[537] TLBs

Tyler Harter
9/21/14

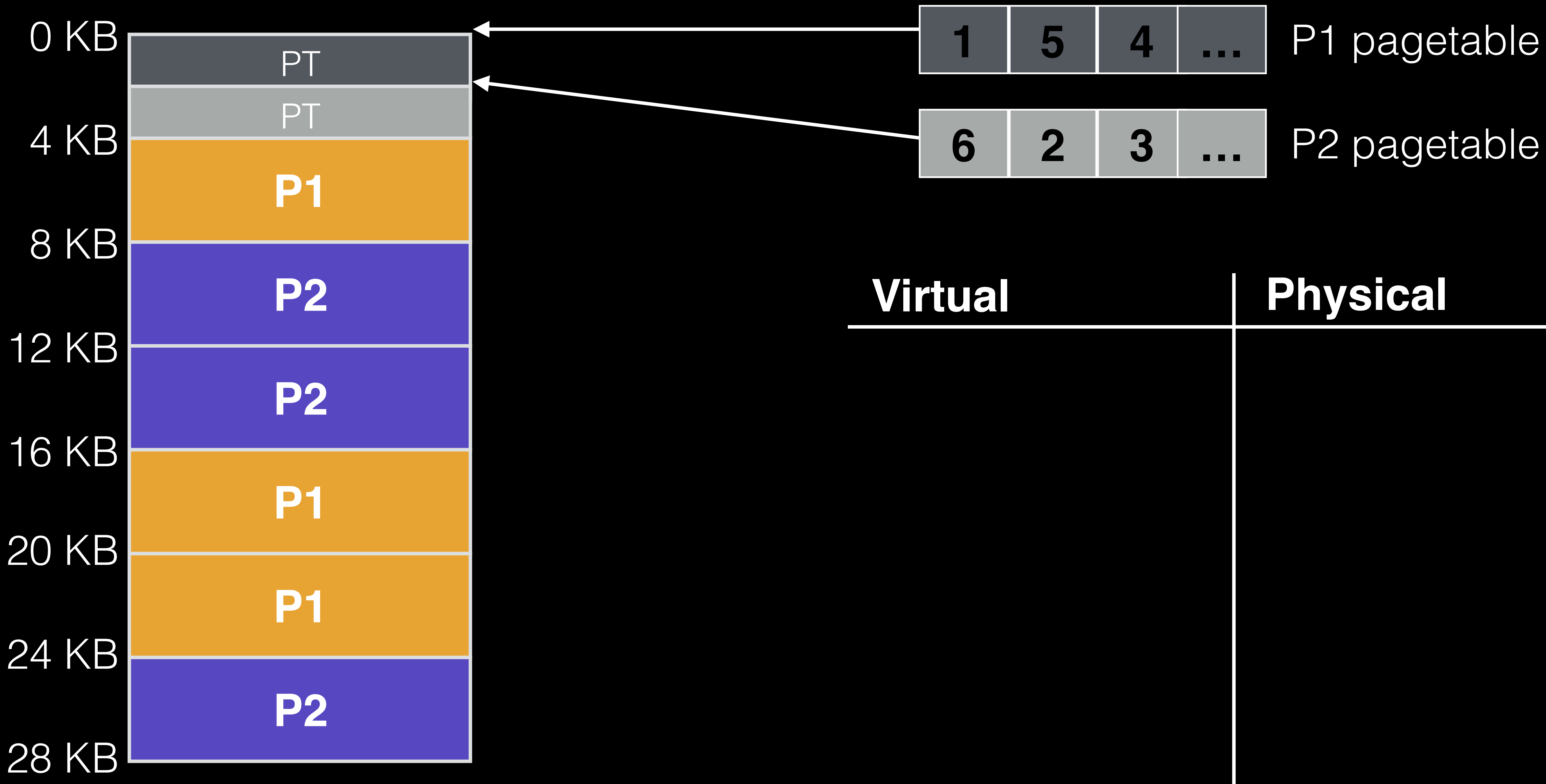
Overview

Review Paging

TLBs (Chapter 18)

