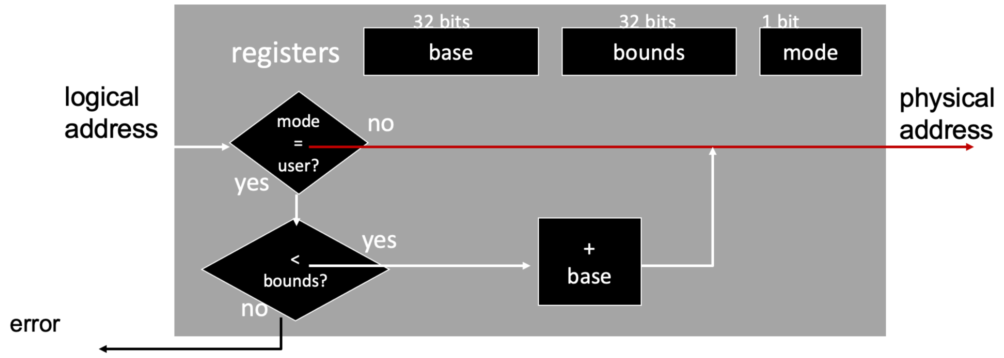
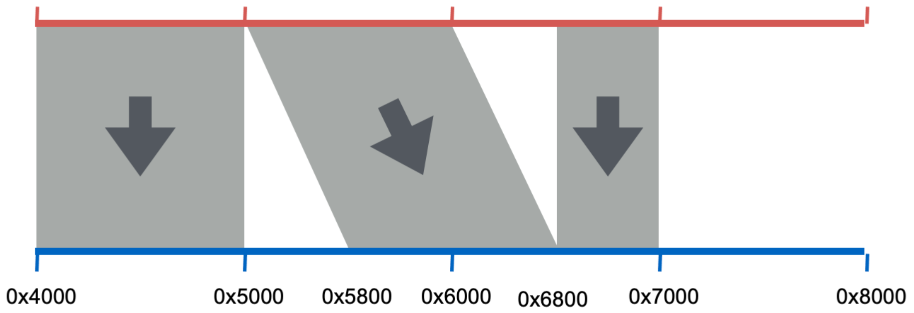
Paging

Review

Base + Bounds



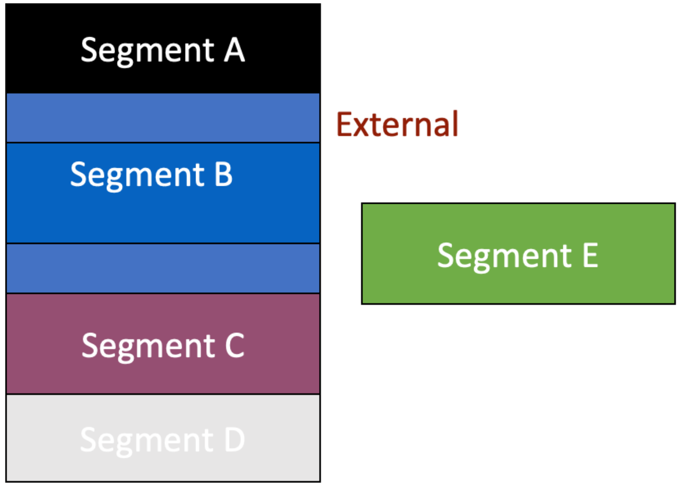
Virtual addresses



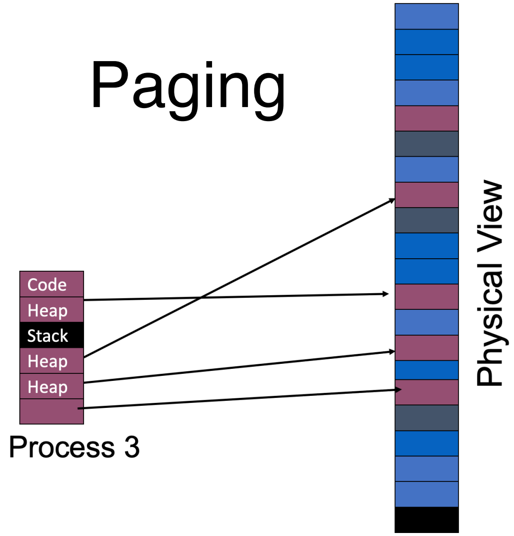
Physical addresses

Segment	Base	Bounds	R	W
0	0x2000	0x6ff	1	0
1	0x0000	0x4ff	1	1
2	0x3000	0xfff	1	1
3	0x0000	0x000	0	0

