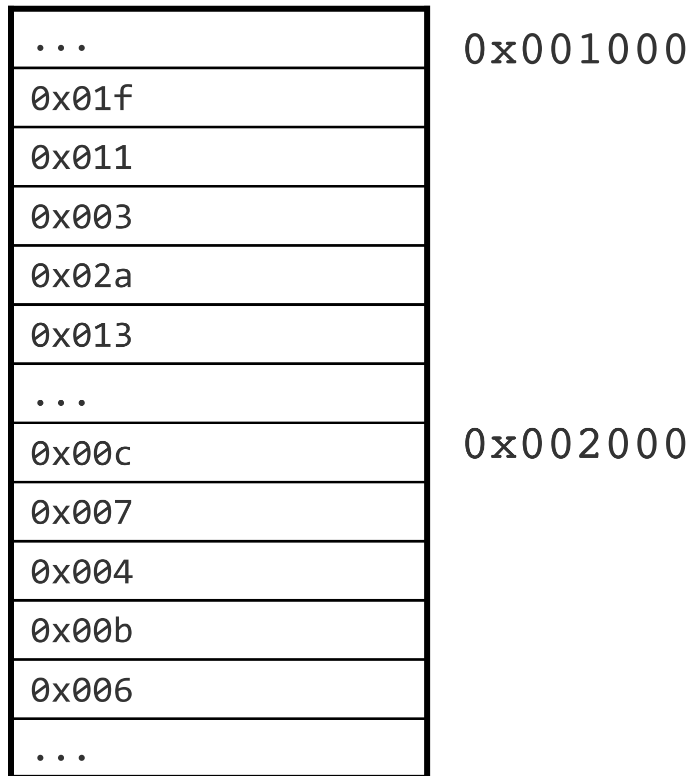
Logical address divided into three portions

Implementation

- Each segment has a page table
- Each segment track base (physical address) and bounds of the **page table**

EXAMPLE: PAGING AND SEGMENTATION

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1



0x002070 read:

0x202016 read:

0x104c84 read:

0x010424 write:

0x210014 write:

0x203568 read:

ADVANTAGES OF PAGING AND SEGMENTATION

Advantages of Segments

- Supports sparse address spaces.
- Decreases size of page tables. If segment not used, not need for page table

Advantages of Pages

- No external fragmentation
- Segments can grow without any reshuffling
- Can run process when some pages are swapped to disk (next lecture)

Advantages of Both

- Increases flexibility of sharing
  - Share either single page or entire segment
  - How?

DISADVANTAGES OF PAGING AND SEGMENTATION

Potentially large page tables (for each segment)

- Must allocate each page table contiguously
- More problematic with more address bits
- Page table size?
  - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:

= Number of entries * size of each entry

= Number of pages * 4 bytes

= 2^18 * 4 bytes = 2^20 bytes = 1 MB!!!

NEXT STEPS

Project 2a: Due Friday

Next class: Better pagetables, swapping!