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?

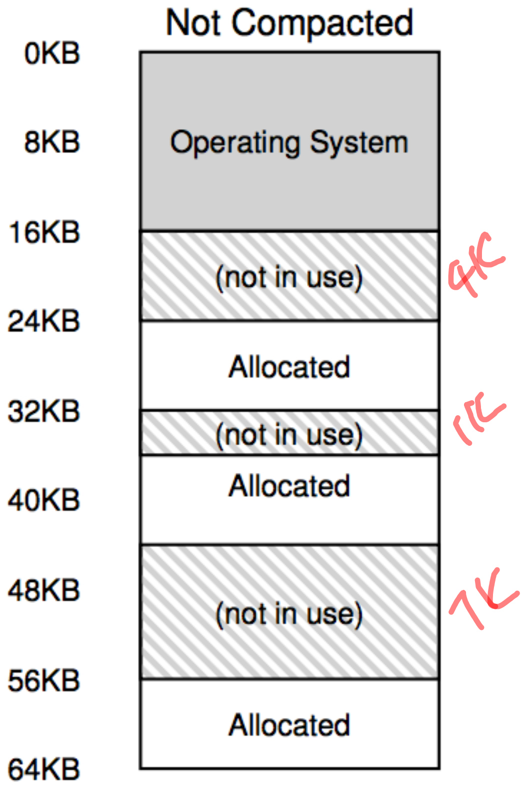
RECAP

REVIEW: MATCH DESCRIPTION

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

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

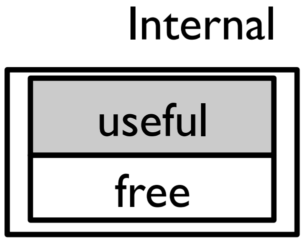
FRAGMENTATION



Definition: Free memory that can't be usefully allocated

Types of fragmentation
- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester

Allocate
12K



PAGING

Goal: Eliminate requirement that address space is contiguous

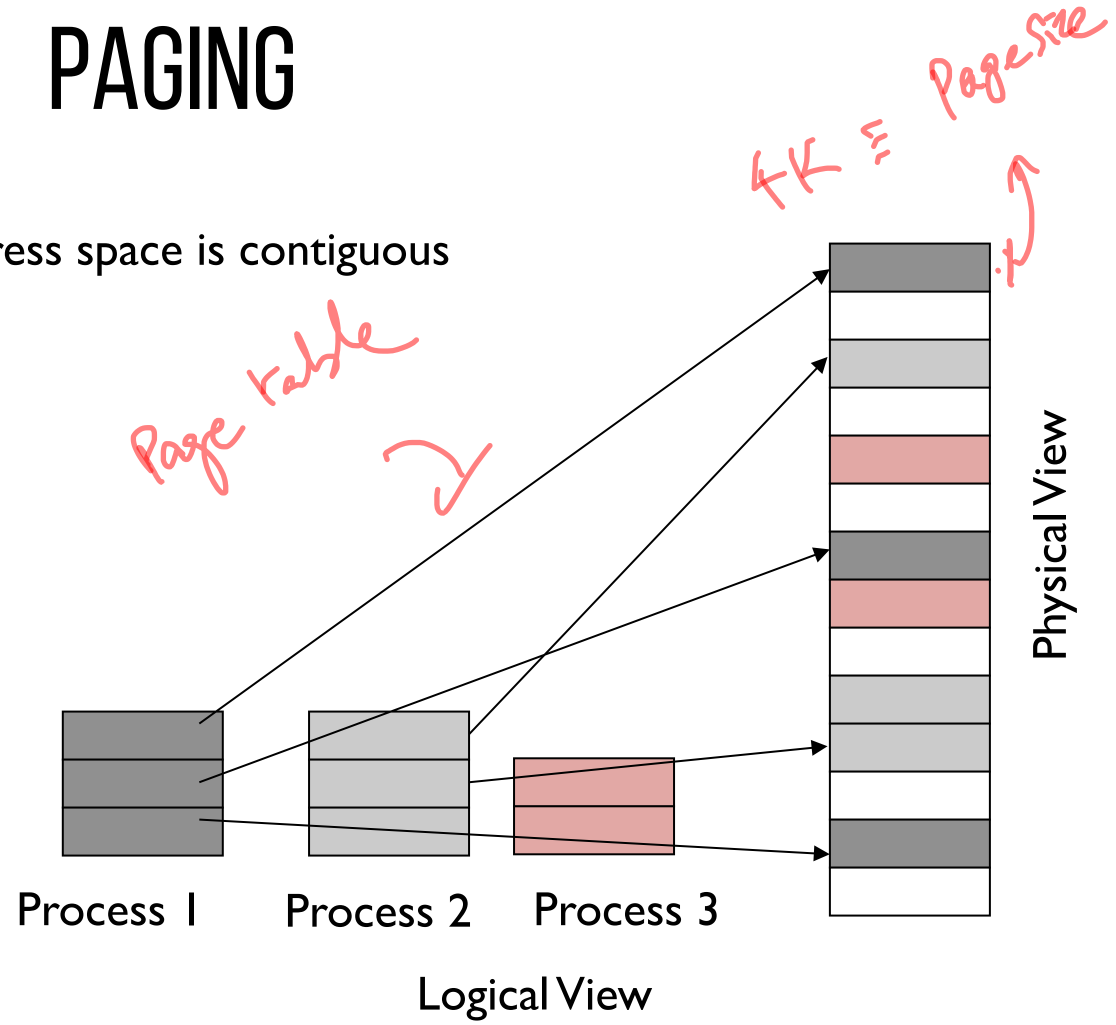
Eliminate external fragmentation

Grow segments as needed

Idea:

Divide address spaces and physical memory into fixed-sized pages

Size: 2^n, Example: 4KB

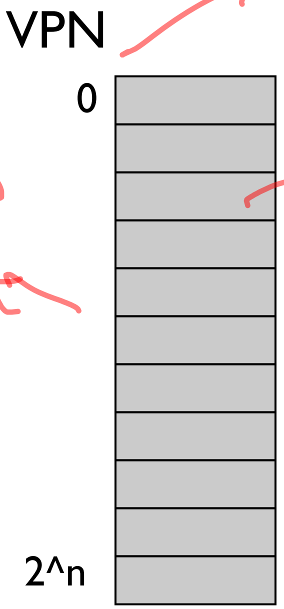


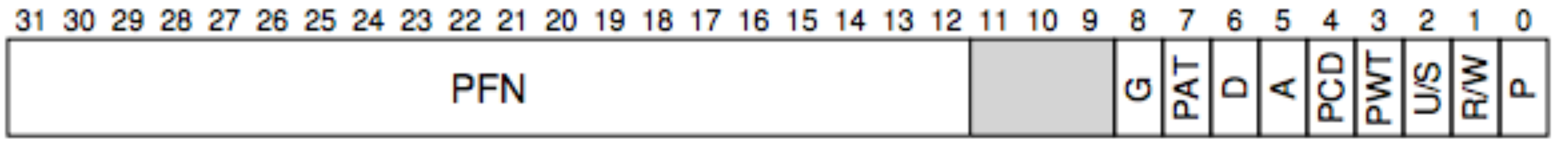
PAGETABLES

What is a good data structure ?

Simple solution: Linear page table aka *array*

Page table per-process





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

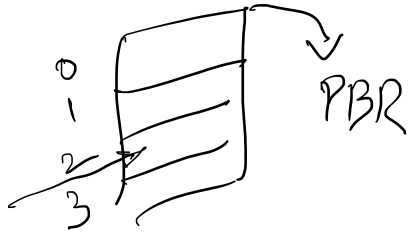
14 bit address space

$\underbrace{\boxed{2}}_{\text{VPN}} \quad \underbrace{\qquad\qquad}_{\text{12 bits offset}}$

|||

4K = Page size

Page Table



$\boxed{10} \rightarrow$ offset of 2

$\underbrace{\quad}_{\text{PFN}} \quad \underbrace{\qquad\qquad}_{\text{Offset}} \rightarrow$ Physical Address

MEMORY ACCESSES WITH PAGING

Addr Inst. read

```
0x0040: movl  0x1400, %edi
```

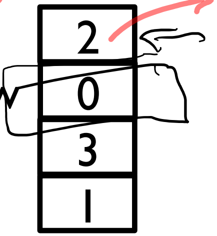
Assume PT is at phys addr 0x3000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

14 bit address space

3000 + Num × size 0x3000

4 byts

Simplified view of page table



0 40 12 bits

0010 2

16 bits PPN

12 bits offset

Fetch instruction at logical addr 0x0040

- Access page table to get ppn for vpn __
- Mem ref 1: 0x3000
- Learn vpn 0 is at ppn 2 ← see here
- Fetch instruction at 0x2040 (Mem ref 2)

vpn offset

Exec, load from logical addr 0x1400

- Access page table to get ppn for vpn 1
- Mem ref 3: 0x3004
- Learn vpn 1 is at ppn 0
- Movl from ______ into reg (Mem ref 4)

0x0400

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

$\longrightarrow$ 2 memory location

4 memory accesses

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 ?