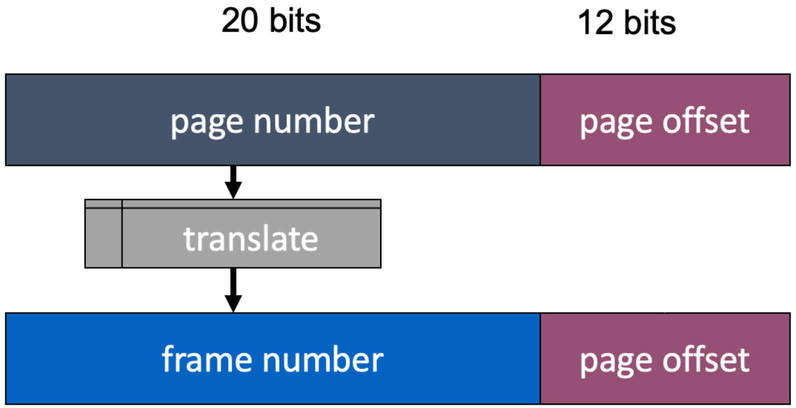
External Fragmentation



Paging

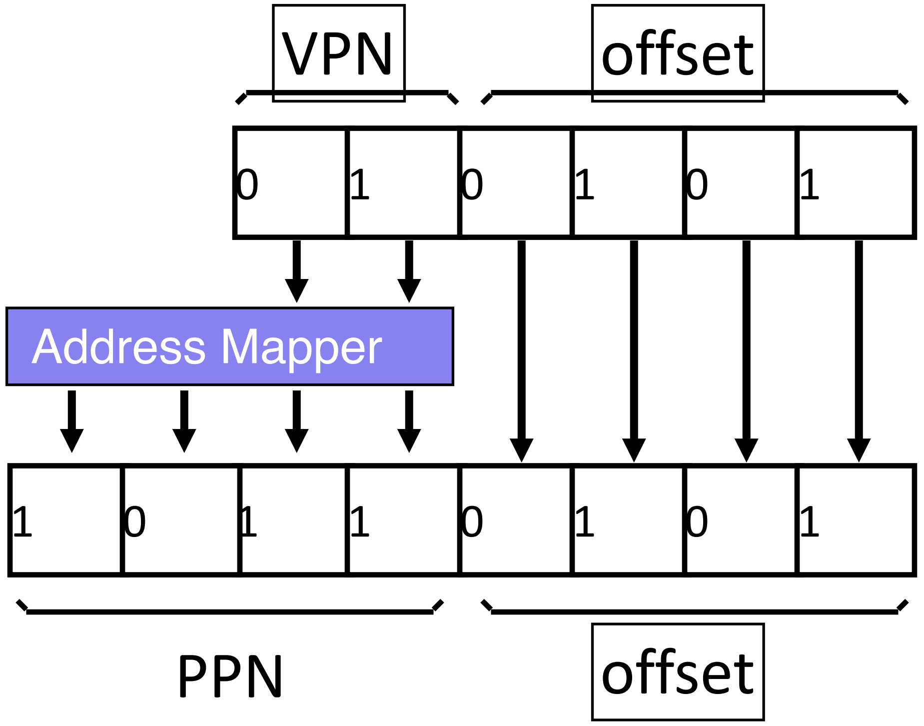


VPN to PFN Translation



Virtual => Physical PAGE Mapping

Number of bits in virtual address format does not need to equal number of bits in physical address format



How should OS translate VPN to PPN?

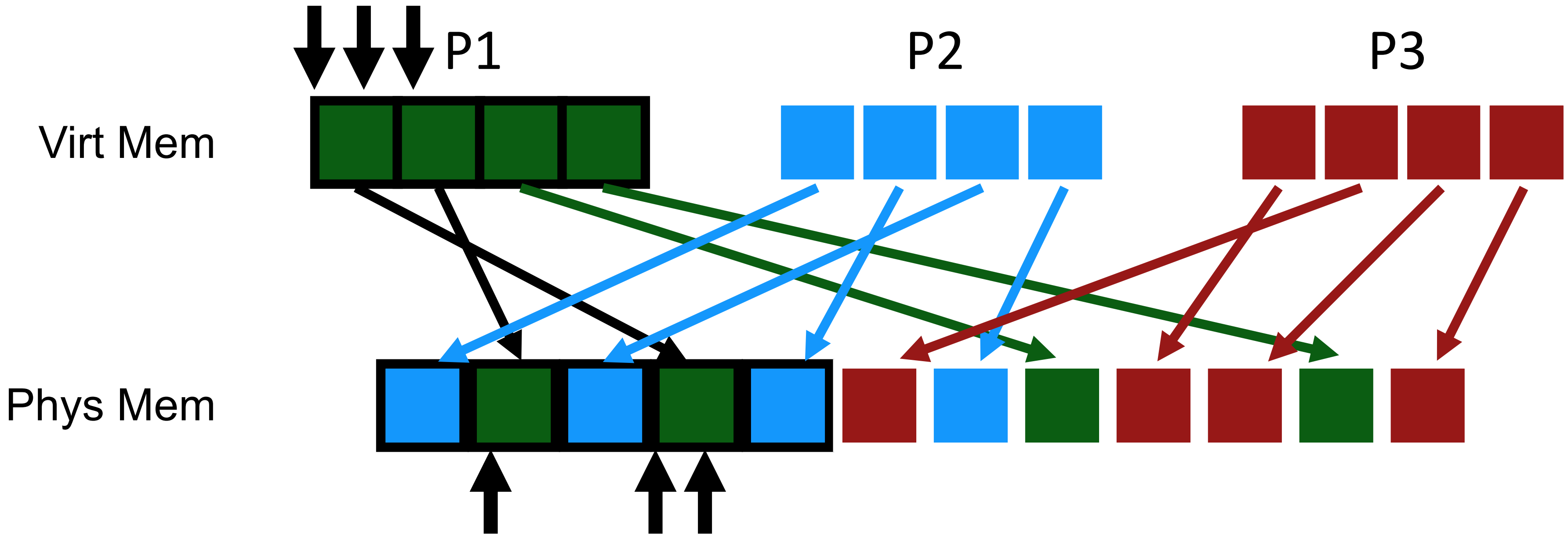
For segmentation, OS used a formula (e.g., phys addr = virt_offset + base_reg)