TLB measurement demo (if time)

Review: Paging



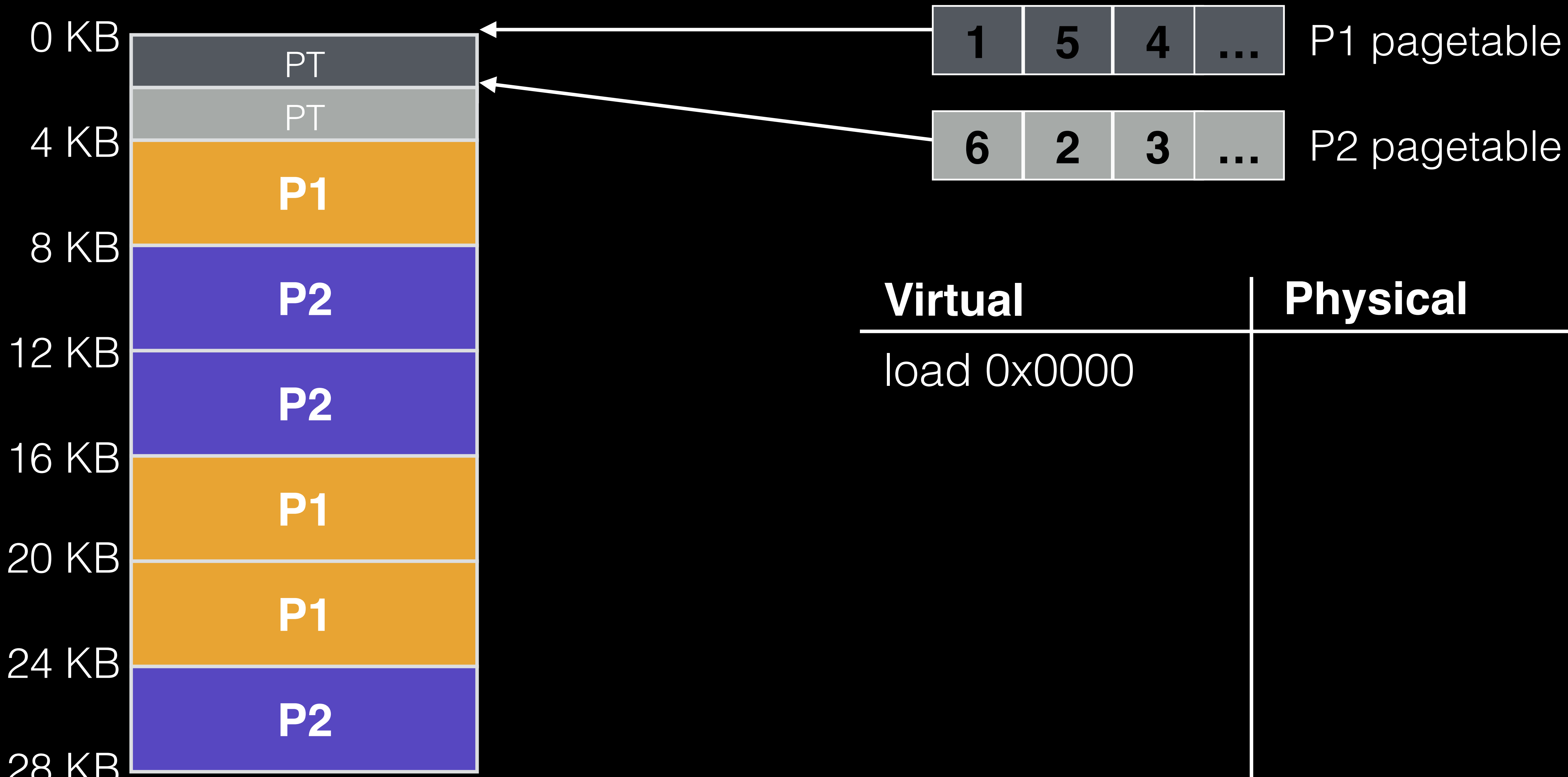
0 KB
4 KB
8 KB
12 KB
16 KB
20 KB
24 KB
28 KB