20 bits VPN $\Leftarrow$ 2^{20} elements $\times$ 4 bytes $\Leftarrow$ 4 MB

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 → a[0]

load 0x3004

load 0x3008

load 0x300C

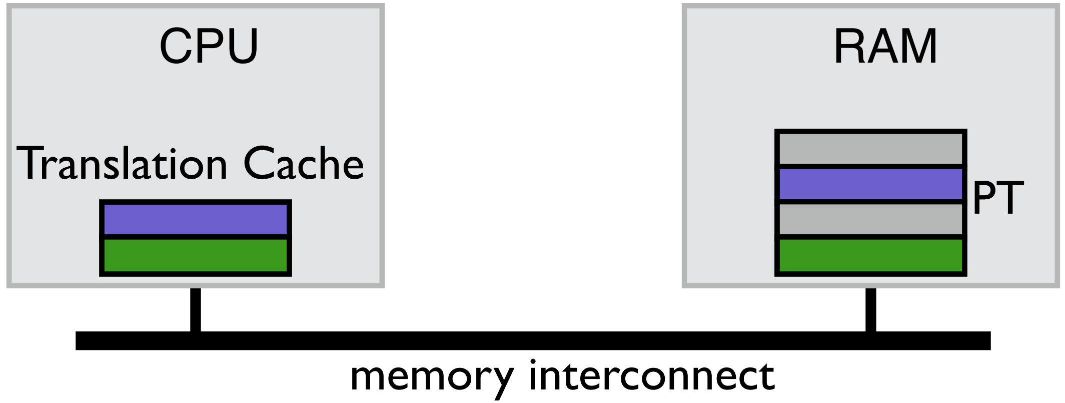
What physical addresses?

load 0x100C → Page table
load 0x7000 → a[0]
load 0x100C → Pagetable
load 0x7004 → a[1]
load 0x100C
load 0x7008
load 0x100C
load 0x700C

STRATEGY: CACHE PAGE TRANSLATIONS

Cache popular entries

MMU



TLB: TRANSLATION LOOKASIDE BUFFER

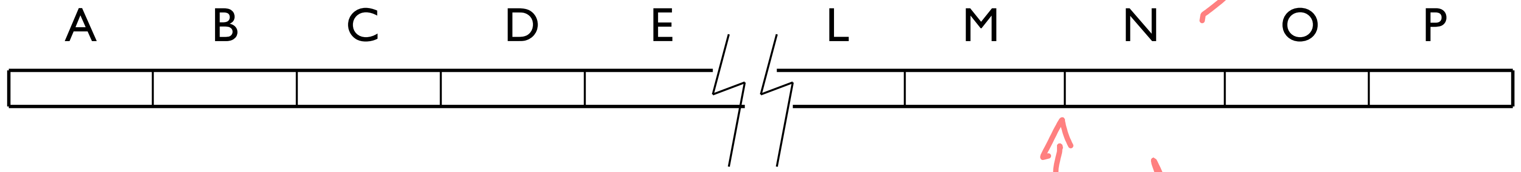
TLB ORGANIZATION

similar to L1 cache in design

TLB Entry

Tag (virtual page number)	Physical page number (page table entry)

Physical page number



virtual page number

Fully associative

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

ARRAY ITERATOR (W/ TLB)

