

MEMORY: TLBS

Andrea Arpaci-Dusseau
CS 537, Fall 2019

ADMINISTRIVIA

- Project 1 graded out of 120 points
 - Details in 3 files in your handin/pl directory
 - Contact TA (yifan) if significant grading discrepancy
- Project 2 due last night
- Project 3 available: Shell in Linux
 - Subject of tomorrow's discussion sections (fork() and exec())
- Midterm 1: Thursday, Oct 10th from 7:30-9:30pm
 - Fill out Exam Conflict form in Canvas by Thursday Sept 26
 - Fill out form again if filled out for old date of Oct 9th if still conflict
 - Next discussion sections on lecture review? Post sample exam soon
- Canvas Homeworks
 - Due each Tuesday and Thursday

AGENDA / LEARNING OUTCOMES

Memory virtualization

Review paging...

How can page translations be made faster?

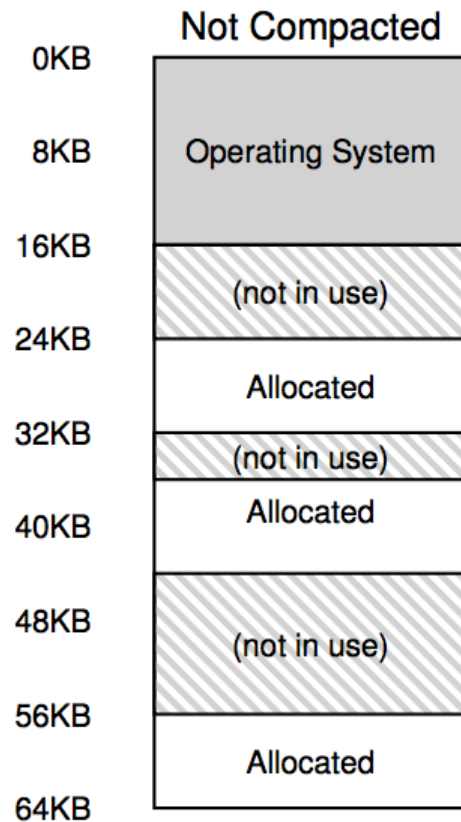
What is the basic idea of a TLB (Translation Lookaside Buffer)?

What types of workloads perform well with TLBs?

How do TLBs interact with context-switches?

