Department of Physics,
Computer Science & Engineering

CPSC 410 – Operating Systems I

Virtualizing Memory: Faster with TLB

Keith Perkins

Adapted from “CS 537 Introduction to Operating Systems” Arpaci-Dusseau

Questions answered in this lecture:

Review paging…

How can page translations be made faster?

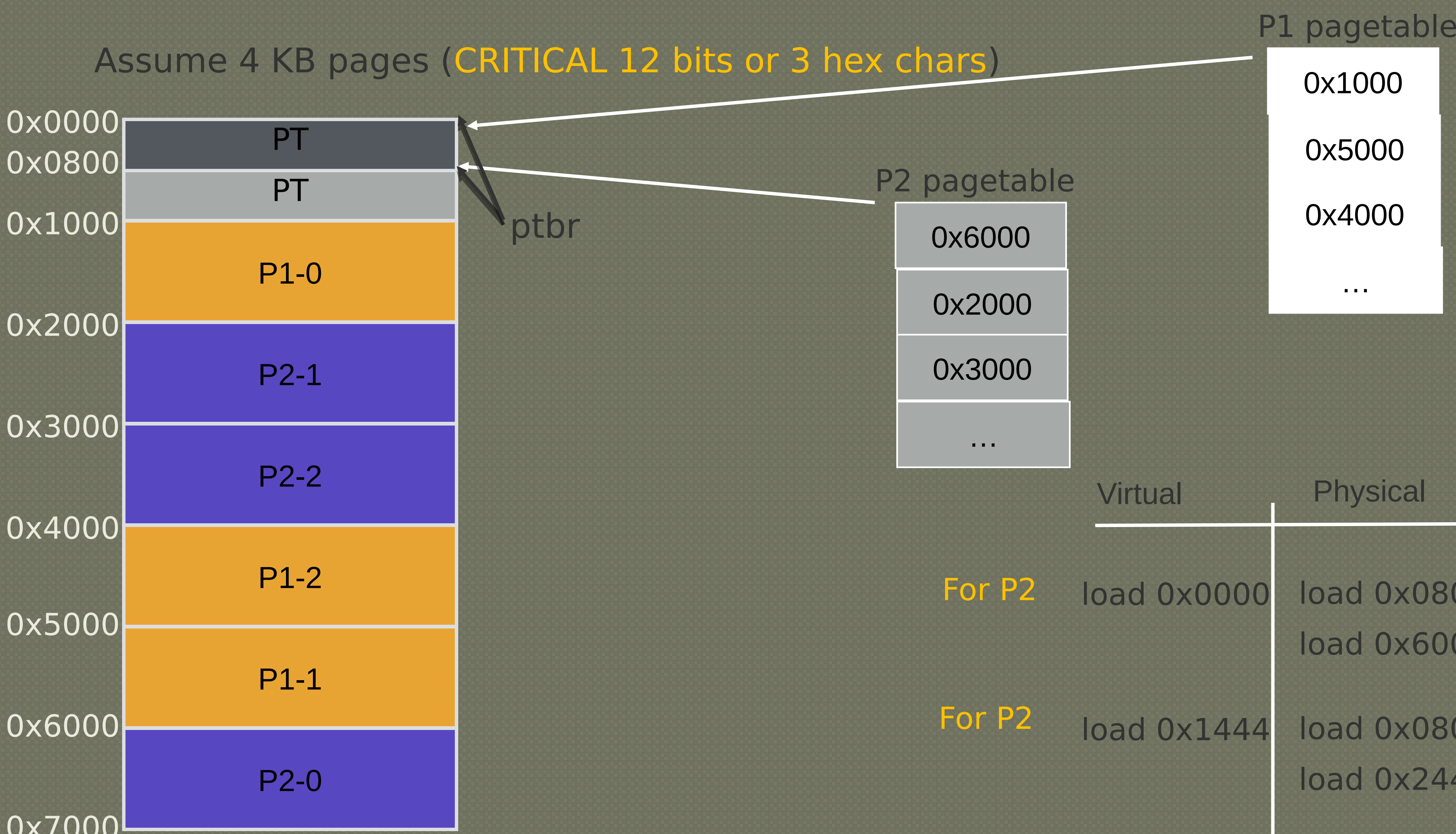
What is the basic idea of a TLB (Translation Lookaside Buffer)?

What types of workloads perform well with TLBs?

How do TLBs interact with context-switches?

Review: Paging

Assume 4 KB pages (CRITICAL 12 bits or 3 hex chars)



	Virtual	Physical
For P2	load 0x0000	load 0x0800 load 0x6000
For P2	load 0x1444	load 0x0808 load 0x2444
For P1	load 0x1444	load 0x0008 load 0x5444