For paging, OS needs more general mapping mechanism

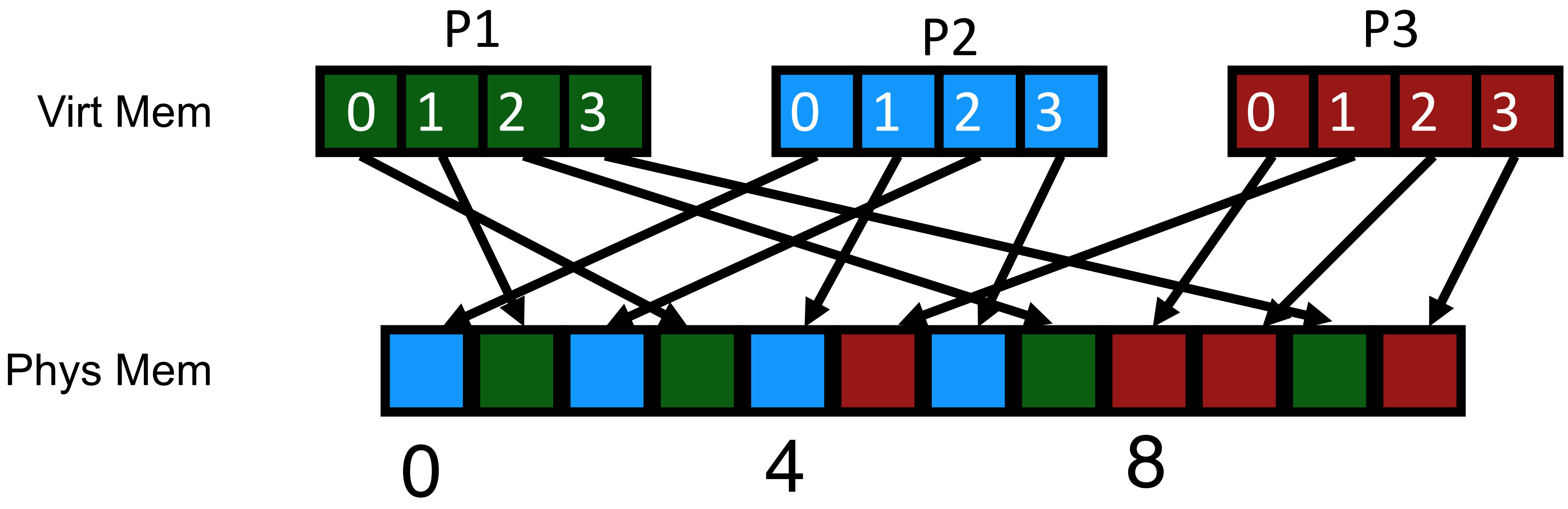
What data structure is good?

Big array: page table

The Mapping



Let’s fill in the Page Table



Page Tables:

P1	P2	P3
3	0	8
1	4	5
7	2	9
10	6	11

Where are page tables stored?

Ideally, put it in fast hardware (MMU)…

How big is a typical page table?

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

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = 2^(bits for vpn)
- Bits for vpn = 32– number of bits for page offset

  = 32 – lg(4KB) = 32 – 12 = 20
- Num entries = 2^20 = 1 MB
- Page table size = Num entries * 4 bytes = 4 MB per process

Where are page tables stored?

Implication: Store each page table in memory

- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

- Change contents of page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

Other PT info

What other info is in pagetable entries besides translation?

- valid bit
- protection bits
- present bit (needed later)
- reference bit (needed later)
- dirty bit (needed later)

Page table entries are just bits stored in memory

- Agreement between hardware and OS about interpretation

Memory Accesses with Pages

```
0x0010: movl 0x1100, %edi
0x0013: addl $0x3, %edi
0x0019: movl %edi, 0x1100
```

Assume PT is at phys addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view
of page table

2
0
80
99

Earlier: How many mem refs with segmentation?

5 (3 instrs, 2 movl)

Physical Memory Accesses with Paging?

1) Fetch instruction at logical addr 0x0010; vpn?

- Access page table to get ppn for vpn 0
- Mem ref 1: 0x5000
- Learn vpn 0 is at ppn 2
- Fetch instruction at 0x2010 (Mem ref 2)

Exec, load from logical addr 0x1100; vpn?

- Access page table to get ppn for vpn 1
- Mem ref 3: 0x5004
- Learn vpn 1 is at ppn 0
- Movl from 0x0100 into reg (Mem ref 4)

Use of a page table doubles memory references

Advantages of Paging

No external fragmentation
- Any page can be placed in any frame in physical memory

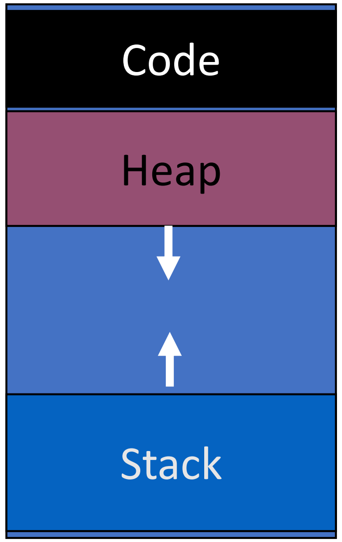
Fast to allocate and free
- Alloc: No searching for suitable free space
- Free: Doesn’t have to coallesce with adjacent free space
- Just use bitmap to show free/allocated page frames

Simple to swap-out portions of memory to disk (later lecture)
- Page size matches disk block size
- Can run process when some pages are on disk
- Add “present” bit to PTE

Disadvantages of Paging

Internal fragmentation: Page size may not match size needed by process

- Wasted memory grows with larger pages



Additional memory references → time-inefficient!

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

Storage for page tables substantial → space-inefficient!