PT
PT
P1
P2
P2
P1
P1
P2

1 5 4 ... P1 pagetable

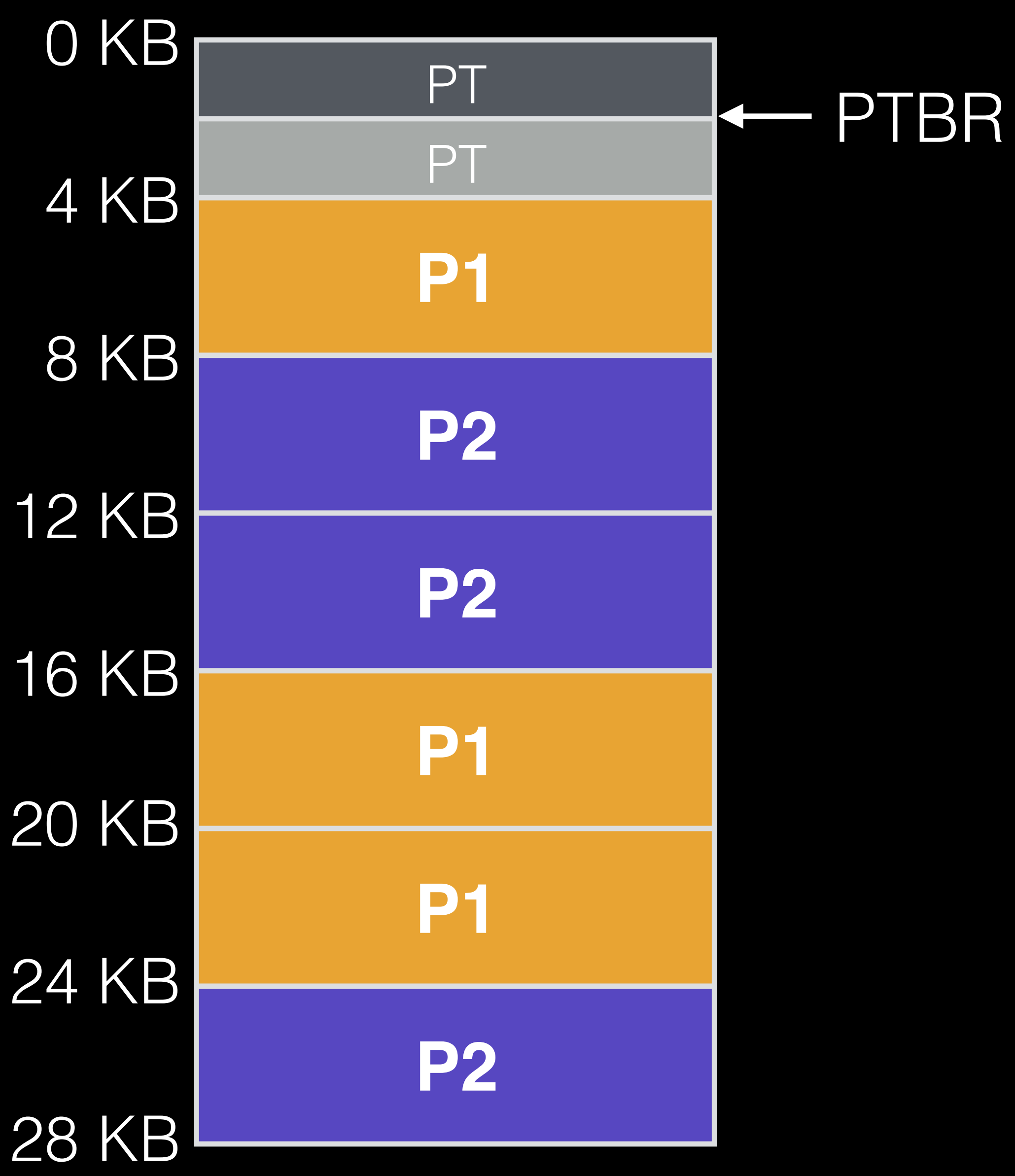
6 2 3 ... P2 pagetable