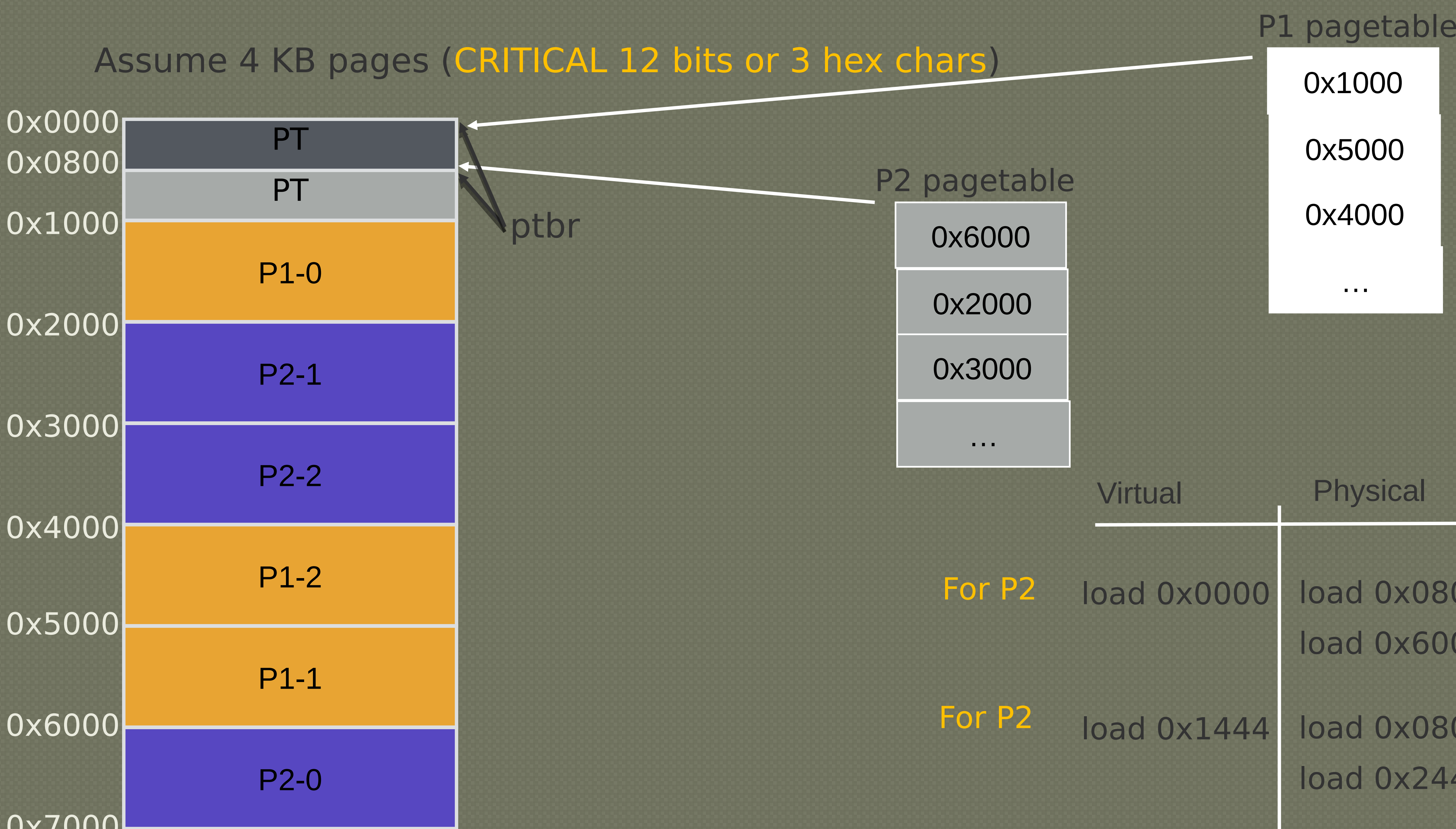
What do we need to know?

Location of page table in memory (ptbr)

Size of each page table entry (assume 8 bytes)

Review: Paging

Assume 4 KB pages (CRITICAL 12 bits or 3 hex chars)



	Virtual	Physical
For P2	load 0x0000	load 0x0800
		load 0x6000
For P2	load 0x1444	load 0x0808
		load 0x2444
For P1	load 0x1444	load 0x0008
		load 0x5444