Page Table
0x0000

array size 2048

```
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

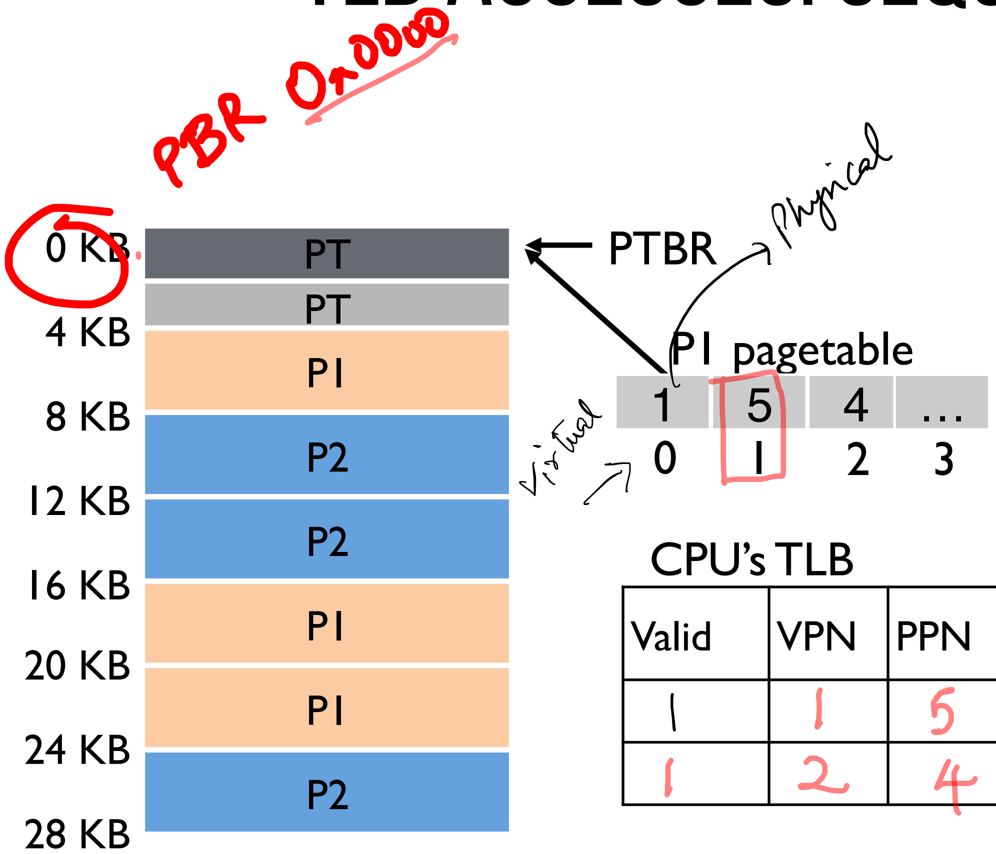
load 0x1004

load 0x1008

load 0x100C
…

What will TLB behavior look like?

TLB ACCESSES: SEQUENTIAL EXAMPLE



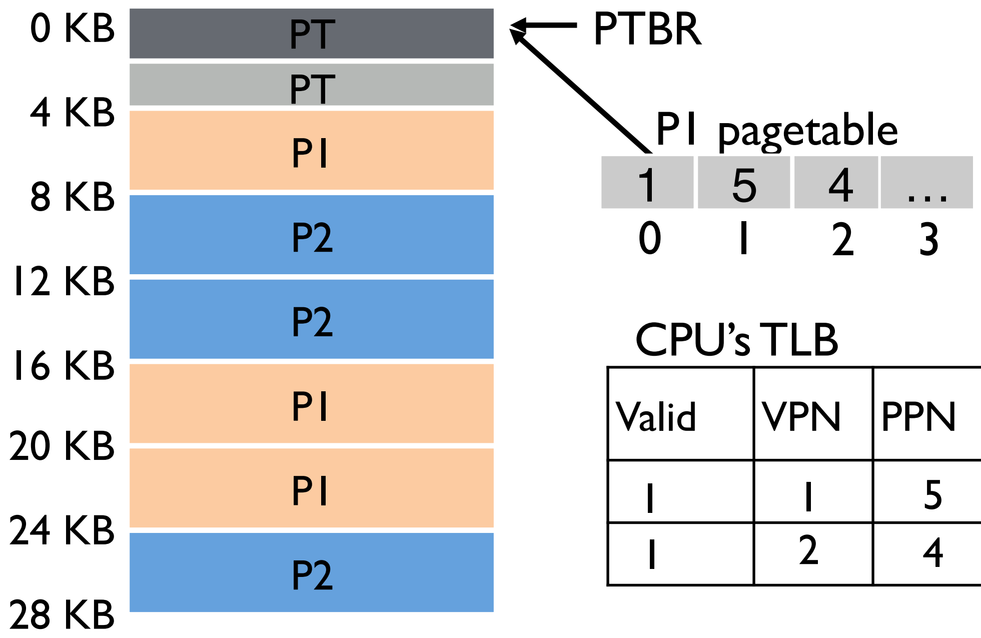
CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Page table ↑

Virt	Phys
load 0x1000	TLB Miss 0x0004 → load 0x5000
load 0x1004	→ TLB Hit 0x5004
load 0x1008	
load 0x100c	
…	
load 0x2000	TLB miss 0x0008 → load 0x4000
load 0x2004	

TLB ACCESSES: SEQUENTIAL EXAMPLE



CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virt

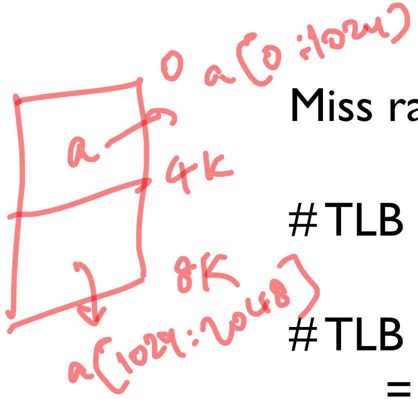
load 0x1000
load 0x1004
load 0x1008
load 0x100c
…
load 0x2000
load 0x2004

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

PERFORMANCE OF TLB?

page size = 4K



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

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

# TLB lookups? number of accesses to a = 2048

# TLB misses?

= number of unique pages accessed

$= \frac{2048 \times 4}{4K} = \frac{8K}{4K} = \underline{\underline{2}}$

Miss rate? $= 2/2048 \simeq 2/2000 = 0.1\%$

Hit rate? $1 - \text{miss rate} = 99.9\%$

Would hit rate get better or worse with smaller pages?

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];
}
```

