

# MEMORY: TLBS, SMALLER PAGETABLES

Shivaram Venkataraman

CS 537, Spring 2019

# 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**

# REVIEW: MATCH DESCRIPTION

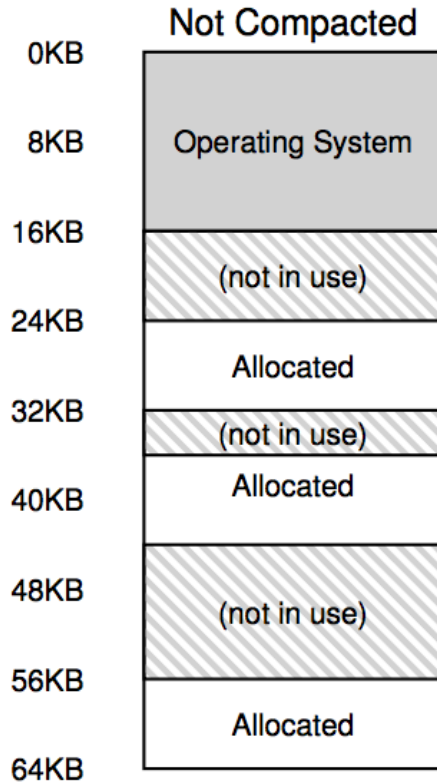
Description

1. one process uses RAM at a time
2. rewrite code and addresses before running
3. add per-process starting location to virt addr to obtain phys addr
4. dynamic approach that verifies address is in valid range
5. several base+bound pairs per process

Name of approach

Candidates: Segmentation, Static Relocation, Base, Base+Bounds, Time Sharing

# FRAGMENTATION

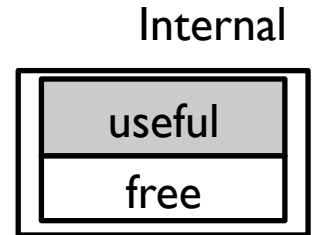


Definition: Free memory that can't be usefully allocated

Types of fragmentation

External: Visible to allocator (e.g., OS)

Internal: Visible to requester



# PAGING

Goal: Eliminate requirement that address space is contiguous

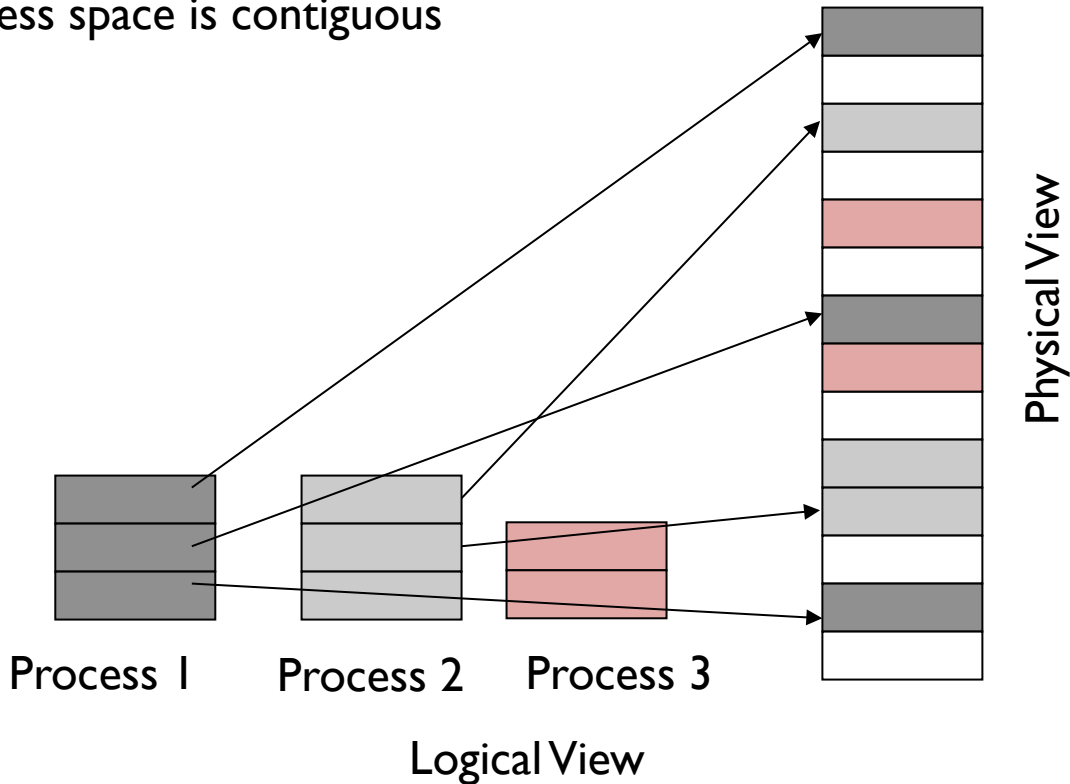
Eliminate external fragmentation

Grow segments as needed

Idea:

Divide address spaces and physical memory into fixed-sized pages

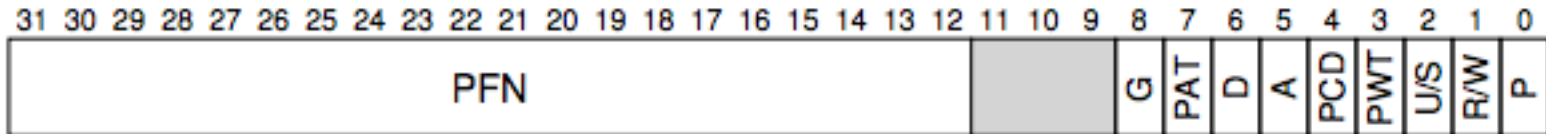
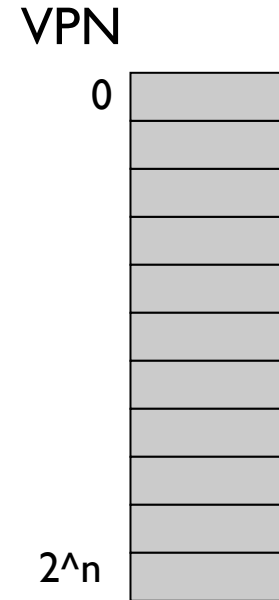
Size:  $2^n$ , Example: 4KB



# PAGETABLES

What is a good data structure ?

Simple solution: Linear page table aka *array*





# 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 into register

# MEMORY ACCESSSES WITH PAGING

0x0040: movl 0x1400, %edi

Assume PT is at phys addr 0x3000

Assume PTE's are 4 bytes

Assume 4KB pages

How many bits for offset? 12

Simplified view  
of page table

2
0
3
1

Fetch instruction at logical addr 0x0040

- Access page table to get ppn for vpn \_\_\_\_
- **Mem ref 1:**
- Learn vpn \_\_\_\_ is at ppn \_\_\_\_
- Fetch instruction at \_\_\_\_\_ (**Mem ref 2**)

Exec, load from logical addr 0x1400

- Access page table to get ppn for vpn \_\_\_\_
- **Mem ref 3:**
- Learn vpn \_\_\_\_ is at ppn \_\_\_\_
- Movl from \_\_\_\_\_ into reg (**Mem ref 4**)

# QUIZ: HOW BIG IS A PAGETABLE?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

# 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 ?

# EXAMPLE: ARRAY ITERATOR

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

Assume 'a' starts at 0x3000  
Ignore instruction fetches  
and access to 'i'

What virtual addresses?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

What physical addresses?