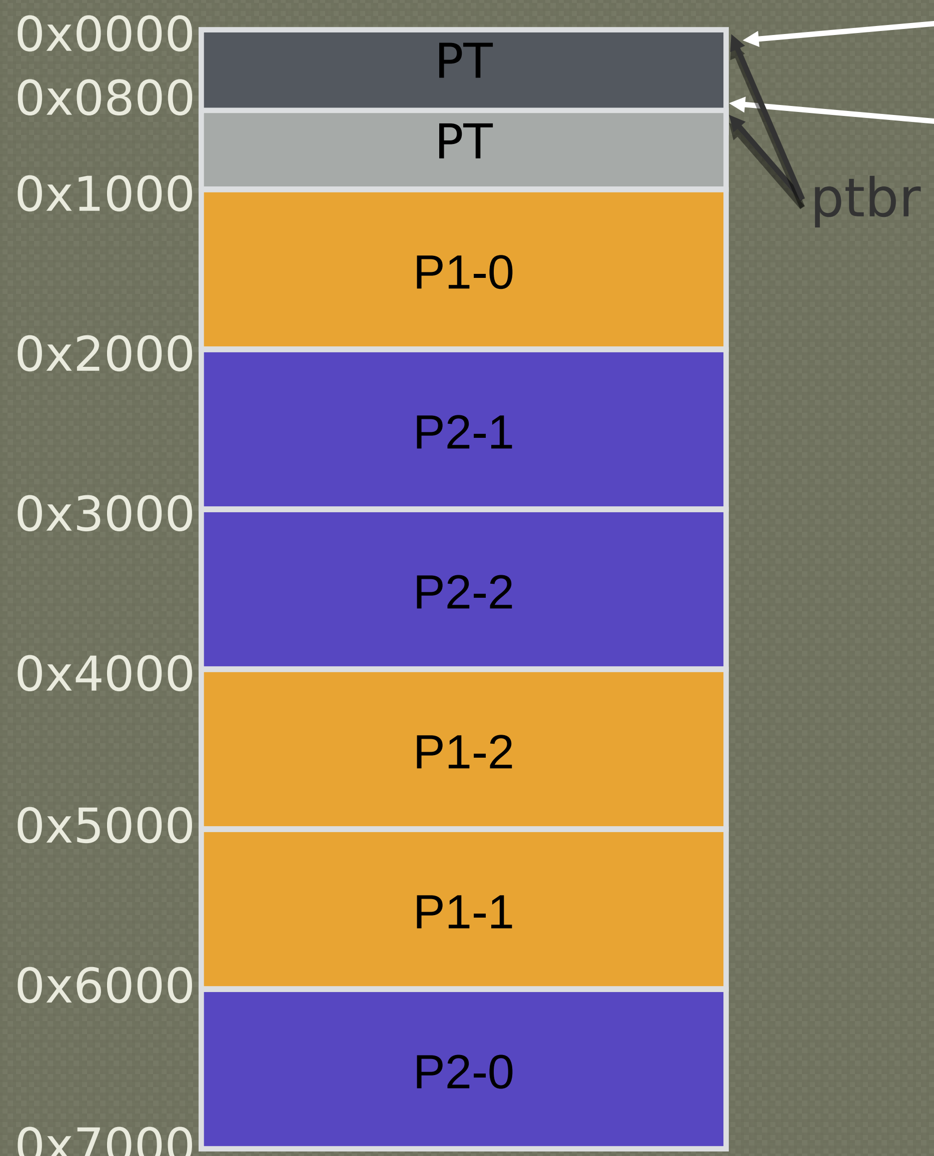
What do we need to know?

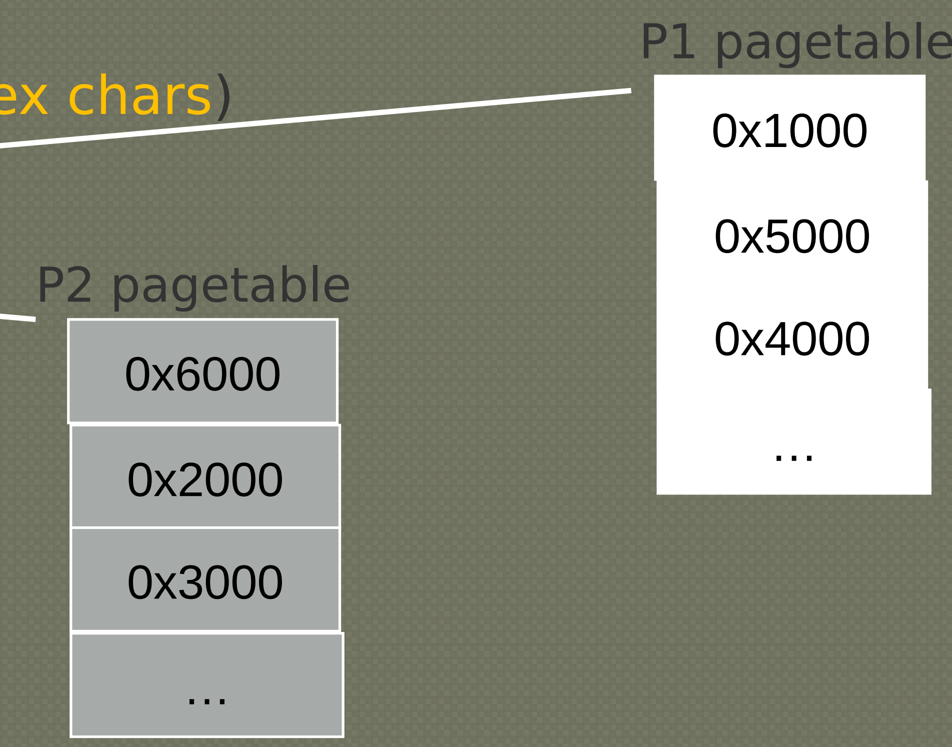
Location of page table in memory (ptbr)

Size of each page table entry (assume 8 bytes – What if 4 bytes?)

Review: Paging

Assume 4 KB pages (CRITICAL 12 bits or 3 hex chars)





	Virtual	Physical
For P2	load 0x0000	load 0x0800
		load 0x6000
For P2	load 0x1444	load 0x0804
		load 0x2444
For P1	load 0x1444	load 0x0004
		load 0x5444

What do we need to know?

Location of page table in memory (ptbr)

Size of each page table entry (assume 8 bytes – What if 4 bytes?)

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 [today's focus]
  - Doubles 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=PTBR + VPN)

(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 will we avoid in today's lecture?

BTW Don't always have to read PTE from memory!

Example: Array Iterator

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

Assume 'a' starts at 0x3000

Ignore instruction fetches

0x1000	0	
0x1004	1	
0x1008	2	
0x100c	3	0x7000

What virtual addresses?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

...

What physical addresses?