↓ sequential

Workload B

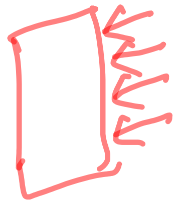
N = size of array

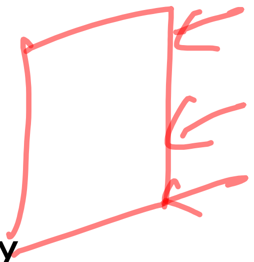
```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
```

↓ random

repeat

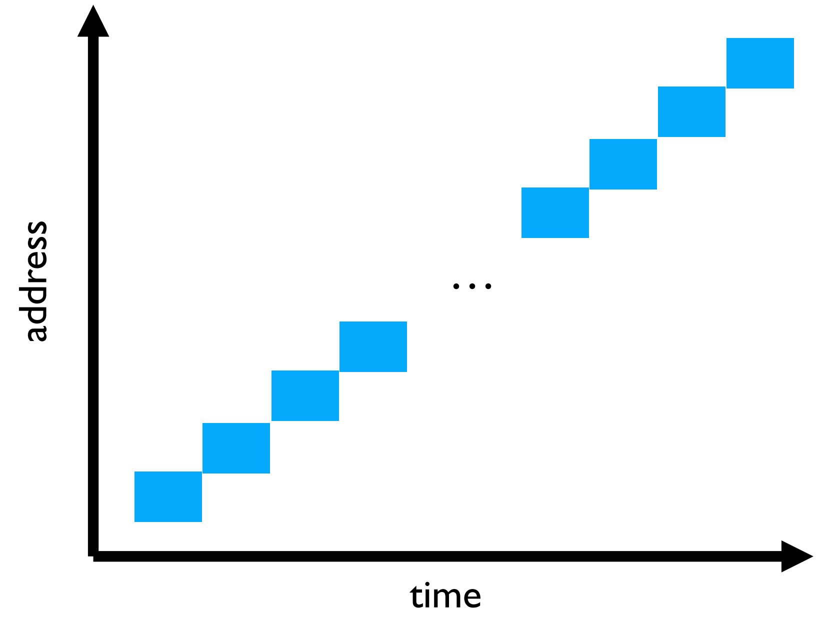
WORKLOAD ACCESS PATTERNS





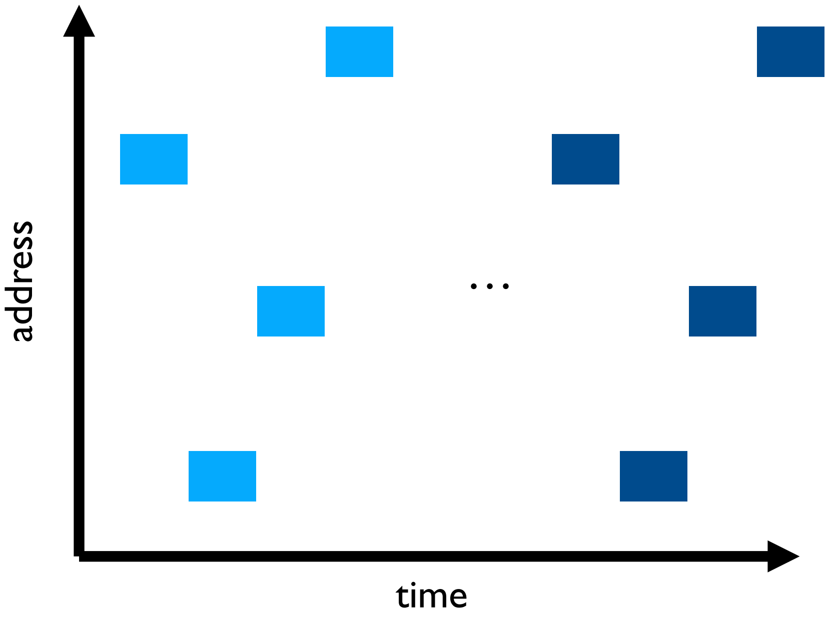
Spatial Locality

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 → 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?