- Simple page table: Requires PTE for all pages in address space
  - Naively, page table entry needed even if page not allocated
- Problematic with dynamic stack and heap within address space
- Page tables must be allocated contiguously in memory
  - Due to linear access of page table entries

Reducing Page Table sizes

How big are page tables?

1. PTE's are **2 bytes**, and **32** possible virtual page numbers

   32 * 2 bytes = 64 bytes

2. PTE's are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**

   2 bytes * 2^(24 – lg 16) = **2^21 bytes** (2 MB)

3. PTE's are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**

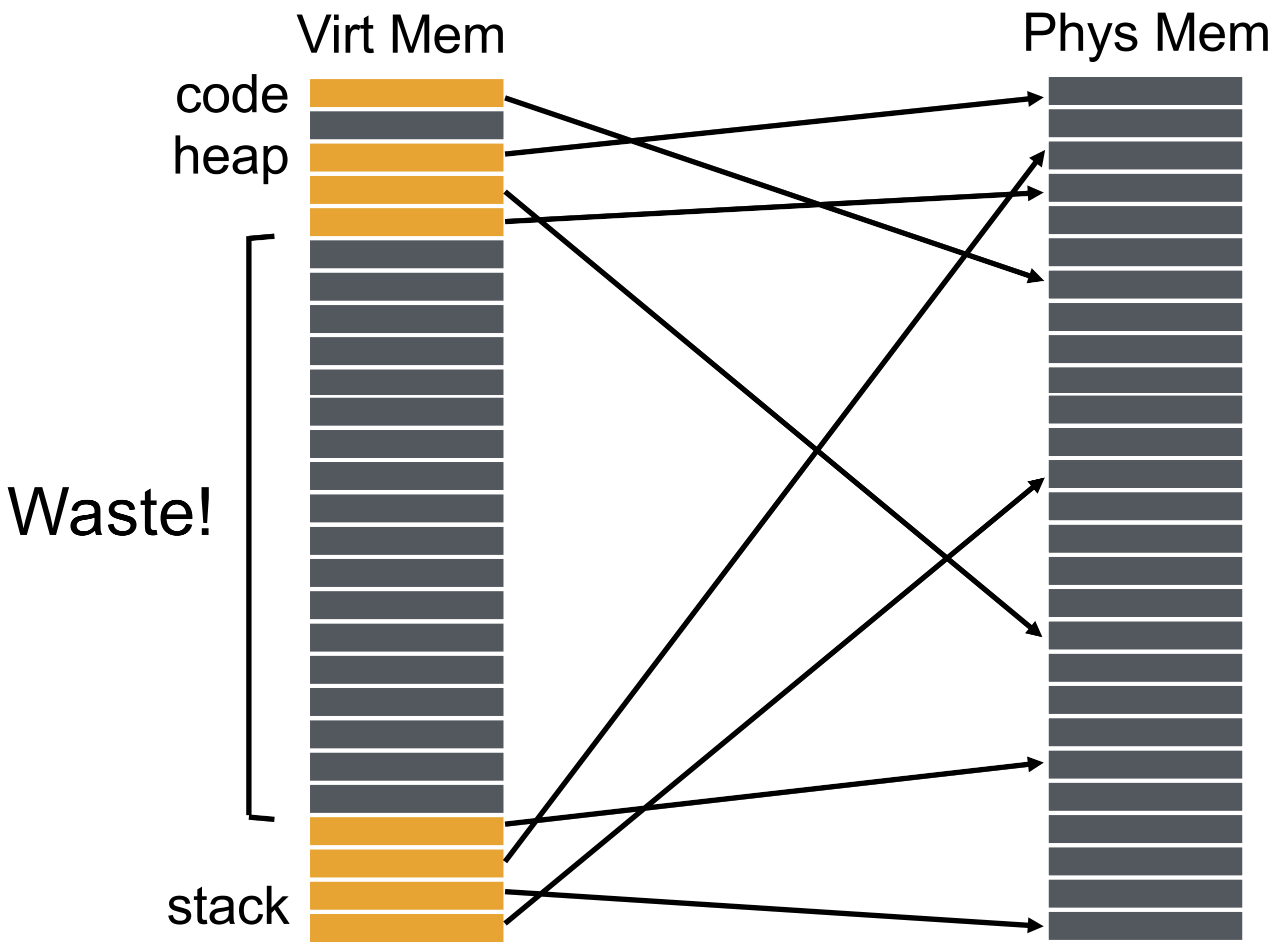
   4 bytes * 2^(32 – lg 4K) = **2^22 bytes** (2 MB)

4. PTE's are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

   4 bytes * 2^(64 – lg 4K) = **2^54 bytes**

How big is each page table?

Why ARE Page Tables so Large?



Many invalid page table entries

Format of linear page tables:

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

how to avoid storing these?

Avoid the simple linear page table

Use more efficient (but complex) data structures, instead of the simple big array

Any data structure is possible in principle*

*assuming software managed TLB

Some approaches

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

Approach 1: Inverted Page Table

Inverted Page Tables

- 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

Valid PTEs are Contiguous

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

Use ideas from segmentation!