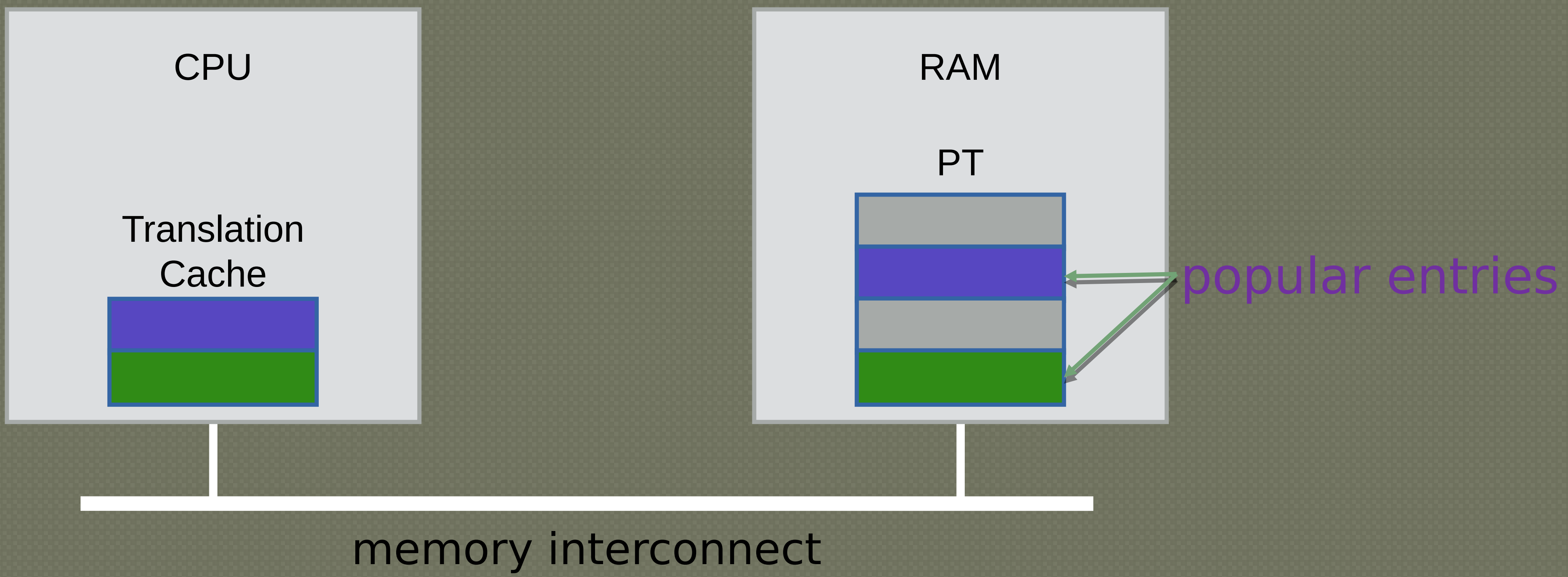
load 0x100C
load 0x7000 +0
load 0x100C
load 0x7000 + 4
load 0x100C
load 0x7000 + 8
load 0x100C
load 0x7000+ 12

Aside: What can you infer?

- ptbr: 0x1000; PTE 4 bytes each
- VPN 3 -- PPN 7
- Have 12 bits of offset

Observation: Repeatedly access same PTE because program repeatedly accesses same virtual page

Strategy: Cache Page Translations



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

Array Iterator (w/ TLB)