Virtual	Physical



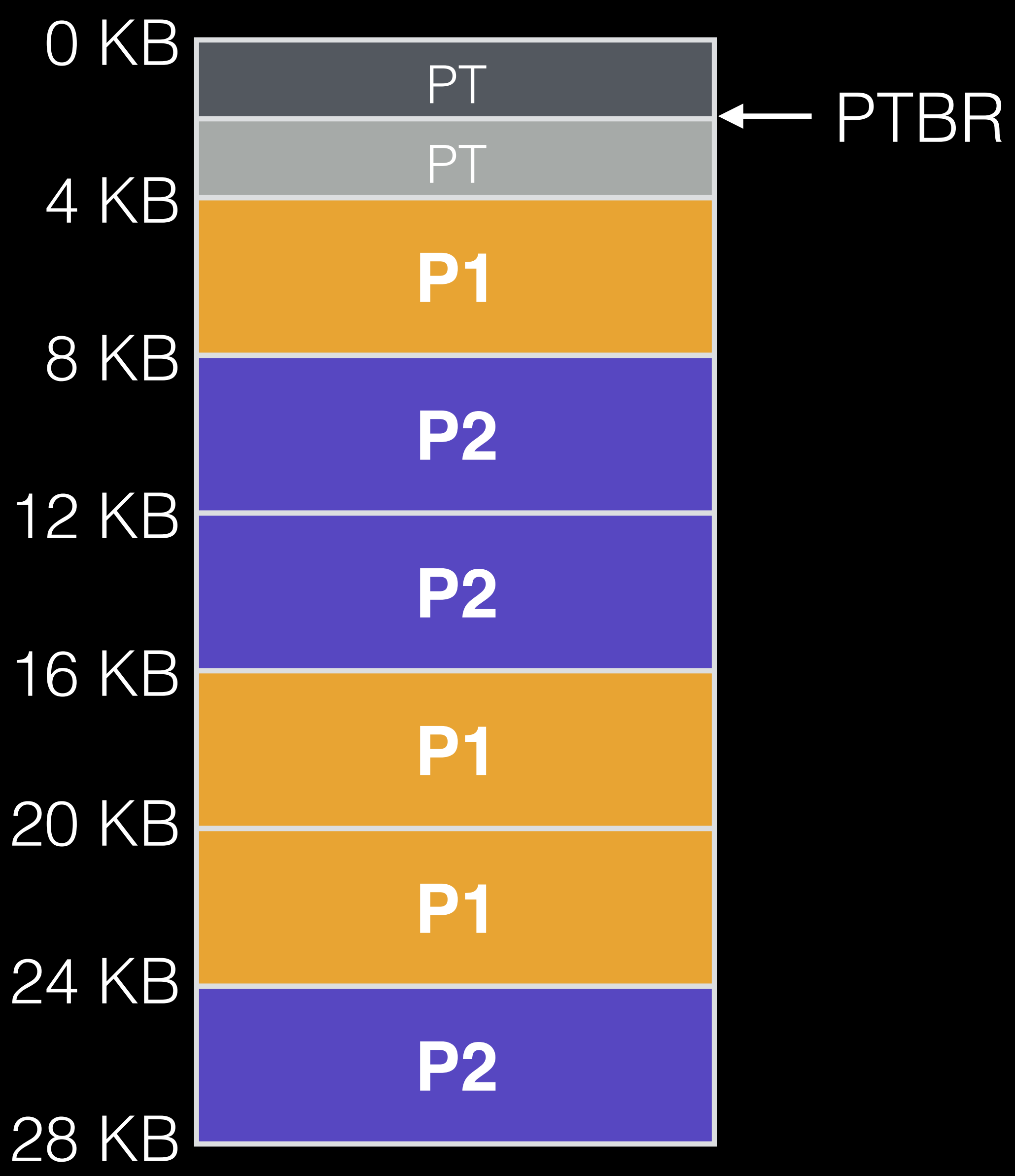
Virtual	Physical
load 0x0000	

what must you know?