RECAP

FRAGMENTATION

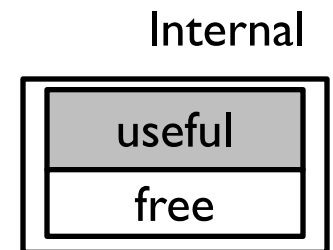


Definition: Free memory that can't be usefully allocated

Types of fragmentation

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

Internal: Visible to requester



PAGING

Goal: Eliminate requirement that address space is contiguous

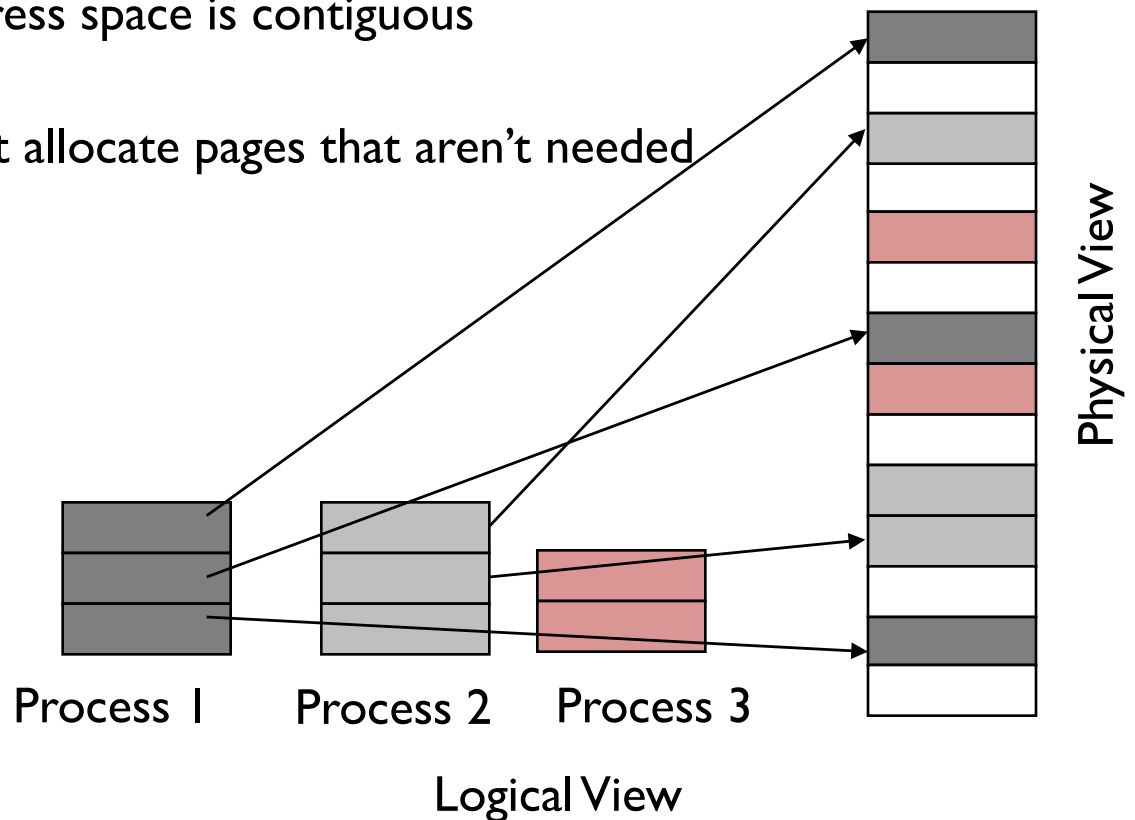
Eliminate external fragmentation

Grow segments as needed – Don't allocate pages that aren't needed

Idea:

Divide address spaces and physical memory into fixed-sized pages

Size: 2^n , Example: 4KB



PAGETABLES

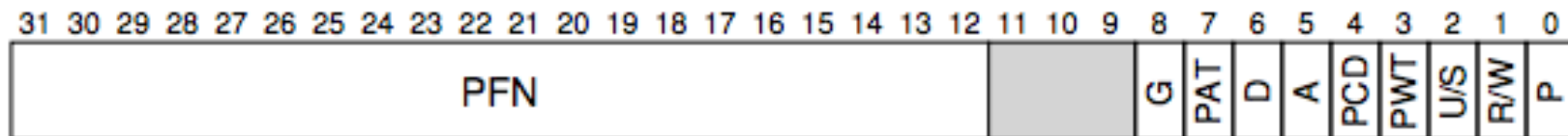
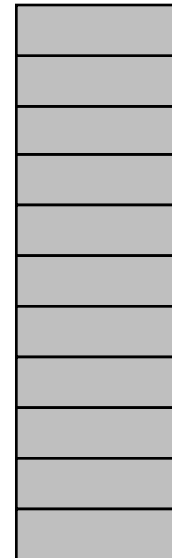
What is a good data structure ?