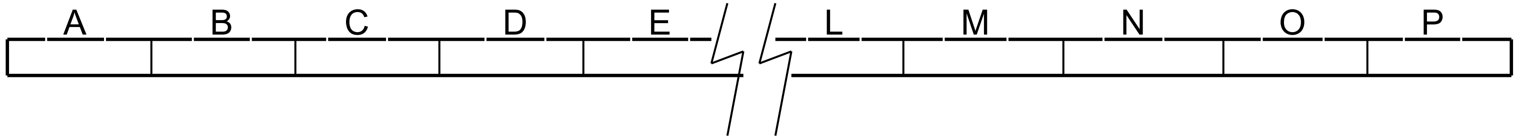
TLB REPLACEMENT POLICIES

$\boxed{\textbf{LRU}}$: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies next week)

Random: Evict randomly choosen entry $\longrightarrow$ Simple. Is this effeitive

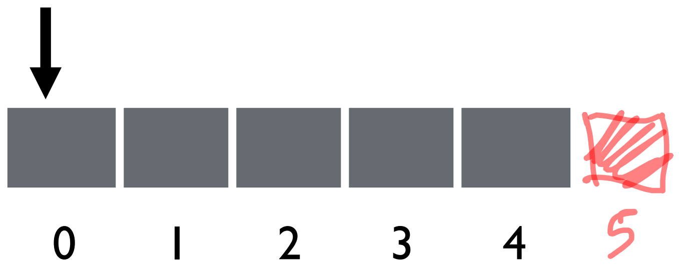
Which is better?



LRU TROUBLES

0x0001
0x1001
0x2001
⋮
0x5001
loop

virtual addresses:



Valid	Virt	Phys
0	~~?~~ ~~0~~ 4	~~?~~ ~~10~~ 33
0	~~?~~ ~~1~~ 5	? ~~20~~ 37
0	~~?~~ 2	~~?~~ 30
0	~~?~~ 3	~~?~~ 23

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?

then you always get a miss

TLB size is smaller than loop

TLB REPLACEMENT POLICIES

LRU: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies next week)

Random: Evict randomly choosen entry

Sometimes random is better than a “smart” policy!

CONTEXT SWITCHES

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

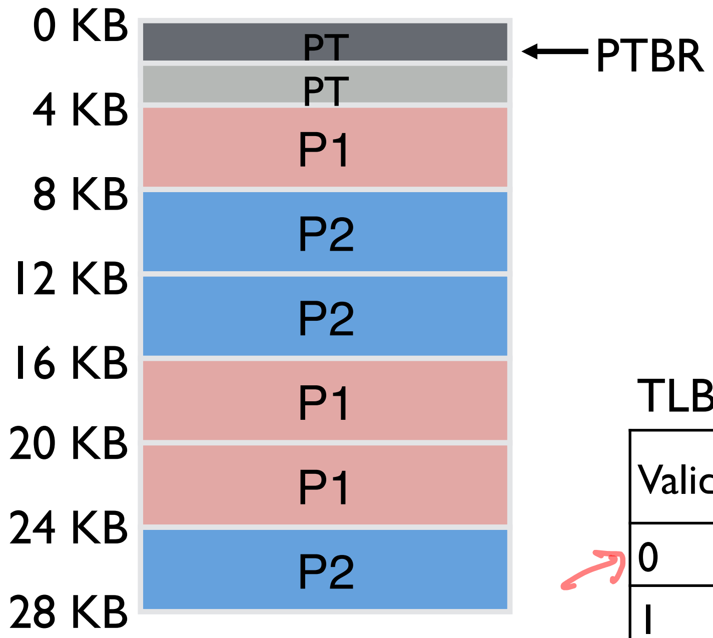
1. Flush TLB on each switch → Set all valid bits to 0 | Page Table → Process specific

   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



1	5	4	…	P1 pagetable (ASID 11)
6	2	3	…	P2 pagetable (ASID 12)

VPN

Virtual	Physical
load 0x1444 ASID: 12	0x2444
load 0x1444 ASID: 11	0x5444

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

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

HW AND OS ROLES

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

Page table base addr

H/W

CPU must know where pagetables are

- $\boxed{\text{CR3}}$ 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 ?