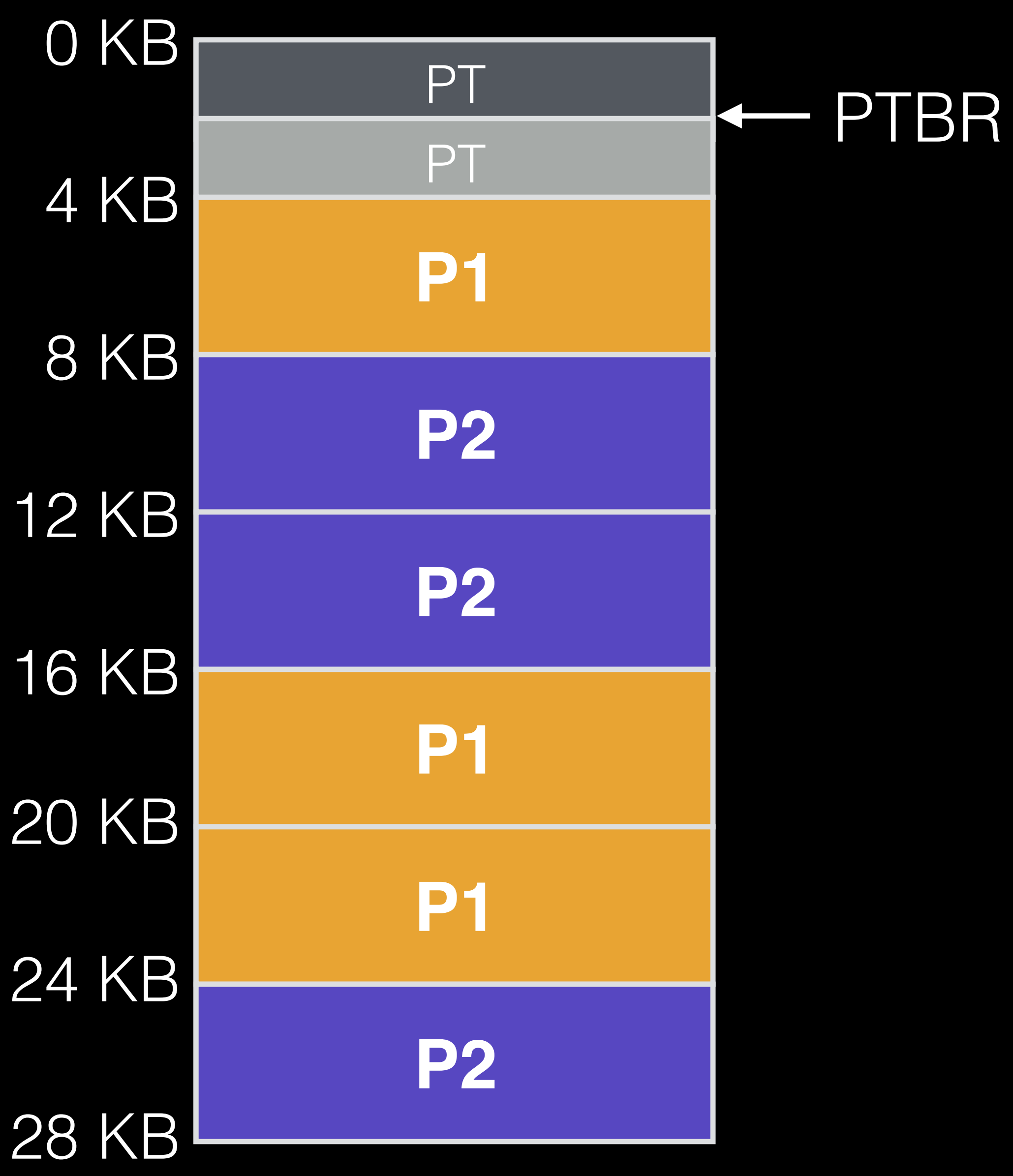
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	