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

Assume following virtual address stream:
load 0x1000

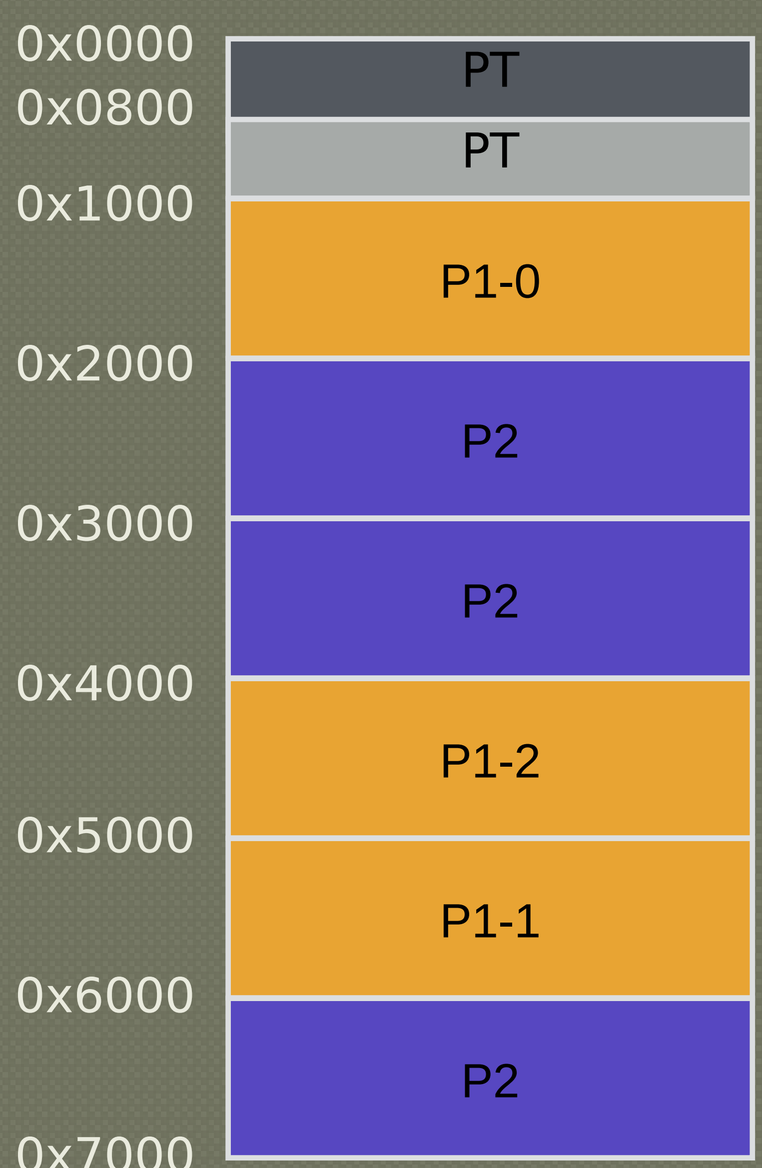
load 0x1004

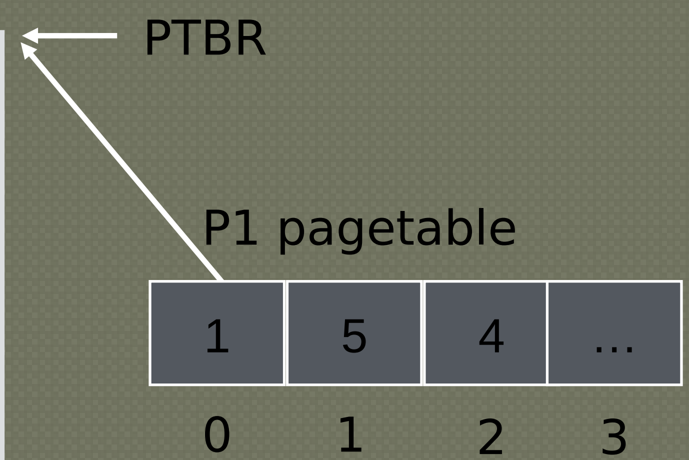
load 0x1008

load 0x100C
...

What will TLB behavior look like?

TLB Accesses: SEQUENTIAL Example





CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virt	Phys
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

Size of each page table entry = 4 bytes

PERFORMANCE OF TLB?

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

An integer is 4 bytes.
a[] is an array, its allocated contiguously in memory. It takes up
2048*4 = 8192 bytes.
Which takes a min of two and a max of 3 4K pages (assume 2)

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 still have 2048 lookups but would have to access more pages

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: The amount of memory accessible from the TLB.

TLB Reach = (TLB Size) X (Page Size)

TLB PERFORMANCE with Workloads

Sequential array accesses almost always hit in TLB

- Very fast!

What access pattern will be slow?