load 0x100C

load 0x7000

load 0x100C

load 0x7004

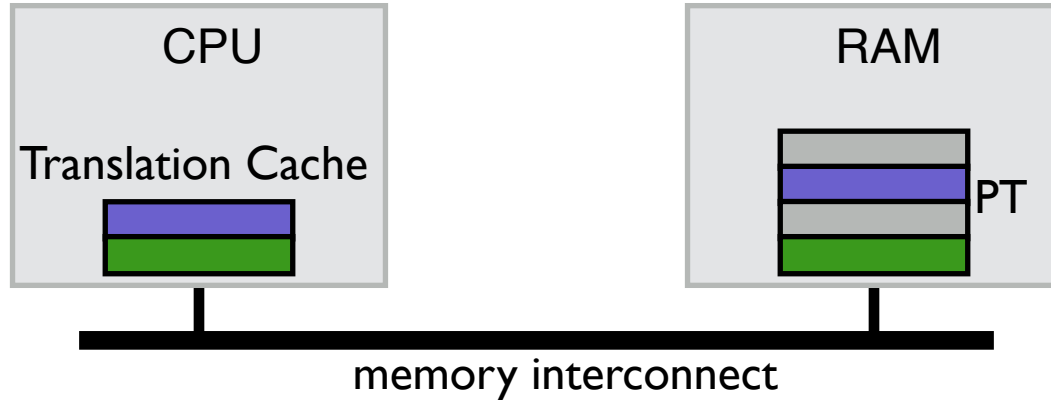
load 0x100C

load 0x7008

load 0x100C

load 0x700C

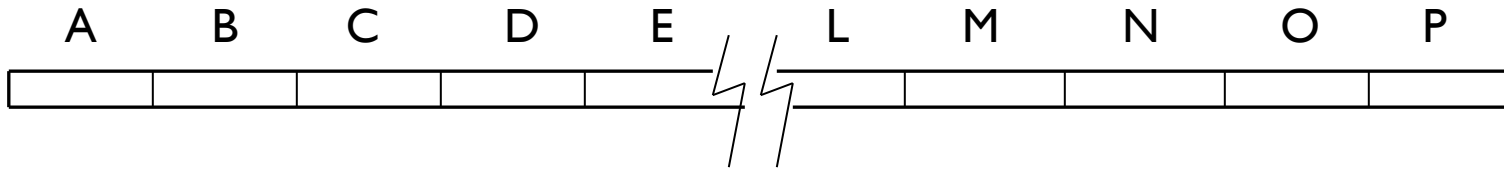
# STRATEGY: CACHE PAGE TRANSLATIONS



**TLB: TRANSLATION LOOKASIDE BUFFER**

# 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

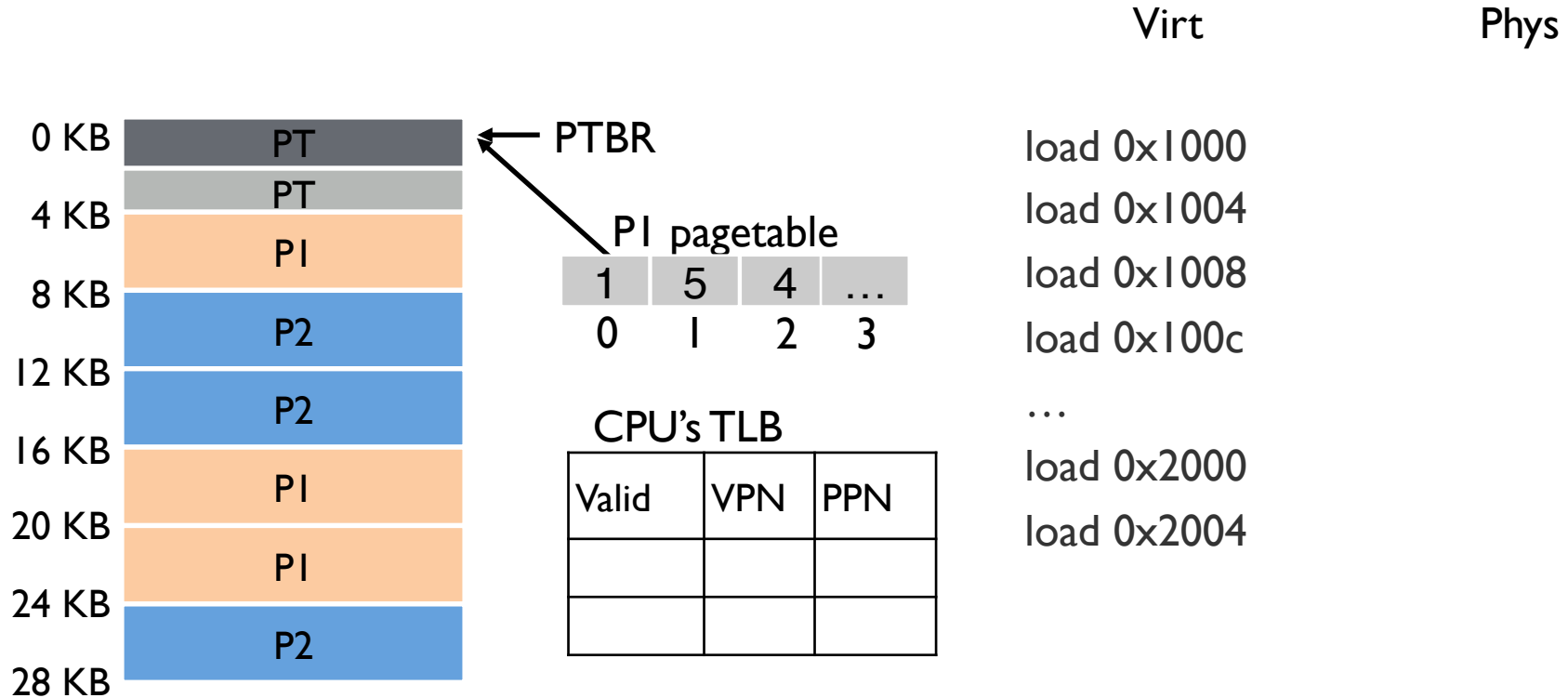
load 0x1008

load 0x100C

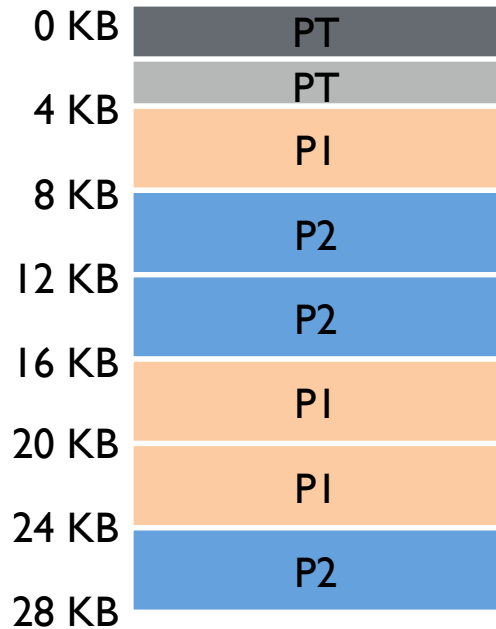
...

What will TLB behavior look like?

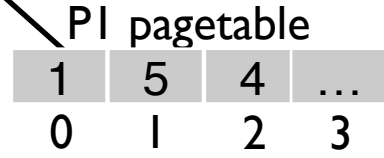
# TLB ACCESSES: SEQUENTIAL EXAMPLE



# TLB ACCESSES: SEQUENTIAL EXAMPLE



PTBR



CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virt	Phys
load 0x1000	load 0x0004
load 0x1004	load 0x5000
load 0x1008	(TLB hit)
load 0x100c	load 0x5004
...	(TLB hit)
load 0x2000	load 0x5008
load 0x2004	(TLB hit)
	load 0x500c
	...
	load 0x0008
	load 0x4000
	(TLB hit)
	load 0x4004

# PERFORMANCE OF TLB?

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

$\# \text{TLB lookups?}$  number of accesses to a =

$\# \text{TLB misses?}$

= number of unique pages accessed

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

Miss rate?

Would hit rate get better or worse  
with smaller pages?

Hit rate?

# TLB PERFORMANCE WITH WORKLOADS

Sequential array accesses almost always hit in TLB

- Very fast!

What access pattern will be slow?