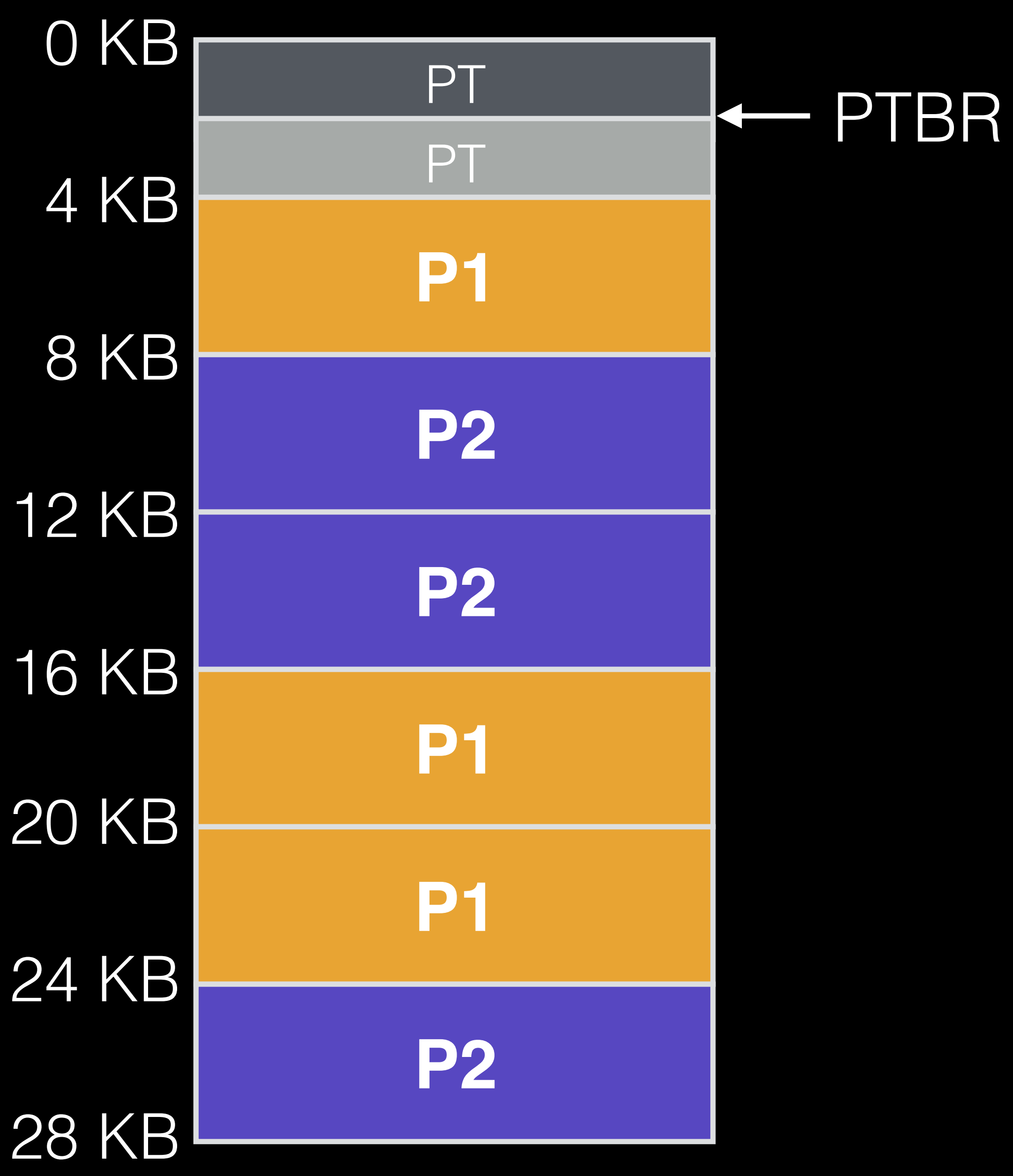
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	**Physical**
load 0x0000	load 0x0800 (2KB)