- Highly random to many different pages, with no repeat accesses

Workload
ACCESS PATTERNS

Workload A

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

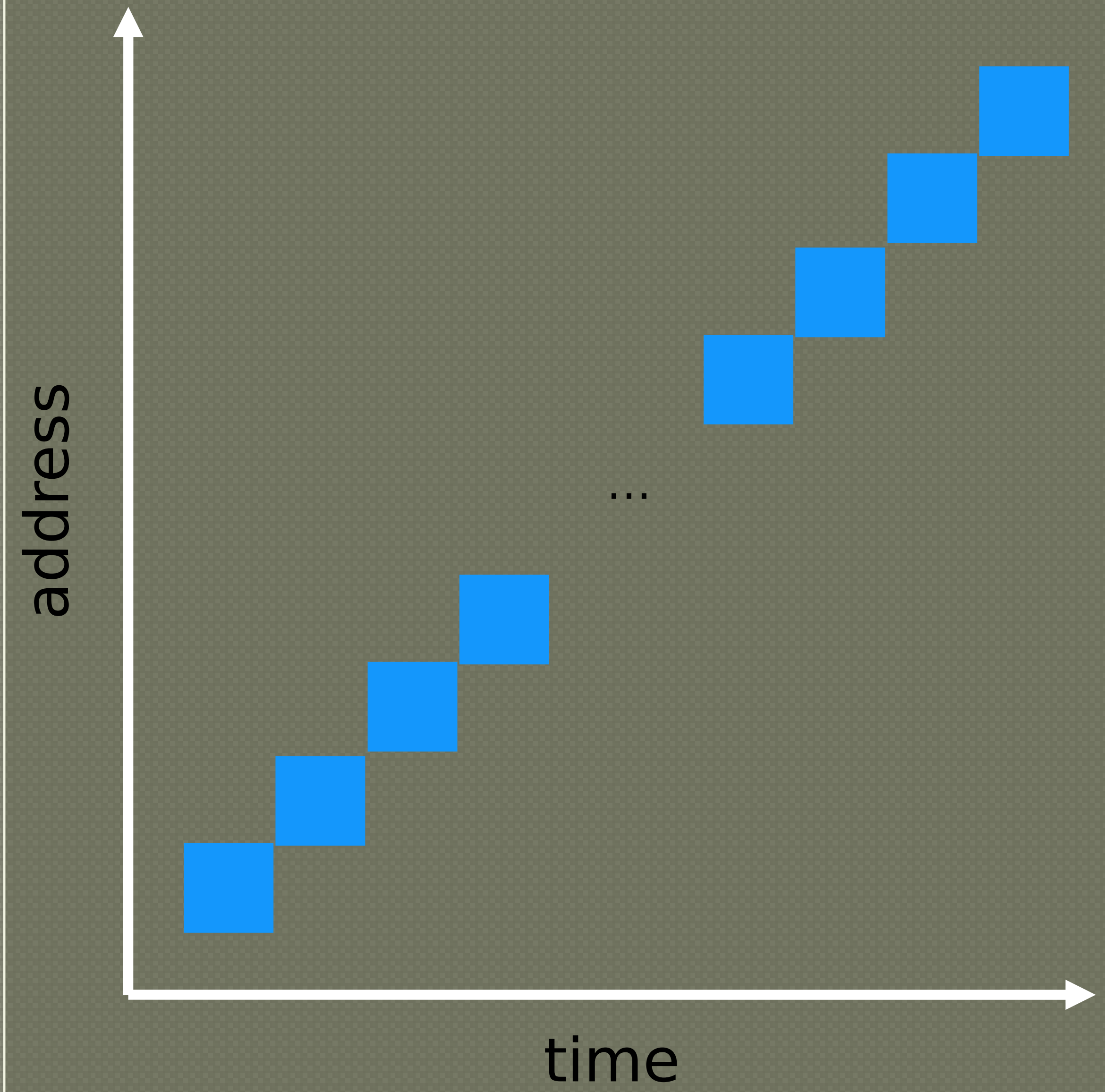
Workload B

```
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

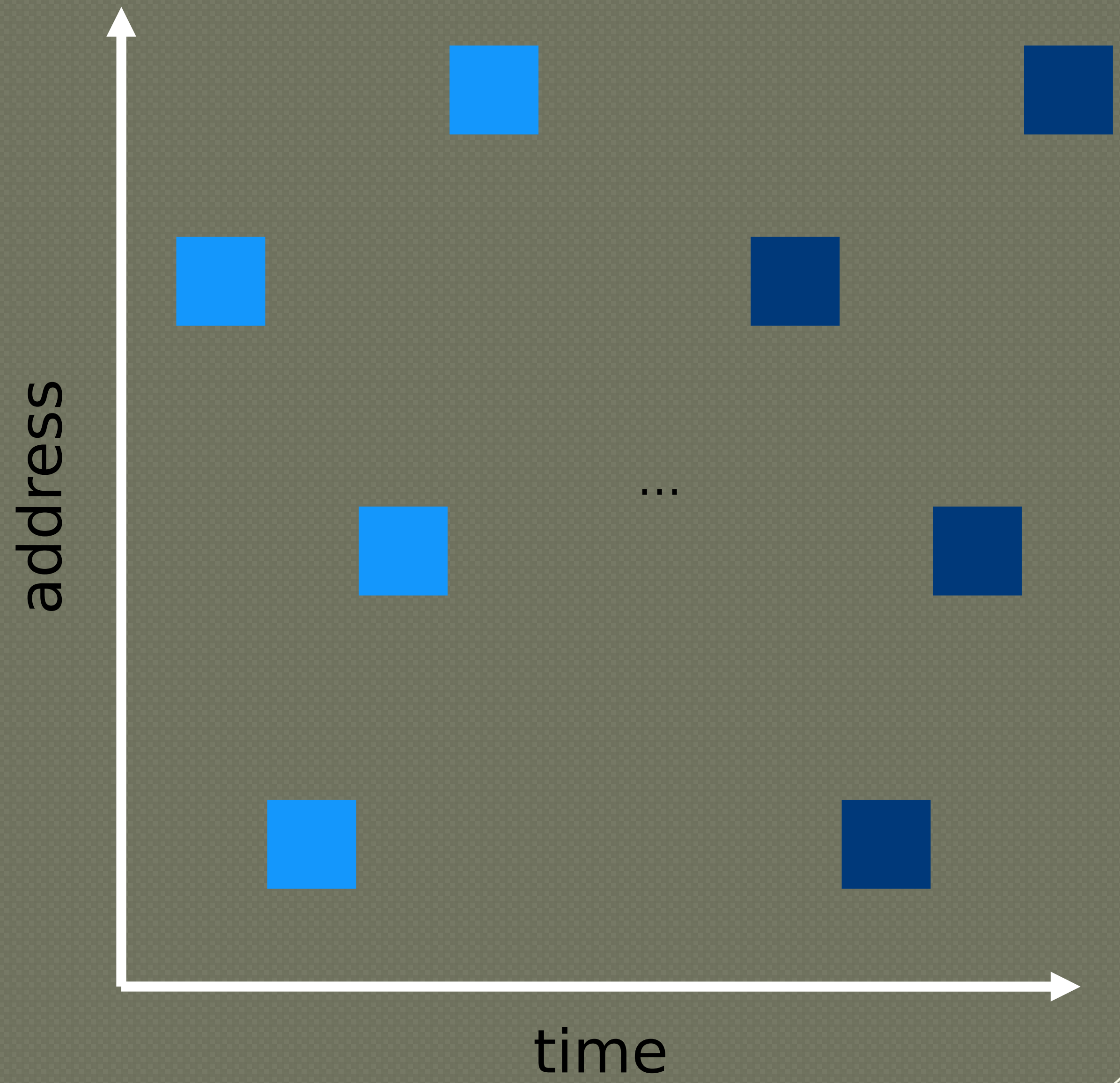
Spatial Locality

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->ppn translation

Same TLB entry re-used

Temporal:

Access same address near in future

Same TLB entry re-used in near future

TLB
Replacement policies

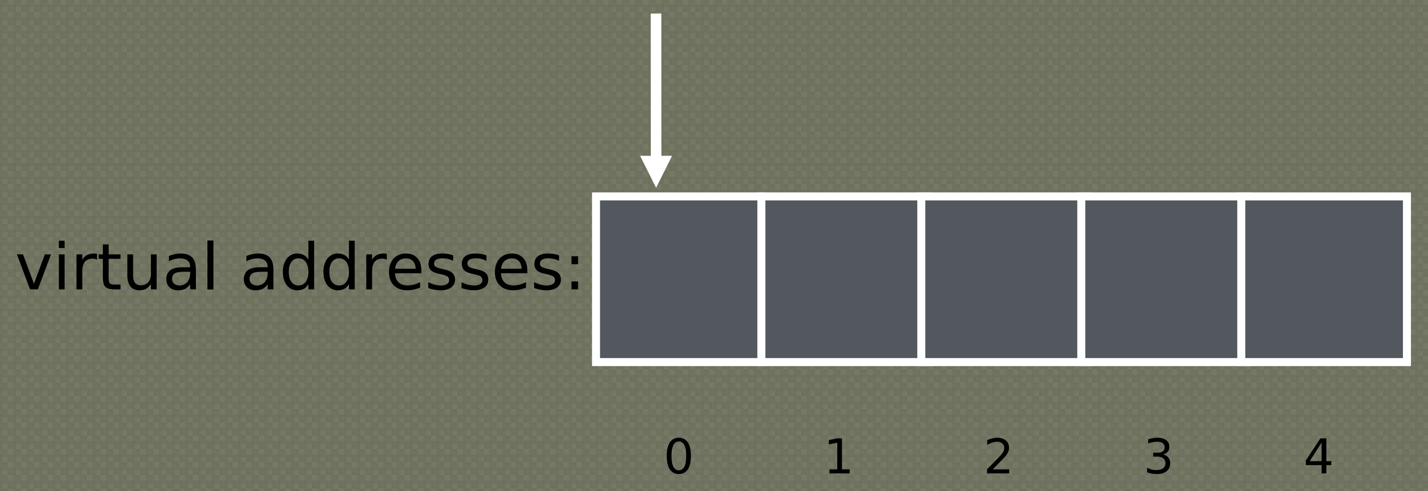
LRU: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies next week)

Random: Evict randomly choosen entry

Which is better?

LRU Troubles



Valid	Virt	Phys
0	?	?
0	?	?
0	?	?
0	?	?

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

What will TLB contents be over time?
How will TLB perform?

LRU Troubles

virtual addresses:	3	4	5	6	7