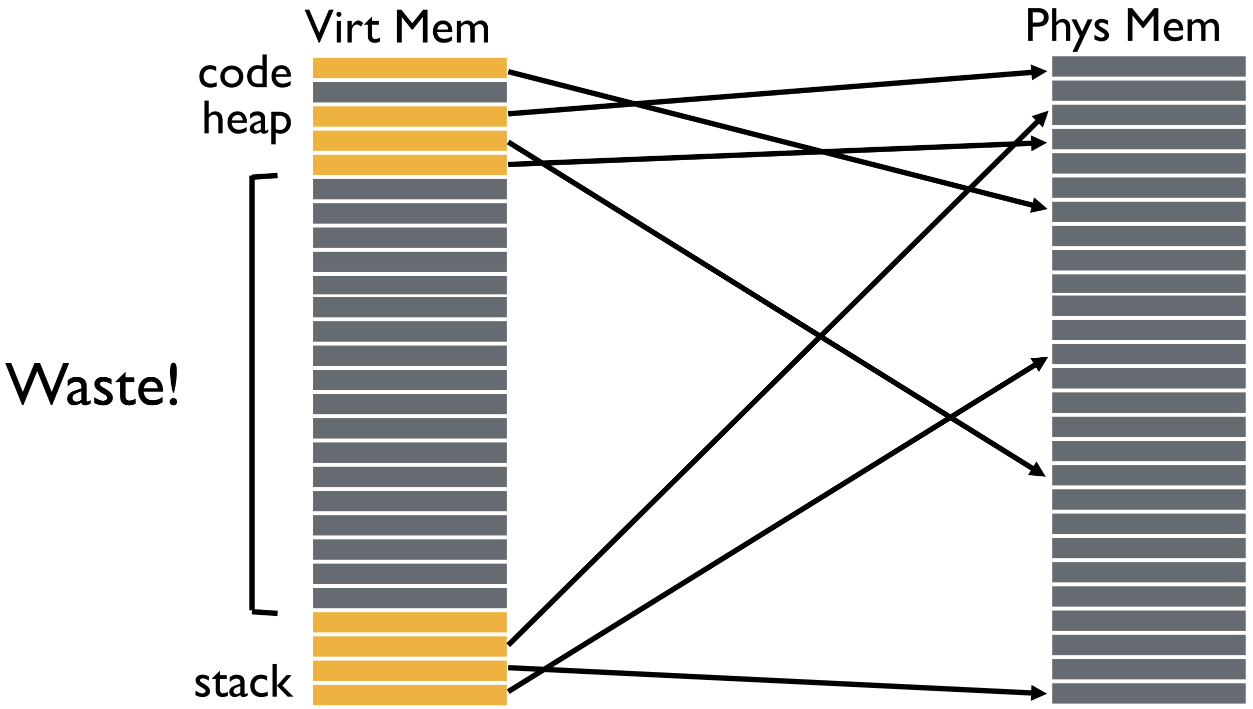
SMALLER PAGE TABLES

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-
-	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
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

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

Implementation

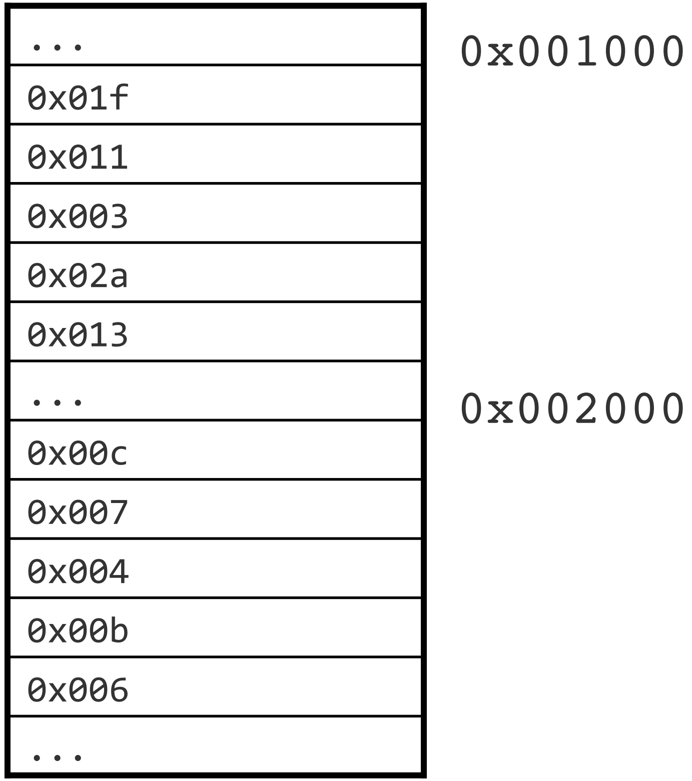
- Each segment has a page table
- Each segment track base (physical address) and bounds of the **page table**

QUIZ: PAGING AND SEGMENTATION

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

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!!!