Simple solution: Linear page table aka *array*

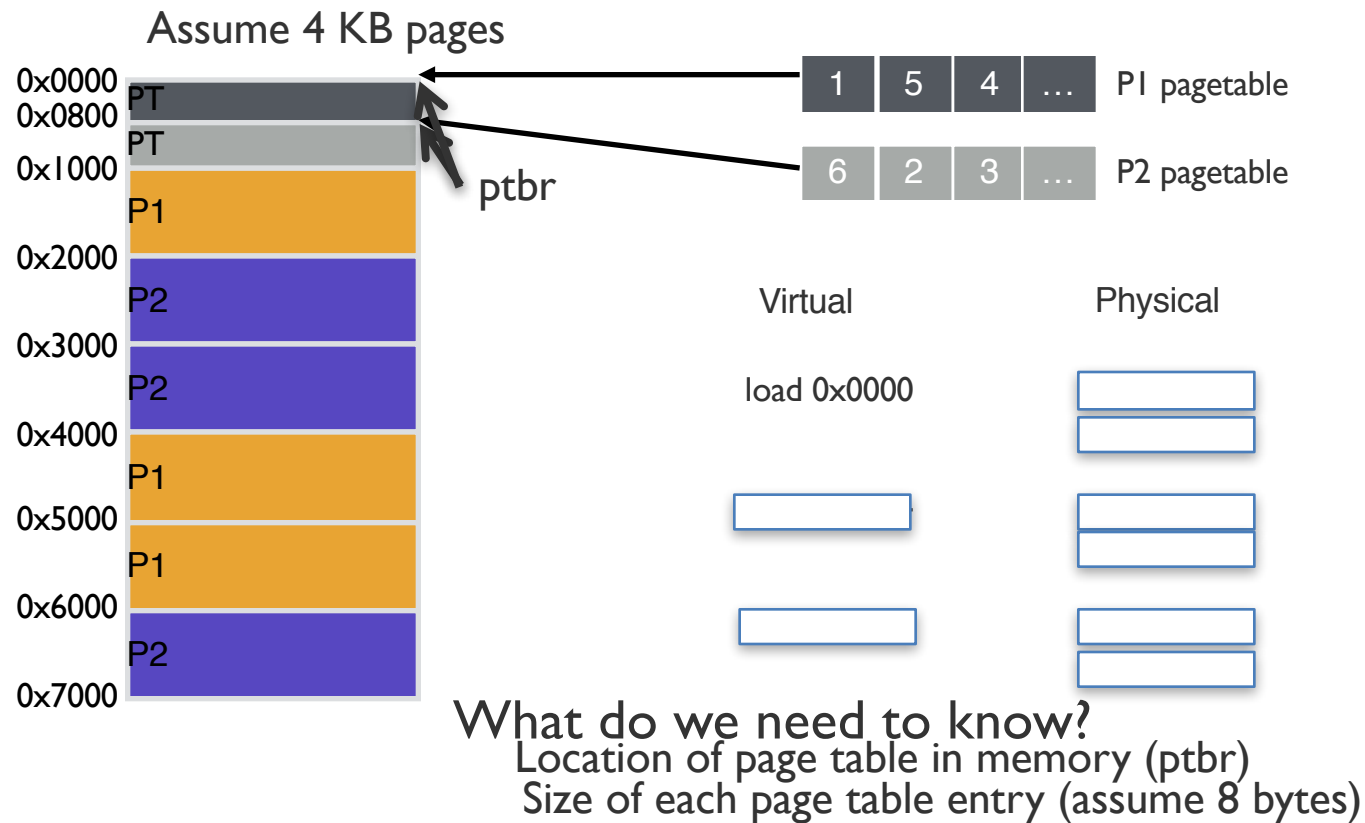
VPN

0

2^n



REVIEW: PAGING



1 MINUTE CHAT: PAGING TRANSLATION STEPS

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 cheap and which are expensive?