	0	1	2	3	4

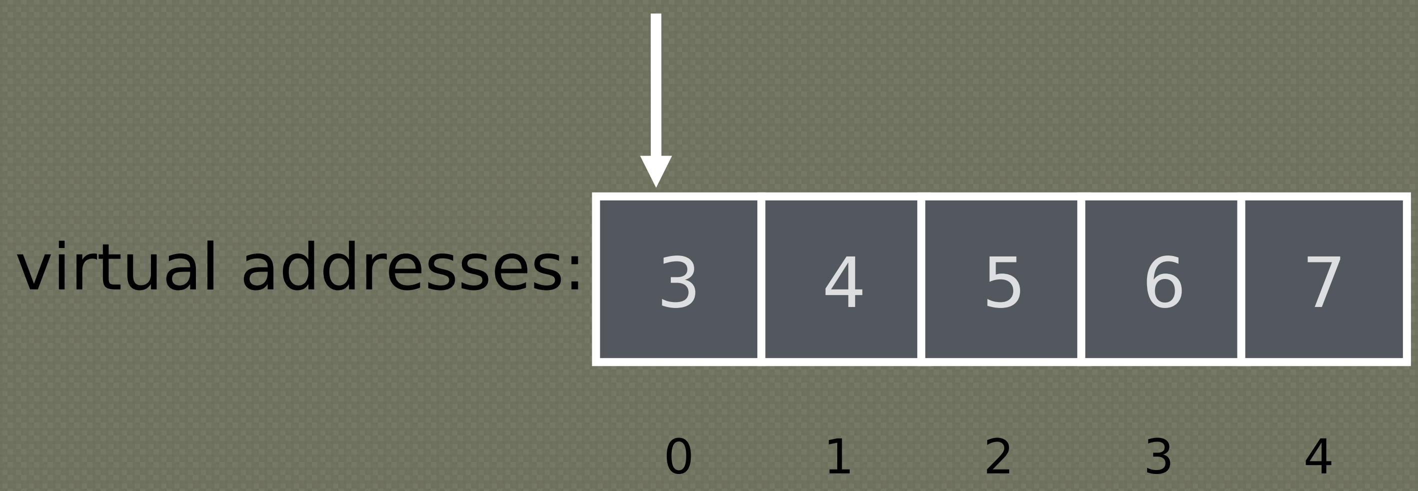
Valid	Virt	Phys
0	?	?
0	?	?
0	?	?
0	?	?

For this workload. What is the hit rate?

0x0000
0x1000
0x2000
0x3000
0x4000

loops

LRU Troubles



Valid	Virt	Phys
1	0	3
1	1	4
1	2	5
1	3	6

For this workload. What is the hit rate?

0x0000
0x1000
0x2000
0x3000
0x4000

What happens here?

LRU Troubles

virtual addresses:

0	1	2	3	4
3	4	5	6	7

Valid	Virt	Phys
1	0	3
1	1	4
1	2	5
1	3	6

For this workload. What is the hit rate?

0x0000
0x1000
0x2000
0x3000
0x4000

Hit rate = #TLB hits/#TLBLookups
#TLBLookups=5
#TLBHits=0
Hitrate=0/5

Would be better to use Random replacement policy

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?

Solutions?

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 unique ASIDs?

    - why not use PIDs?

Context Switches

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

Solutions?

1. Flush TLB on each switch