- Highly random, with no repeat accesses

# WORKLOAD ACCESS PATTERNS

## Workload A

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

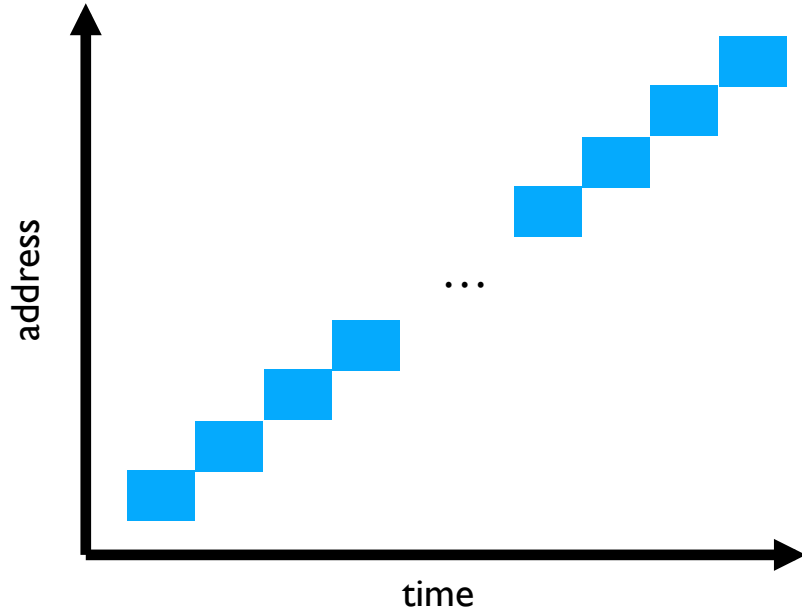
## Workload B

```
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

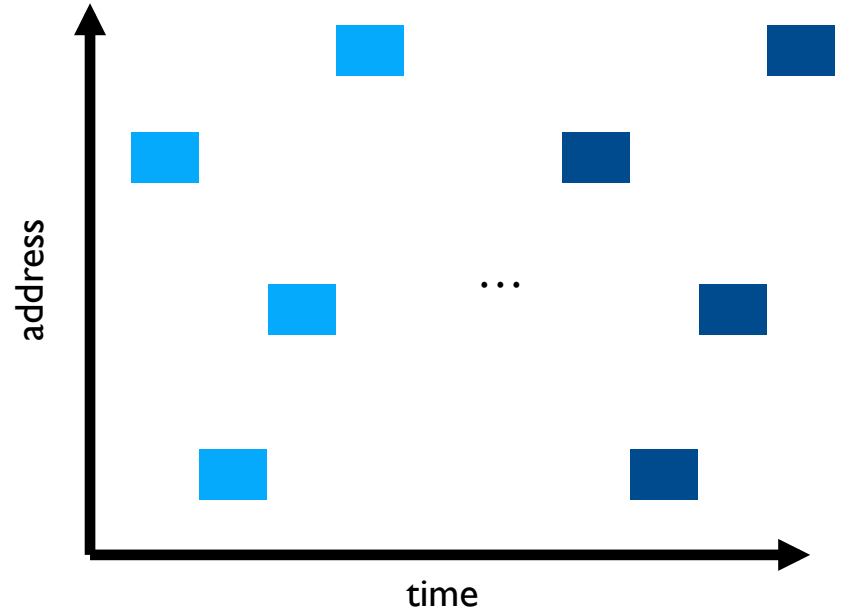
Spatial Locality

Sequential Accesses



Temporal Locality

Repeated Random Accesses



# WORKLOAD LOCALITY

**Spatial Locality:** future access will be to nearby addresses

**Temporal Locality:** future access will be repeats to the same data

What TLB characteristics are best for each type?

Spatial:

- Access same page repeatedly; need same vpn  $\rightarrow$  ppn translation
- Same TLB entry re-used

Temporal:

- Access same address near in future
- Same TLB entry re-used in near future
- How near in future? How many TLB entries are there?



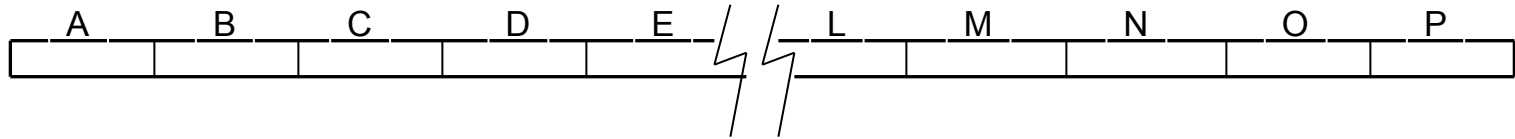
# TLB REPLACEMENT POLICIES

**LRU:** evict Least-Recently Used TLB slot when needed

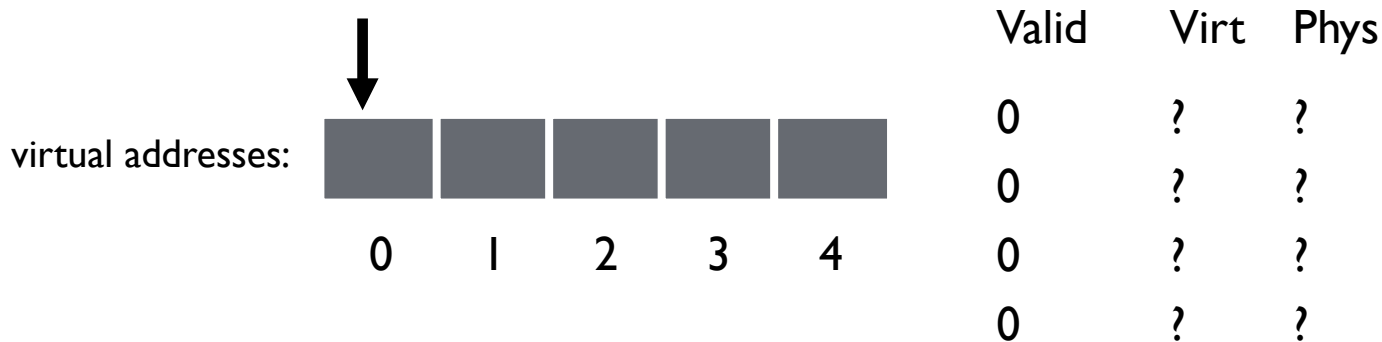
(More on LRU later in policies next week)

**Random:** Evict randomly chosen entry

Which is better?



# 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

(More on LRU later in policies next week)

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

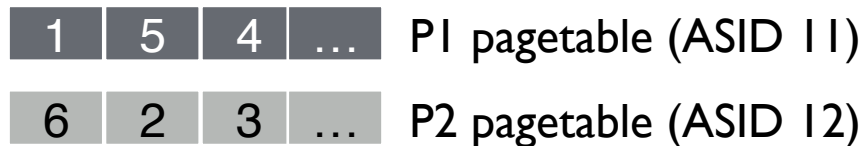
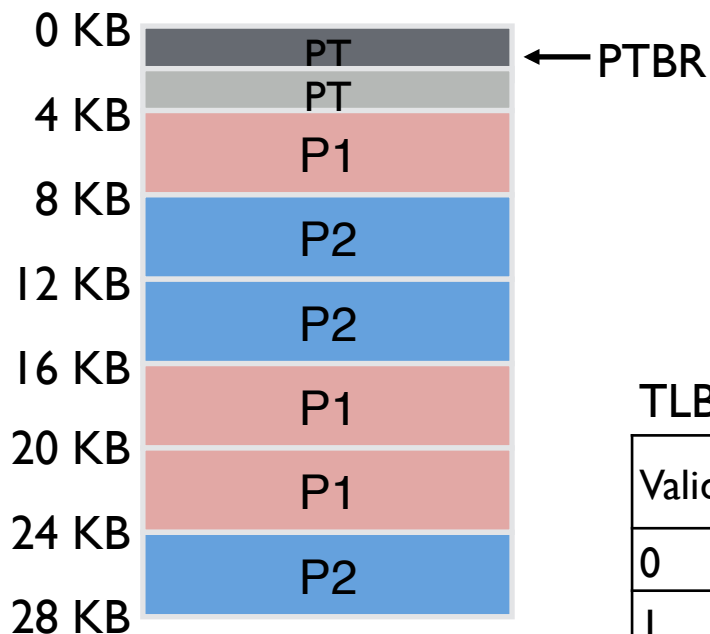