Approach 2: Segmented Page Tables

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)

Ideas

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

Combining Paging and Segmentation

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

seg	base	bounds	R W
0	0x002000	0xff (255)	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f (15)	1 1

Page table

...	
0x01f	0x001000
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	

```
0x002070 read:   0x004070
0x202016 read:   0x003016
0x104c84 read:   error
0x010424 write:  error
0x210014 write:  error
0x203568 read:   0x02a568
```

Advantages of Segments

- Supports sparse address spaces
  - Decreases size of page tables
  - If segment not used, not needed 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 with 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

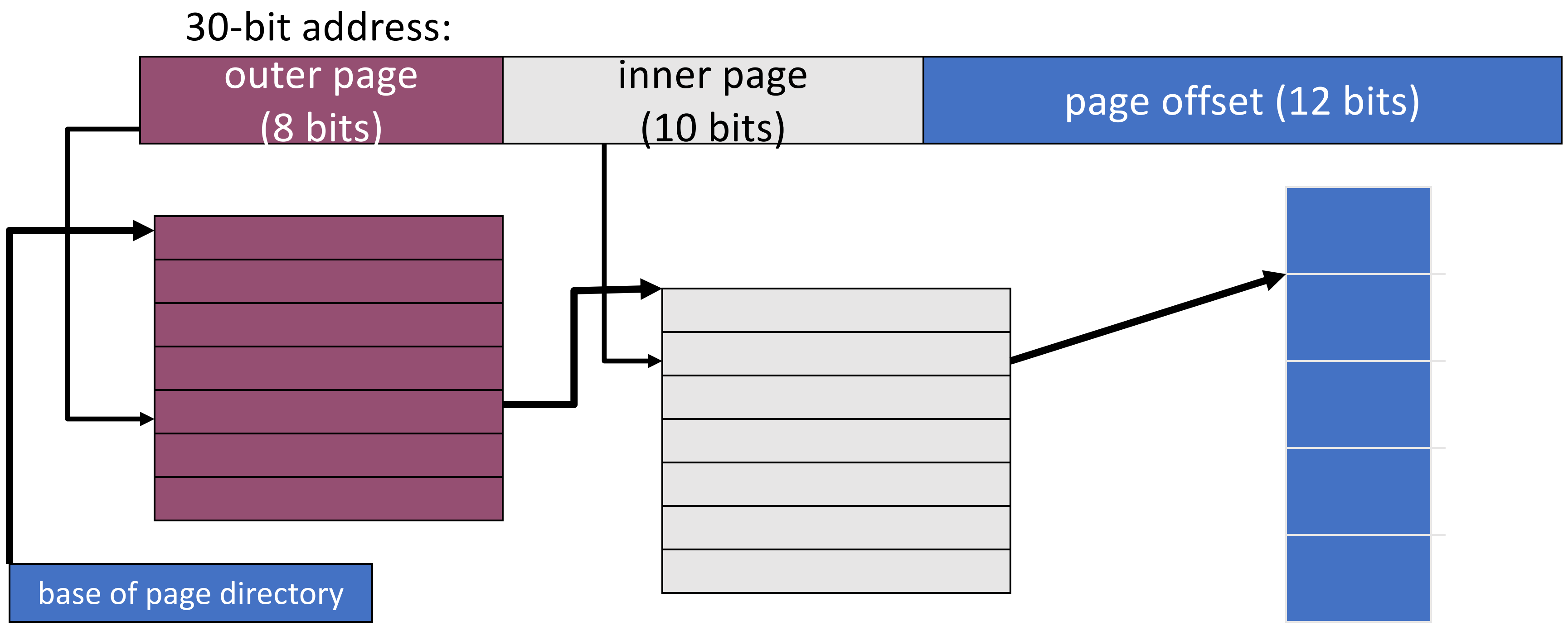
1. Inverted Pagetables
2. Segmented Pagetables
3. Multi-level Pagetables
   - Page the page tables
   - Page the pages of page tables…

3) Multilevel Page Tables

Goal: Allow 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 example

page directory

VPN	PPN	valid
0	0x3	1
1	-	0
2	-	0
-	-	0
-	-	0
-	-	0
-	-	0
-	-	0
-	-	0
-	-	0
-	-	0
-	-	0
-	-	0
-		0
15	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

0x23ABC

translate 0x00000

0x10000

translate 0xFEED0

0x55ED0

20-bit address:

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

Address format for Multilevel Paging

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
  - number PTE per page = (2^12 bytes per page) / (4 bytes per PTE)
  - → 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 directory (outer level) may not fit in a page!

Solution:

outer page?	inner page (10 bits)	page offset (12 bits)

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

← VPN →

PD idx 0	PD idx 1	PT idx	OFFSET

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

Review: Paging pros and cons

Advantages

- No external fragmentation
  - don’t need to find contiguous RAM
- All free pages are equivalent
  - Easy to manage, allocate, and free pages

Disadvantages

- Page tables are too big
  - Must have one entry for every page of address space
- Accessing page tables is too slow [address this shortly]
  - Doubles the number of memory references per instruction

Translation Steps

H/W: for each mem reference:

(cheap) 1. extract **VPN** (virt page num) from **VA** (virt addr)

(cheap) 2. calculate addr of **PTE** (page table entry)

(expensive) 3. read **PTE** from memory

(cheap) 4. extract **PFN** (page frame num)

(cheap) 5. build **PA** (phys addr)

(expensive) 6. read contents of **PA** from memory into register

Which steps are expensive?

Which expensive step(s) can we (not) avoid?

3) Let’s try to avoid having to read PTE from memory!