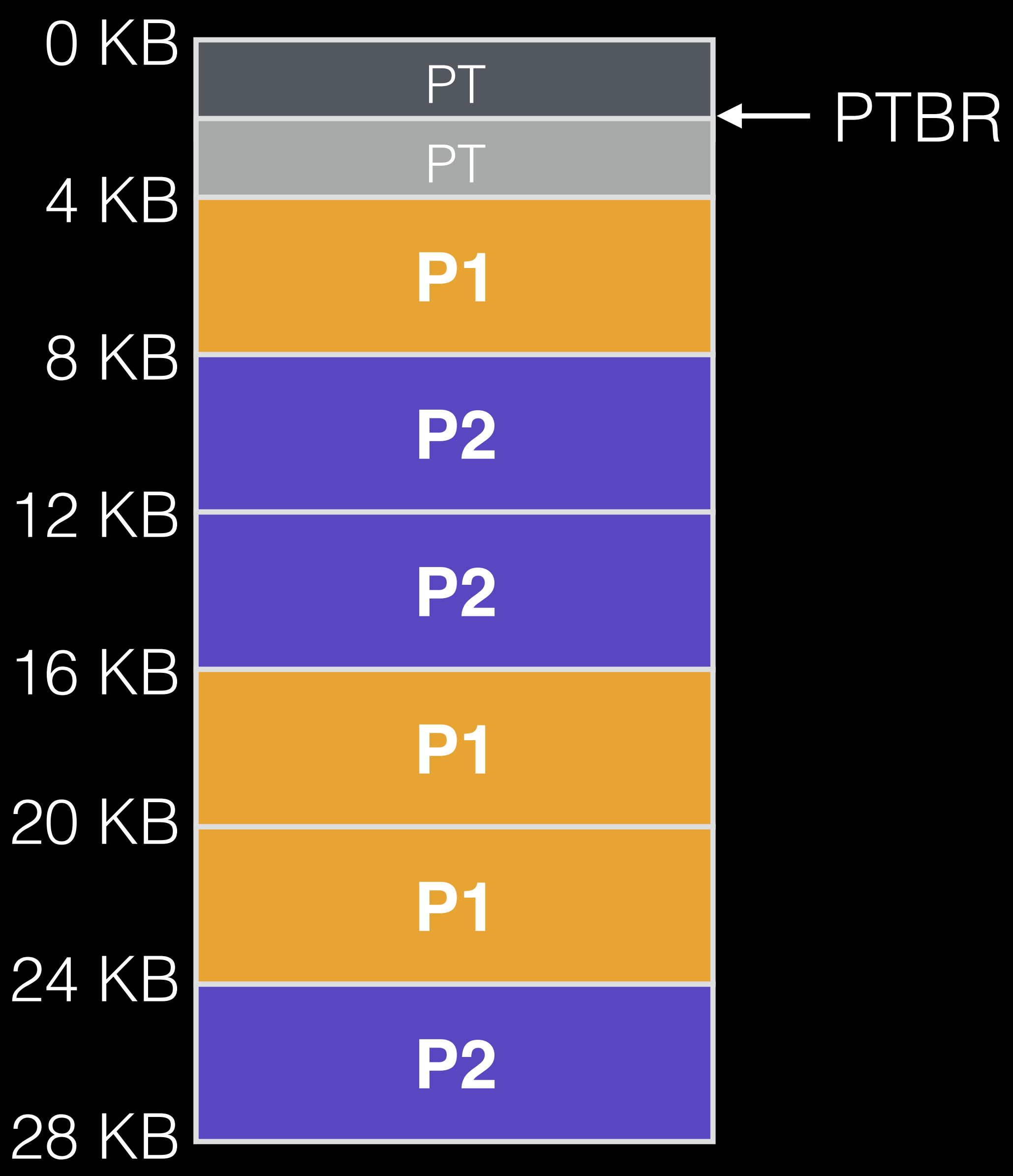
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)