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

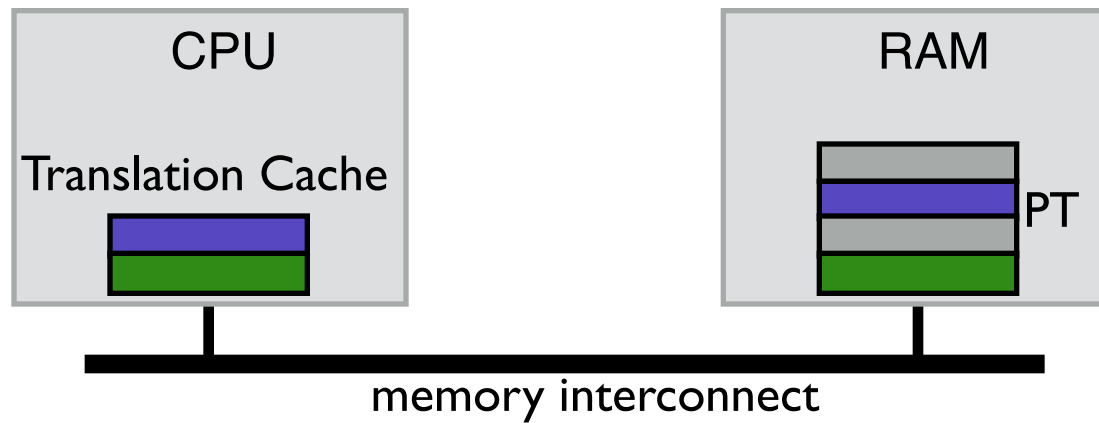
Observation:

Repeatedly access same PTE because program
repeatedly accesses same virtual page

Aside: What can you infer? Why???

- ptbr: 0x1000; PTE 4 bytes each
- VPN 3 -> PPN 7

STRATEGY: CACHE PAGE TRANSLATIONS



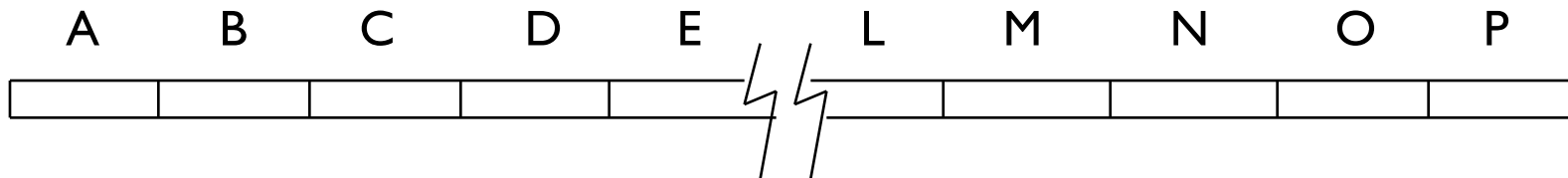
TLB: **T**ranslation **L**ookaside **B**uffer

TLB: TRANSLATION LOOKASIDE BUFFER

TLB ORGANIZATION

TLB Entry

Tag (virtual page number)	Physical page number (page table entry)	Protection bits - rwx
---------------------------	---	-----------------------