2. Track which entries are for which process

– Address Space Identifier

– Tag each TLB entry with an 8-bit ASID

How many ASIDs do we get? Why not use PIDs?

# 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

- Discard process A’s TLB entries for process B’s entries

Architectures can have multiple TLBs

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

# HW AND OS ROLES

Who Handles TLB MISS? **H/W** or **OS**?

**H/W**

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

# HW AND OS ROLES

Who Handles TLB MISS? **H/W** or **OS**?

**OS:**

CPU traps into OS upon TLB miss  
“Software-managed TLB”

OS interprets pagetables as it chooses

Modifying TLB entries is privileged

Need same protection bits in TLB as pagetable - rwx



# TLB SUMMARY

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

Cache recent page translations → TLB

- Hardware 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

# 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 ?

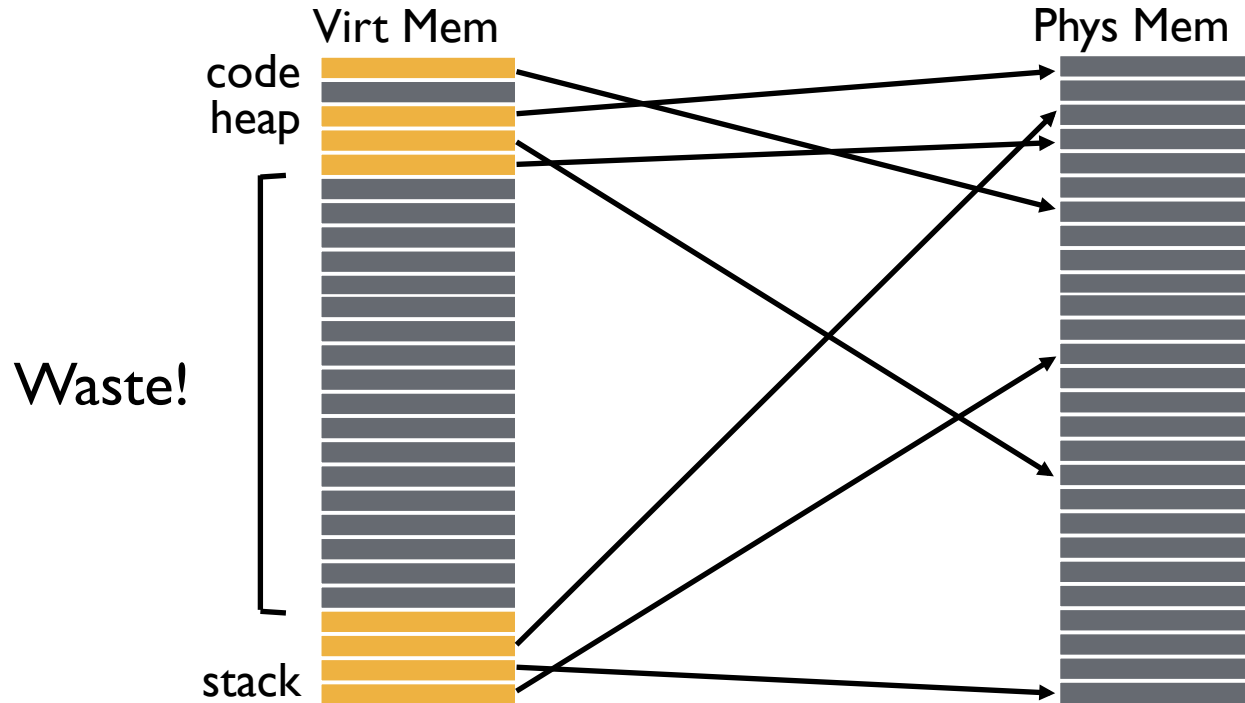
# SMALLER PAGE TABLES

# QUIZ: HOW BIG ARE PAGE TABLES?

1. PTE's are **2 bytes**, and **32** possible virtual page numbers
2. PTE's are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**
3. PTE's are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**
4. PTE's are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