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

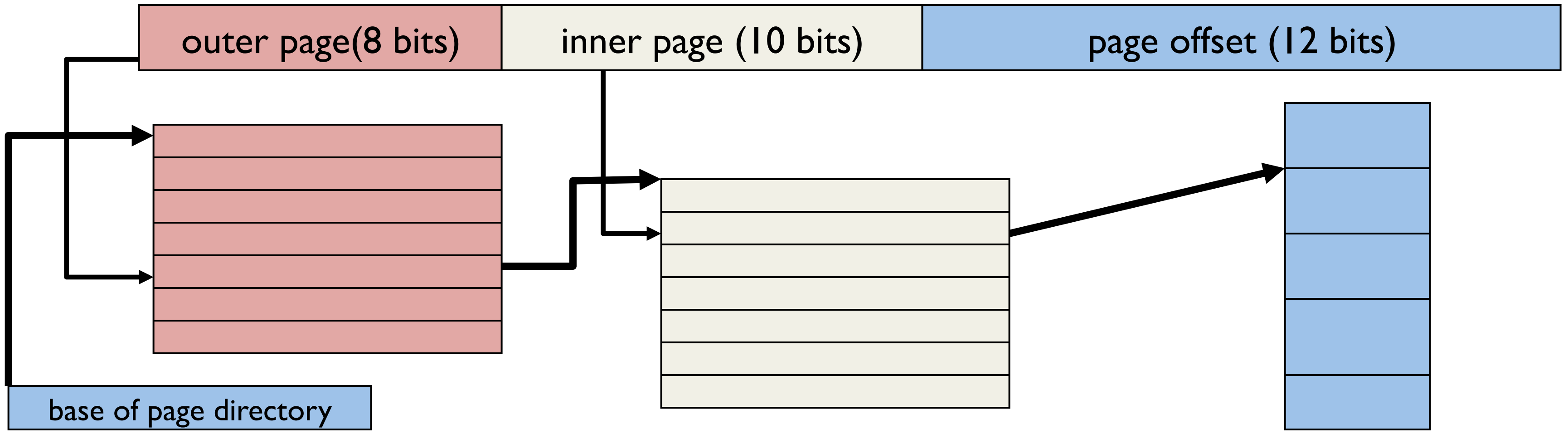
Goal: Allow each page tables to be allocated non-contiguously

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: MULTILEVEL

page directory

PPN	valid
0x3	1
-	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

page of PT (@PPN:0x92)

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

translate 0x01ABC

translate 0x00000

translate 0xFEED0

20-bit address:

outer page(4 bits)	inner page(4 bits)	page offset (12 bits)

QUIZ: ADDRESS FORMAT FOR MULTILEVEL PAGING

30-bit address:

outer page	inner page	page offset (12 bits)

How should logical address be structured?

- How many bits for each paging level?

Goal?

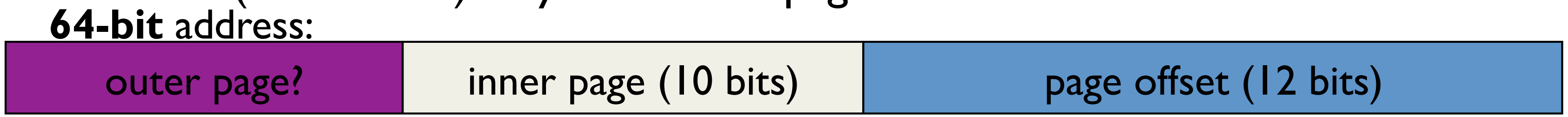
- Each page table fits within a page
- PTE size * number PTE = page size
  - Assume PTE size = 4 bytes
  - Page size = 2^12 bytes = 4KB
  - 2^2 bytes * number PTE = 2^12 bytes
  - → number PTE = 2^10
- → # bits for selecting inner page = 10

Remaining bits for outer page:

- 30 – 10 – 12 = 8 bits

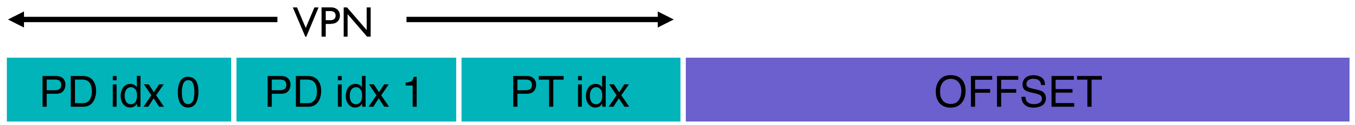
PROBLEM WITH 2 LEVELS?

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



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 → 1K entries per level

1 level: 1K * 4K = **2^22** = 4 MB

2 levels: 1K * 1K * 4K = **2^32** ≈ 4 GB

3 levels: 1K * 1K * 1K * 4K = **2^42** ≈ 4 TB

QUIZ: FULL SYSTEM WITH TLBS

On TLB miss: lookups with more levels more expensive

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

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

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 <ppn, vpn+asid> to find match
- Too much time to search entire table

Better:
- Find possible matches entries by hashing vpn+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!