Fully associative

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

Operations:
Lookup and Replacement

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

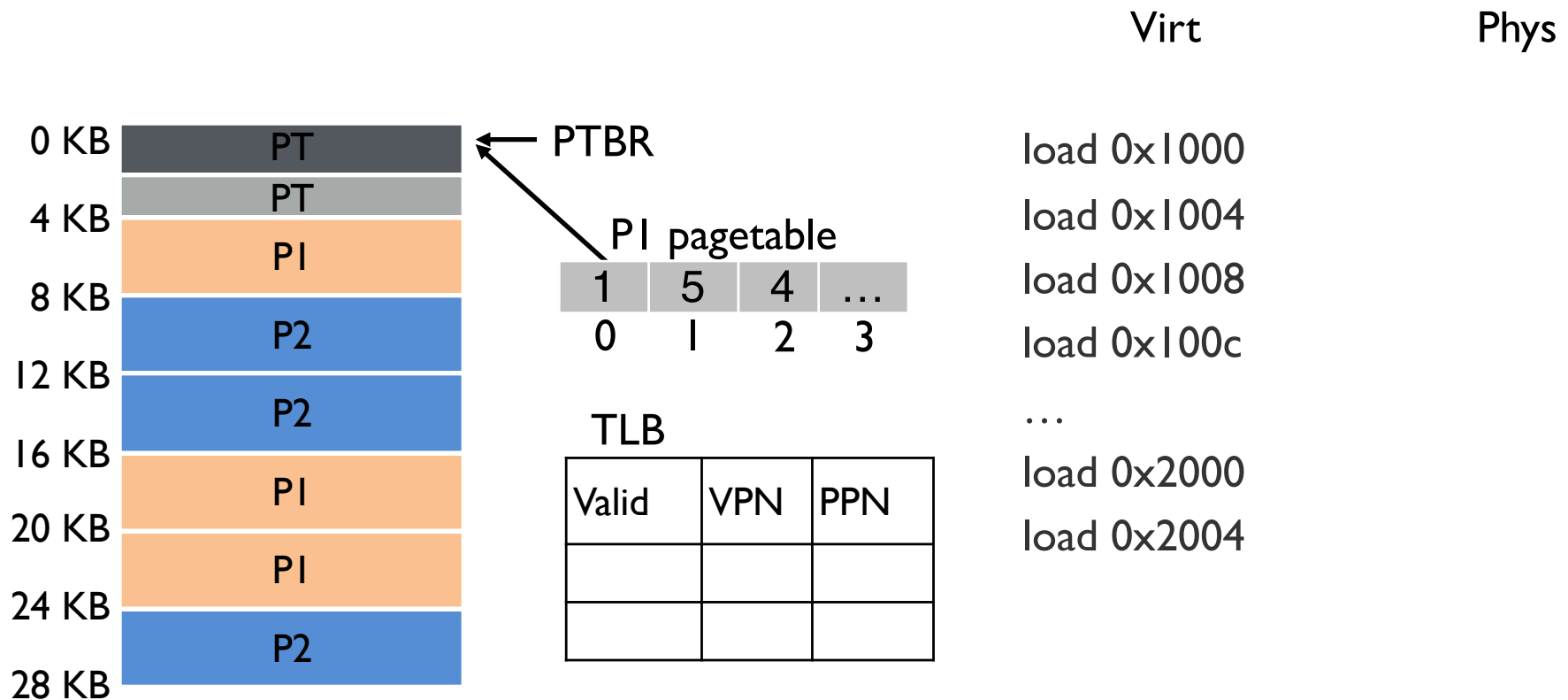
load 0x1008

load 0x100C

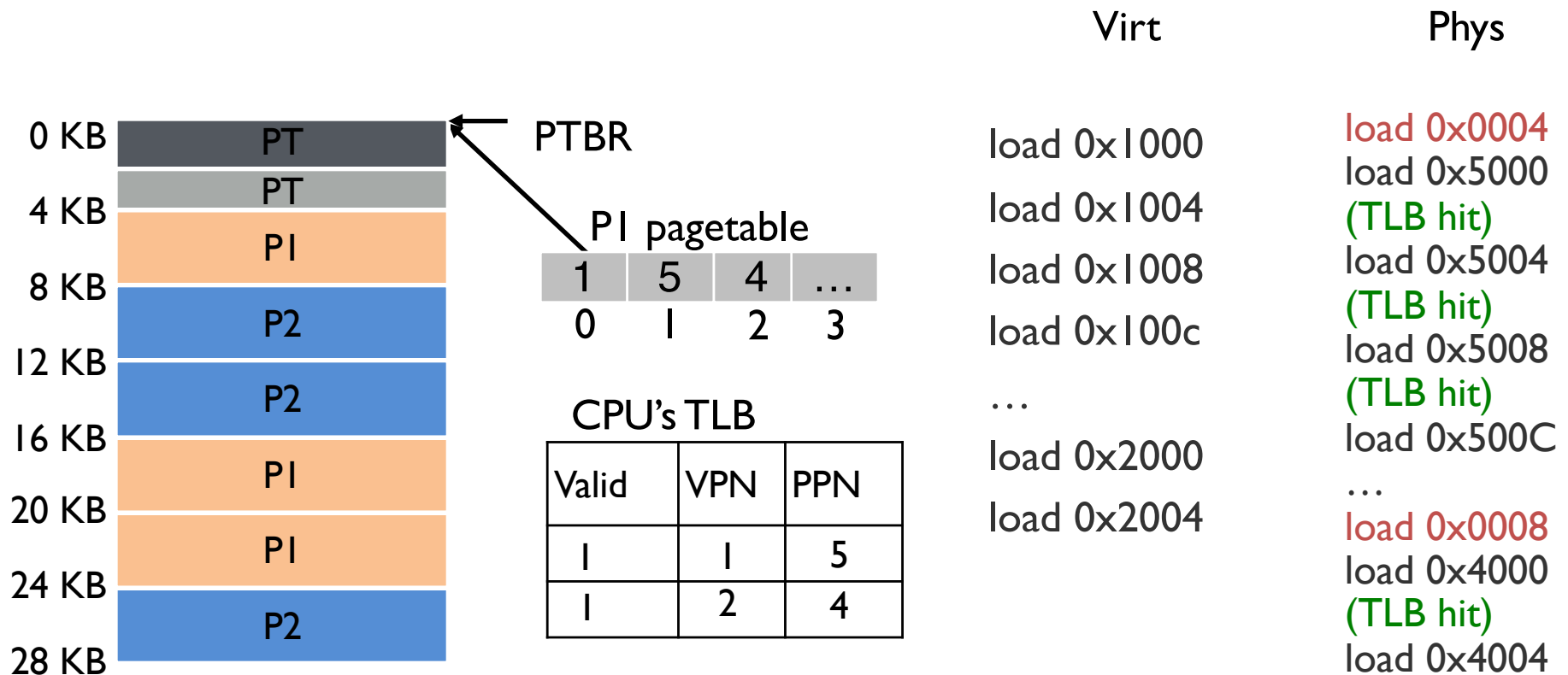
...

What will TLB behavior look like?

TLB ACCESSES: SEQUENTIAL EXAMPLE



TLB ACCESSES: SEQUENTIAL EXAMPLE



PERFORMANCE OF TLB?

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

Would hit rate get better or worse
with more iterations?

Stay same!

Miss first access to each page

Always miss 1/1024

Calculate **miss rate** of TLB for data (ignore code + sum)

#TLB misses / #TLB lookups

#TLB lookups?

= number of accesses to array a[]

= 2048

#TLB misses?

= number of unique pages accessed

= 2048 / (elements of a[] per 4K page)

= 2K / (4KB / sizeof(int)) = 2K / 1K = 2 pages

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 PERFORMANCE

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:

Number of TLB entries * Page Size

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

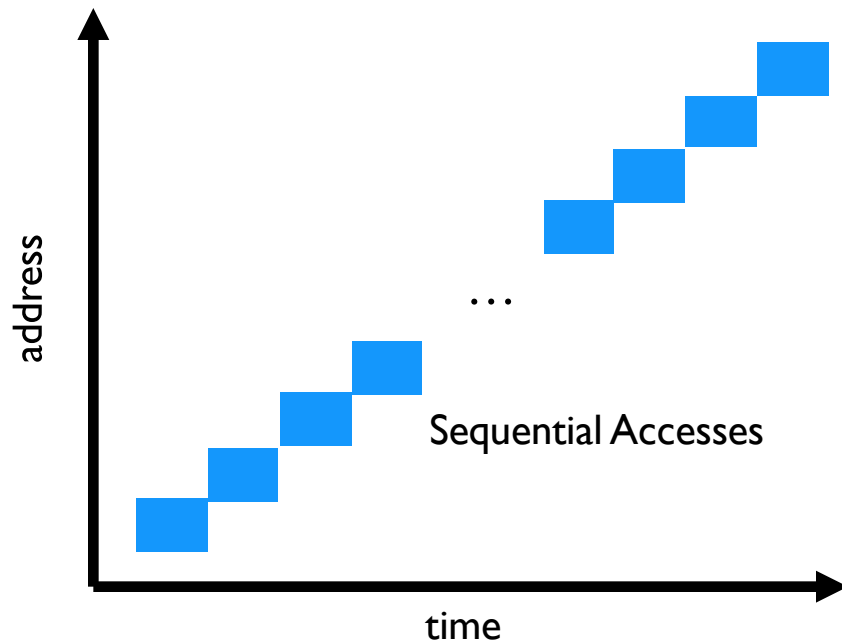
```
int sum = 0;  // large N

srand(1234);
for (i=0; i<1024; i++) {
    sum += a[rand() % N];
}

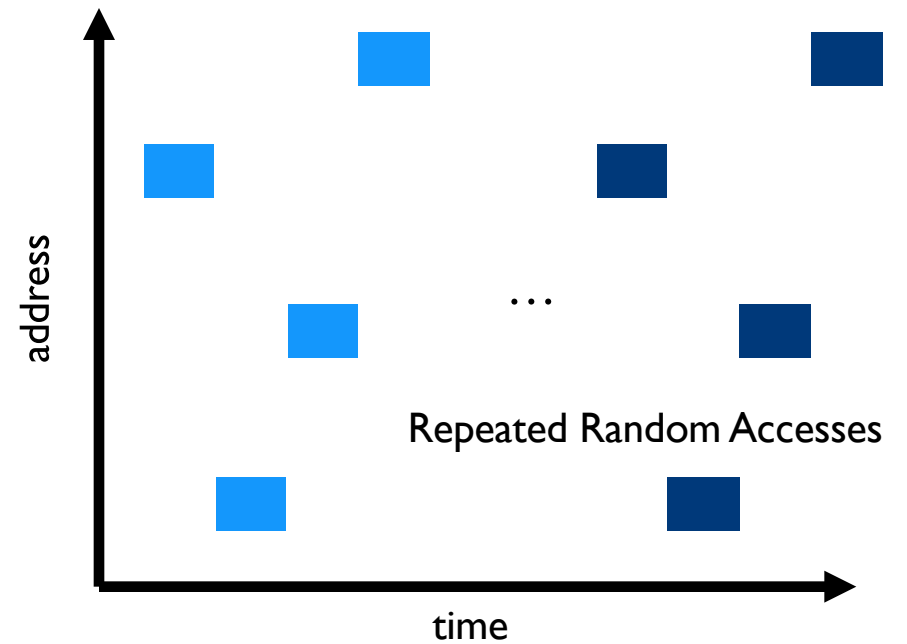
srand(1234);
for (i=0; i<1024; i++) {
    sum += a[rand() % N];
}
```

WORKLOAD ACCESS PATTERNS

Spatial Locality



Temporal Locality



WORKLOAD LOCALITY

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

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

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?

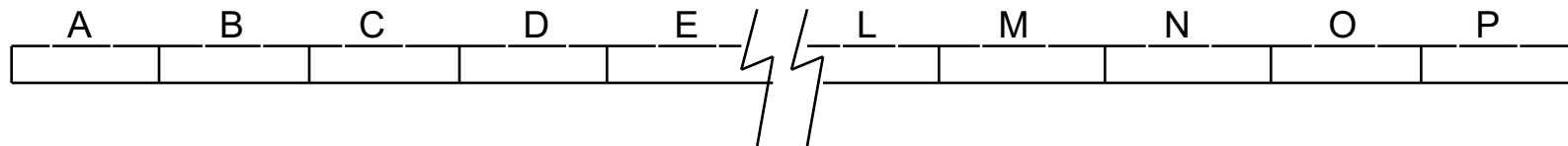
TLB REPLACEMENT POLICIES

More entries in TLB -> More likely to contain entries accessed in past

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?