Translation Lookaside Buffers

How can page translations be made faster?
What is the basic idea of a TLB (Translation Lookaside Buffer)?
What types of workloads perform well with TLBs?
How do TLBs interact with context-switches? (if time permits)

Example: Array Iterator

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

Assume ‘a’ starts at 0x3000

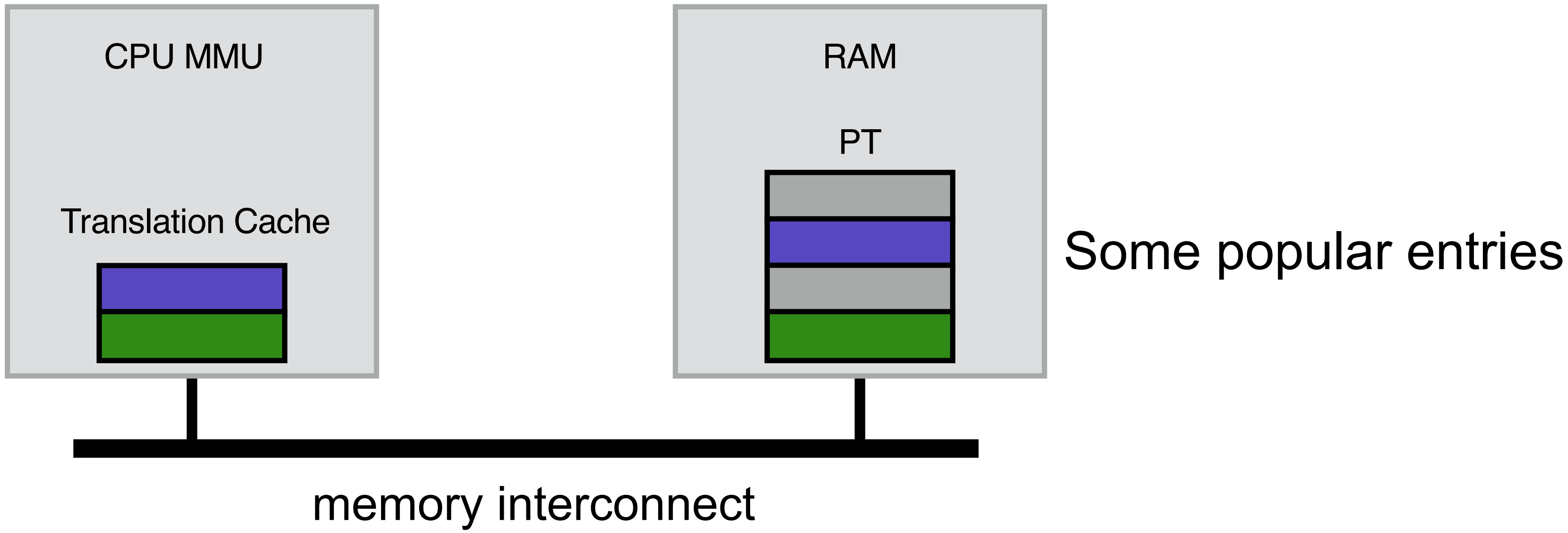
Ignore instruction fetches

What virtual addresses?	What physical addresses?
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
...	load 0x700C

Observation:

Repeatedly access same PTE because program repeatedly accesses same virtual page

Strategy: Cache Page Translations

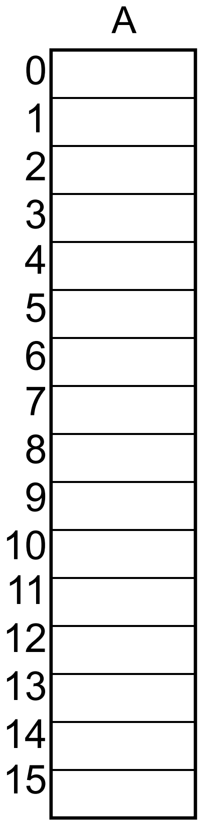


TLB: **T**ranslation **L**ookaside **B**uffer

TLB Organization

TLB Entry

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



Direct mapped (num sets = 16)