How big is each page table?

# WHY ARE PAGE TABLES SO LARGE?



# MANY INVALID PT ENTRIES

	PFN	valid	prot
	10	1	r-x
	-	0	-
	23	1	rw-
how to avoid storing these?	-	0	-
	-	0	-
	-	0	-
	-	0	-
	-	0	-
	...many more invalid...	0	-
	-	0	-
	-	0	-
	-	0	-
	28	1	rw-
4	1	rw-	

# 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
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		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?

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

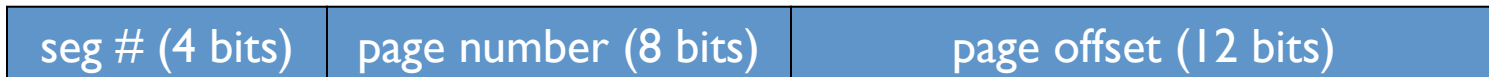
Logical address divided into three portions



Implementation

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

# QUIZ: 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:

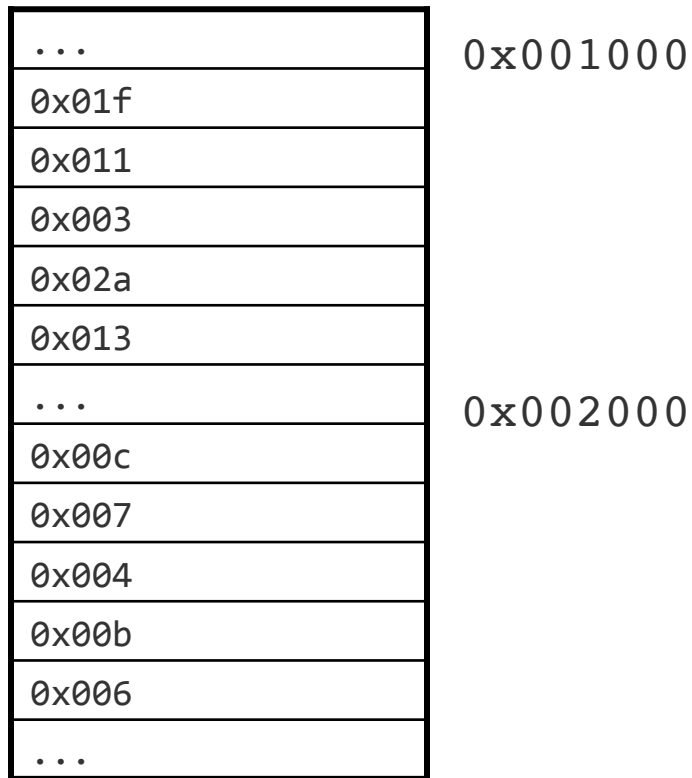
0x202016 read:

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

=  $2^{18} * 4 \text{ bytes} = 2^{20} \text{ bytes} = 1 \text{ MB!!!}$

# OTHER APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
  - Page the page tables
  - Page the pagetables of page tables...
3. Inverted Pagetables

# MULTILEVEL PAGE TABLES

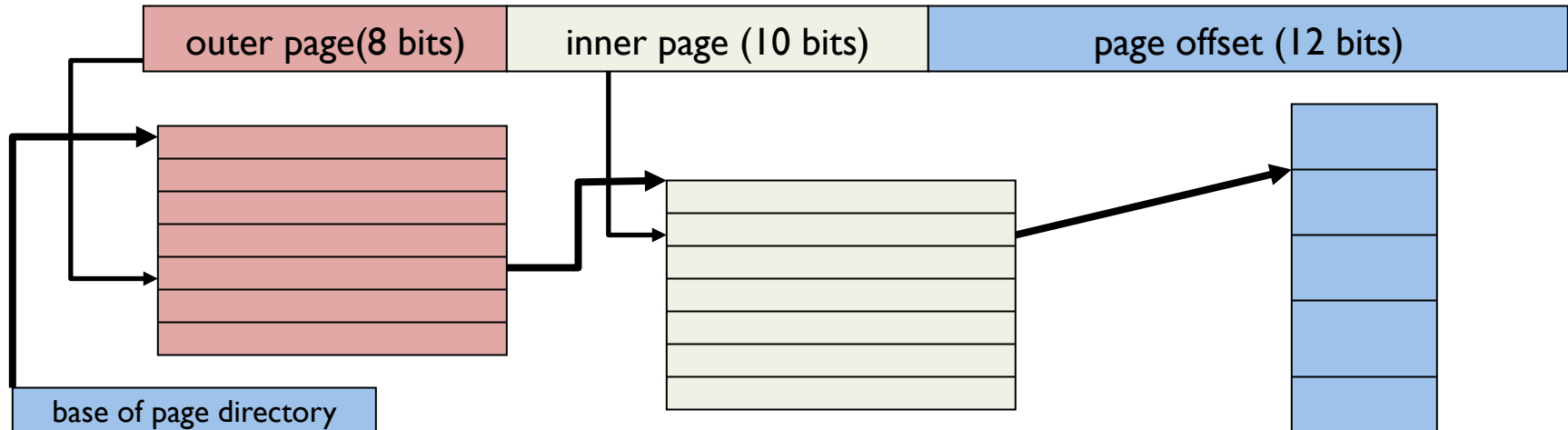
Goal: Allow each page tables to be allocated non-contiguously

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)

# MULTILEVEL PAGE TABLES

30-bit address:

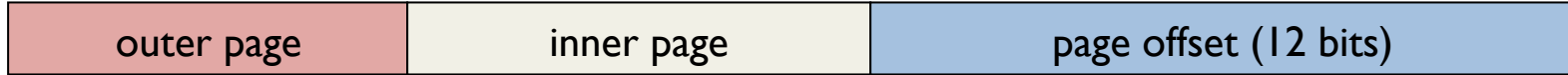






# QUIZ: ADDRESS FORMAT FOR MULTILEVEL PAGING

30-bit address:



How should logical address be structured?

- How many bits for each paging level?

Goal?

- Each page table fits within a page
- PTE size \* number PTE = page size
  - Assume PTE size = 4 bytes
  - Page size =  $2^{12}$  bytes = 4KB
  - $2^2$  bytes \* number PTE =  $2^{12}$  bytes
  - $\rightarrow$  number PTE =  $2^{10}$
- $\rightarrow$  # bits for selecting inner page = 10

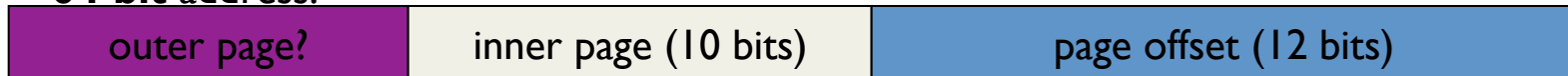
Remaining bits for outer page:

- $30 - 10 - 12 = 8$  bits

# 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 given 1, 2, 3 levels?

4KB / 4 bytes  $\rightarrow$  1K entries per level

1 level: 1K \* 4K =  $2^{22}$  = 4 MB

2 levels: 1K \* 1K \* 4K =  $2^{32}$   $\approx$  4 GB

3 levels: 1K \* 1K \* 1K \* 4K =  $2^{42}$   $\approx$  4 TB

# QUIZ: 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 previous ops changing TLB)

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

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

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

# INVERTED PAGE TABLE

Only need 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

# OTHER APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
  - Page the page tables
  - Page the pagetables of page tables...
3. Inverted Pagetables

# NEXT STEPS

Project 2a: Due Friday

Next class: Better pagetables, swapping!