2-MINUTE CHAT: LRU TROUBLES

Initial TLB			TLB after 4 accesses			
Valid	VPN	PFN		Valid	VPN	PFN
0	?	?		1	0	?
0	?	?		1	1	?
0	?	?		1	2	?
0	?	?		1	3	?
0	?	?				

Workload repeatedly accesses same offset (0x001) across 5 pages (strided access), but only 4 TLB entries

What will TLB contents be over time?

How will TLB perform? **Always misses! 100% miss rate**

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!

TLB PERFORMANCE FOR CODE?

- Code tends to be relatively sequential
 - Branch for if statements and while loops often in same page
 - Good spatial locality
- Procedure calls depend on temporary locality
- Code usually has reasonable TLB performance

CONTEXT SWITCHES

What happens if a process uses TLB entry from another process?

Solutions:

1. Flush TLB on each context switch

Poor performance: lose all recently cached translations, increases miss rate

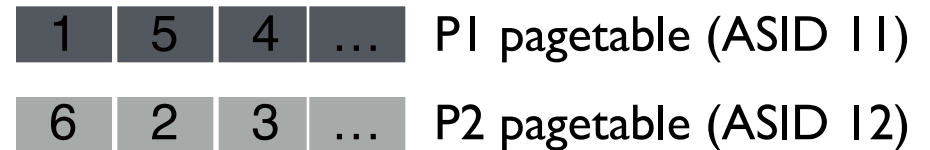
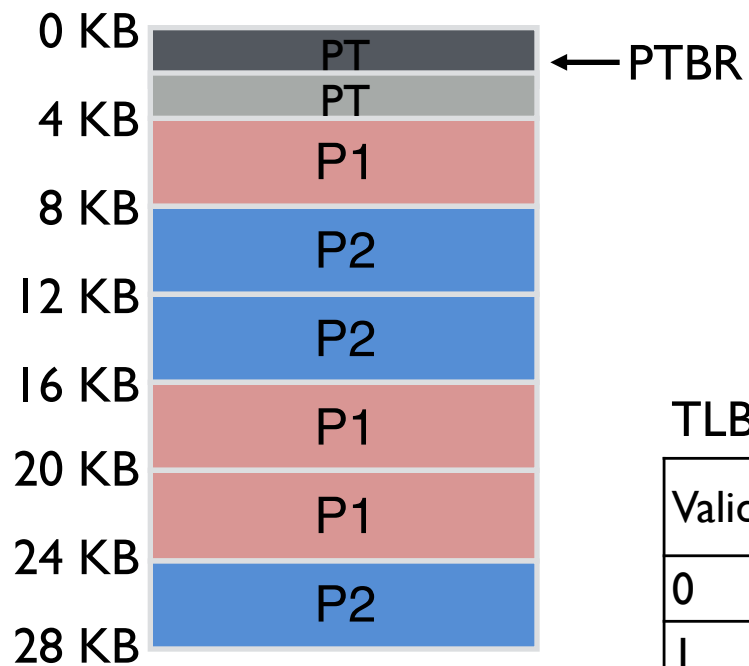
2. Track which TLB entries are for which process

- Address Space Identifier (ASID) – similar to PID

- Tag each TLB entry with 8-bit ASID; How many ASIDs do we get?

- Must match ASID for TLB entry to be used

TLB EXAMPLE WITH ASID



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

TLB:

Valid	Virt	Phys	ASID
0	3	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 Hit?

Who Handles TLB Miss? HW or OS

H/W

H/W must know where pagetables are stored in memory

- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW “walks” known pagetable structure 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; any data structure possible
Modifying TLB entries is privileged instruction

CHARACTERISTICS OF TLBS

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 (if enough TLB entries)

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 ?