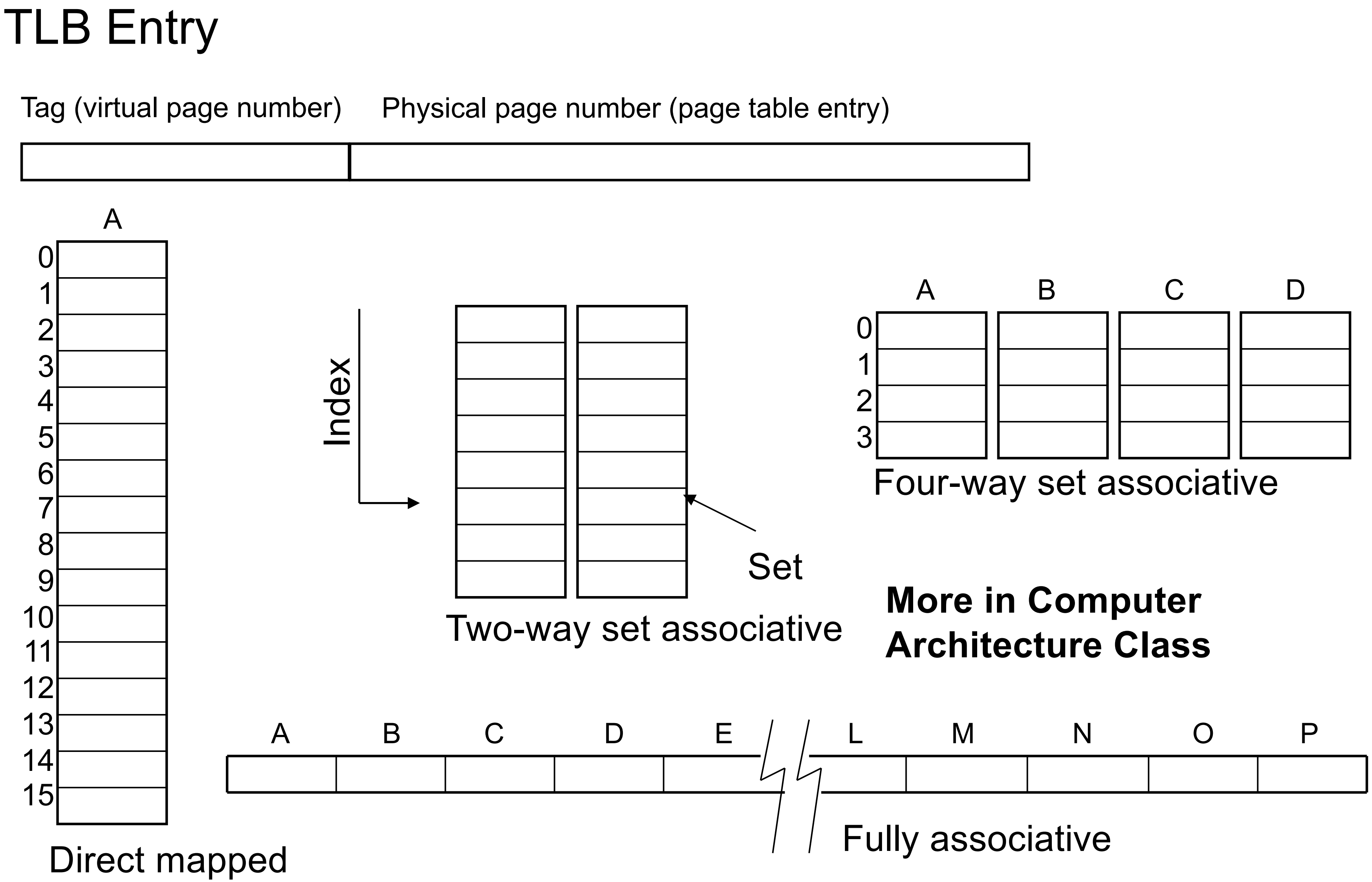
Lookup

- Calculate set (tag % num_sets)
- Search for tag within resulting set

Where is VPN (tag) 18 located?

2

TLB Organization



More in Computer Architecture Class

TLB Associativity Trade-offs

Higher associativity

+ Better utilization, fewer collisions
– Slower
– More hardware

Lower associativity

+ Fast
+ Simple, less hardware
– Greater chance of collisions

TLBs usually fully associative

Array Iterator (with TLB)

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

Assume following virtual address stream:
load 0x1000

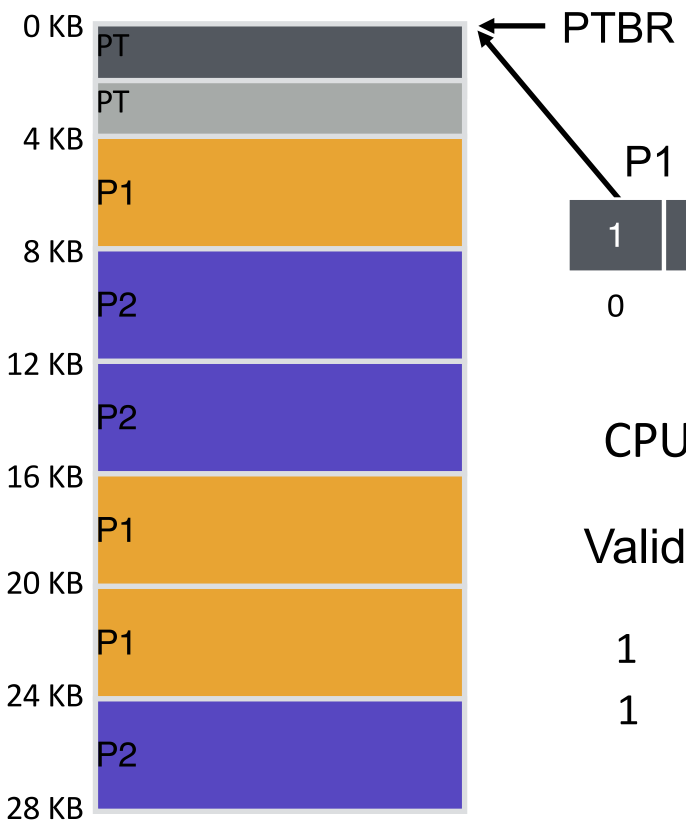
load 0x1004

load 0x1008

load 0x100C
…

What will TLB behavior look like?

TLB Accesses: Sequential Example



P1 pagetable

0	1	2	3
1	5	4	…

CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virtual	Physical
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB hit) load 0x5004
load 0x1008	(TLB hit) load 0x5008
load 0x100c	(TLB hit) load 0x500C
…	…
load 0x2000	load 0x0008 load 0x4000
load 0x2004	(TLB hit) load 0x4004

Performance Of TLB?

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

Calculate miss rate of TLB for data:
TLB misses / # TLB lookups

TLB lookups?
= number of accesses to a = 2048

TLB misses?
= number of unique pages accessed
= 2048 / (elements of ‘a’ per 4K page)
= 2K / (4K / sizeof(int)) = 2K / 1K = 2

Miss rate?
2/2048 = 0.1%

Hit rate? (1 – miss rate)
99.9%

Would hit rate get better or worse with smaller pages?
Worse

TLB

How can system improve TLB performance (hit rate) given fixed number of TLB entries?