   Costly; lose all recently cached translations

2. Track which entries are for which process

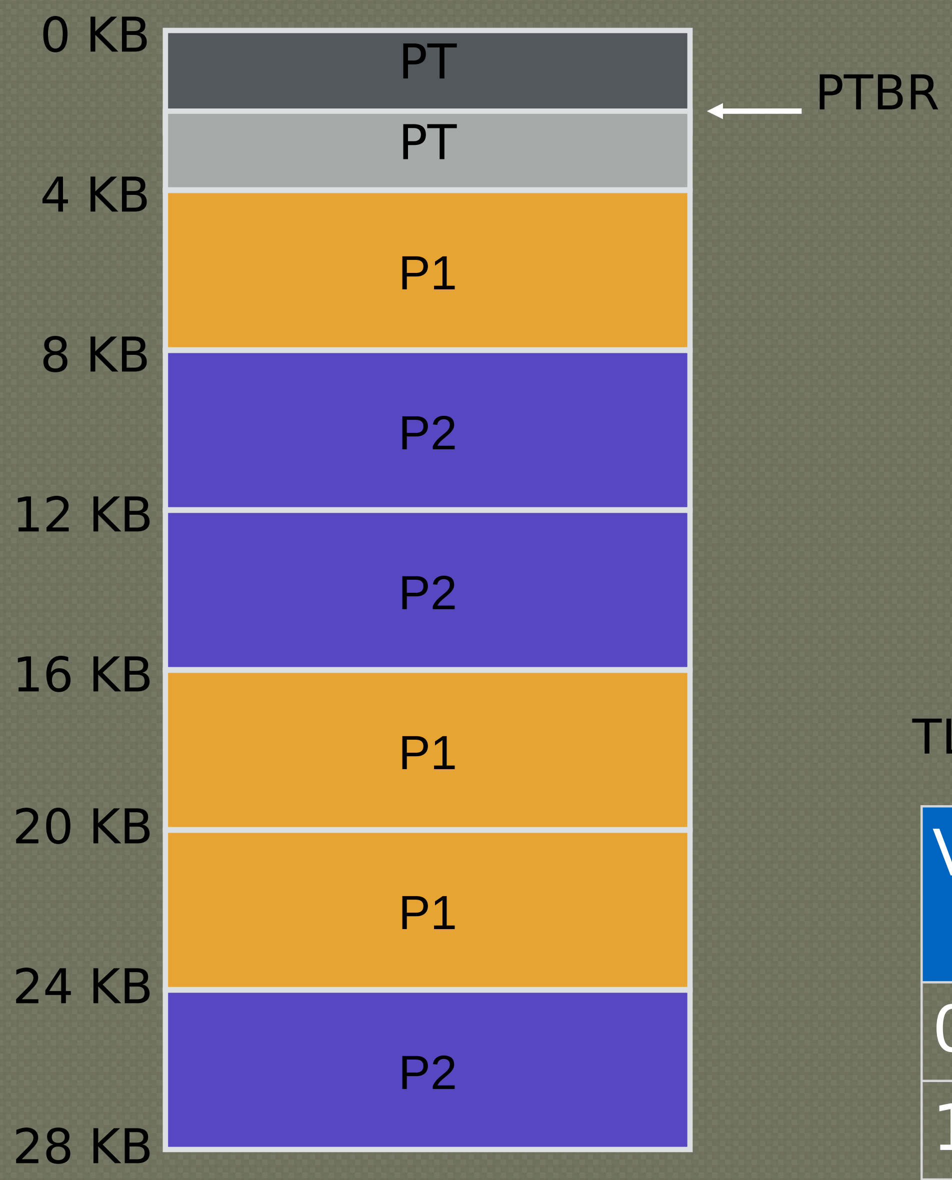
   Address Space Identifier

   Tag each TLB entry with an 8-bit ASID

   - how many unique ASIDs? 2**8 = 256

   - why not use PIDs? PID is 32 bits, TLB is small, cannot hold 2**32 processes page table entries.

TLB Example with ASID



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

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

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

OS: CPU traps into OS upon TLB miss

- “Software-managed TLB”
- OS interprets pagetables as it chooses
- Modifying TLB entries is privileged
   - otherwise what could process do?

Need same protection bits in TLB as pagetable
- rwx

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
  - Increase **TLB reach** by increasing page size
- 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