Increase page size

- Fewer unique page translations needed to access same amount of memory

TLB “reach” in terms of physical memory size:

- Number of TLB entries * Page Size

“Huge pages” used in many real systems…

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

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

Workload B

```c
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];
}
```

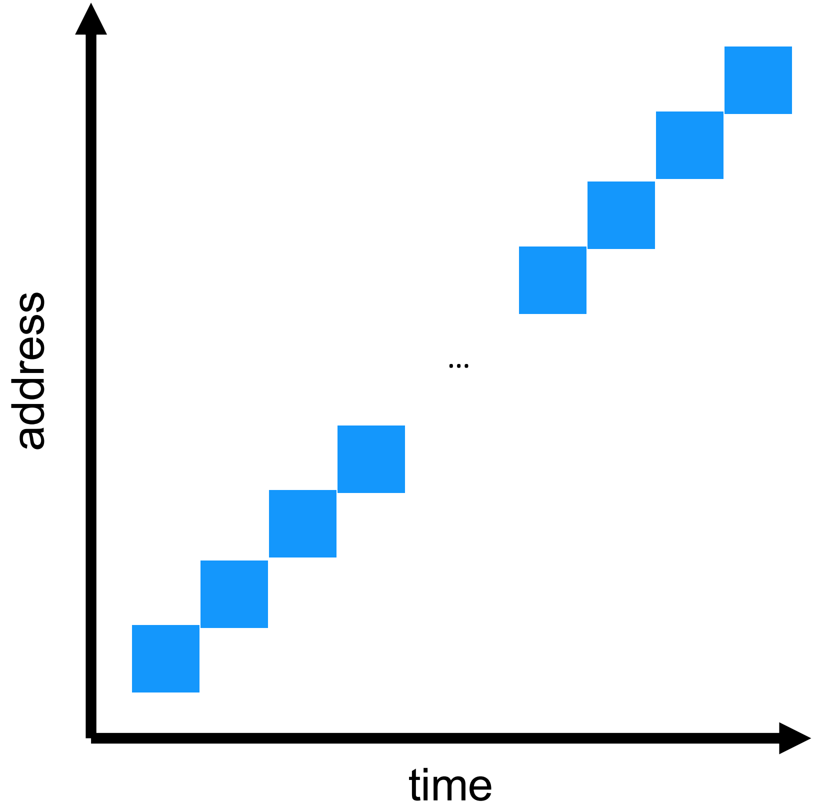
Workload
Access Patterns

Workload A

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

Spatial Locality

Sequential Accesses



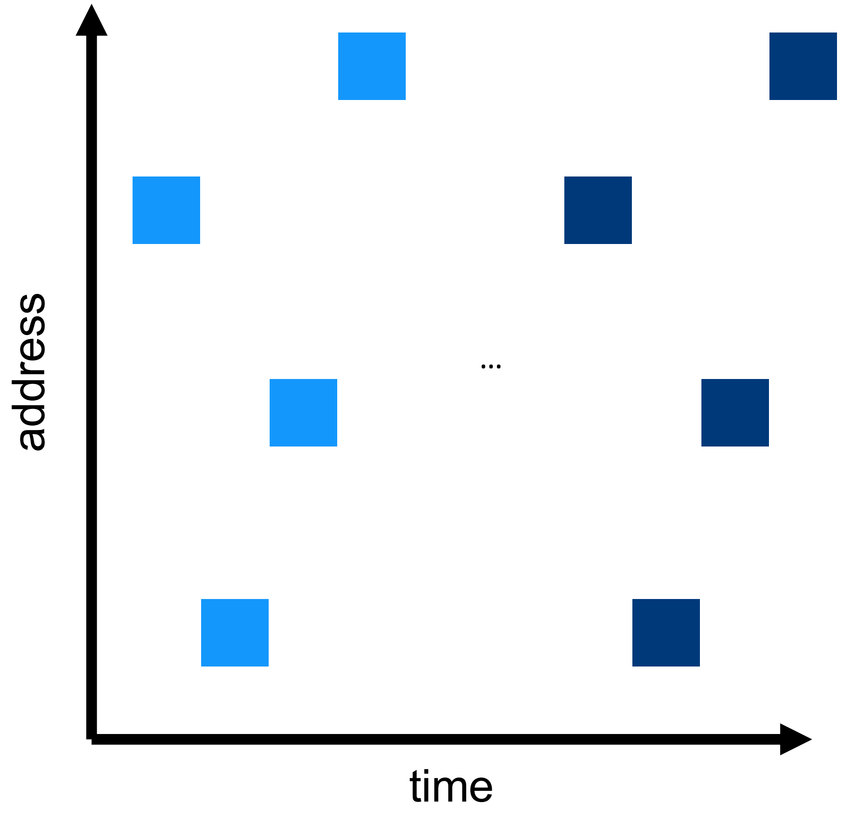
Workload
Access Patterns

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];
}
```

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 → PFN 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?

Differentiating processes

- So far, we assumed VPNs are unique. They are not!
- Option 1: Flush TLBs upon every context switch (valid = 0)
  - Problem: poor performance after each context switch
- Option 2: Attach “address space identifier” to TLB entry

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

A full system with TLBs

On TLB miss: lookups with more paging levels more expensive

How much does a miss cost?

Assume 3-level page table, 256-byte pages, 16-bit addresses
Assume ASID of current process is 211
How many physical accesses for each instruction?

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

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

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

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

0xaa: (TLB miss -> 3 for addr trans) + 1 instr fetch
0x11: (TLB miss -> 3 for addr trans) + 1 movl

Total: 8

0xbb: (TLB hit -> 0 for addr trans) + 1 instr fetch from 0x9113

Total: 1

0x05: (TLB miss -> 3 for addr trans) + 1 instr fetch
0xff: (TLB hit -> 0 for addr trans) + 1 movl into 0x2310

Total: 5

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 data structure that hardware knows how to “walk”

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

Next Topic: What if desired address spaces do not fit in physical memory?