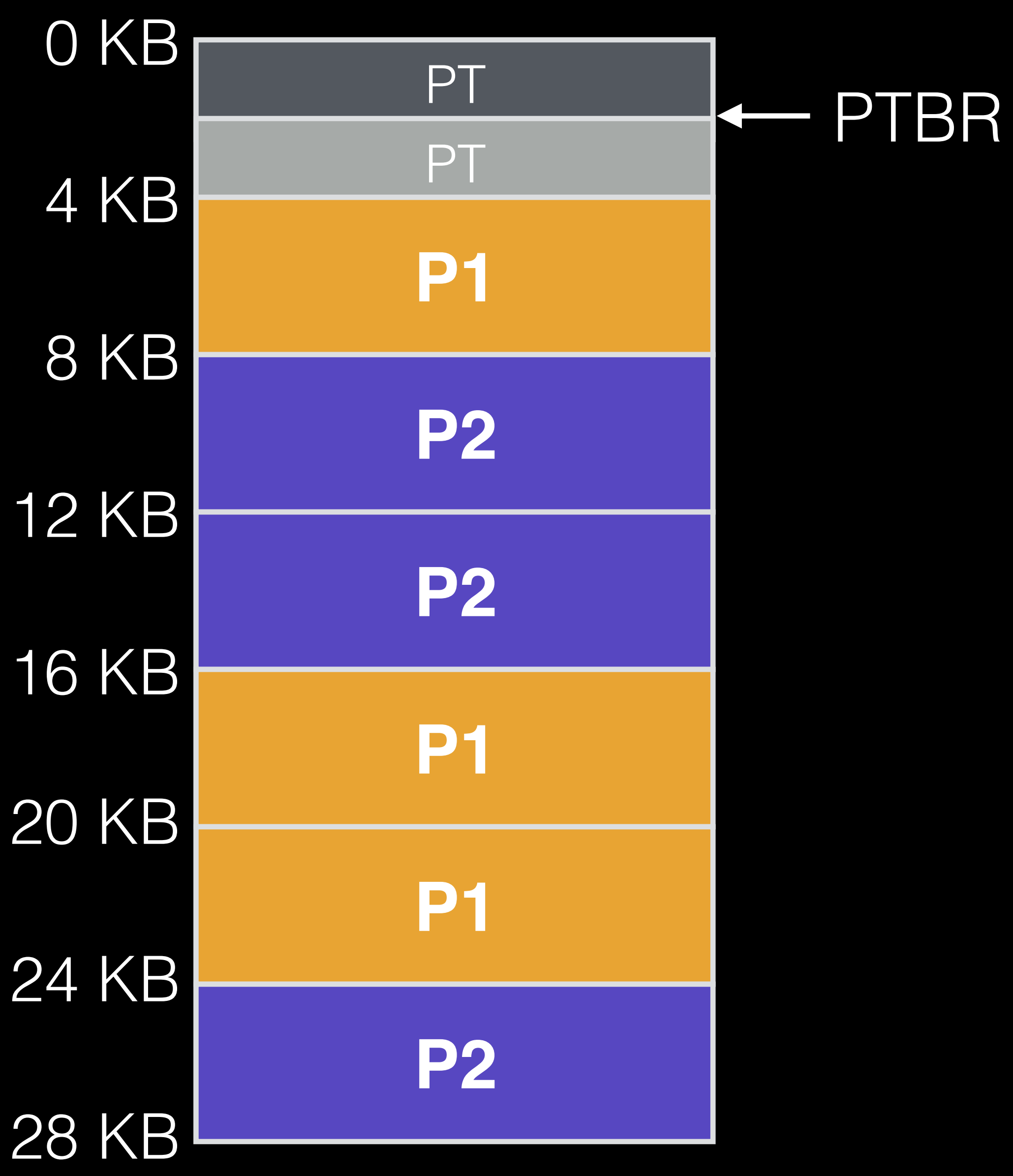
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable
0				

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)



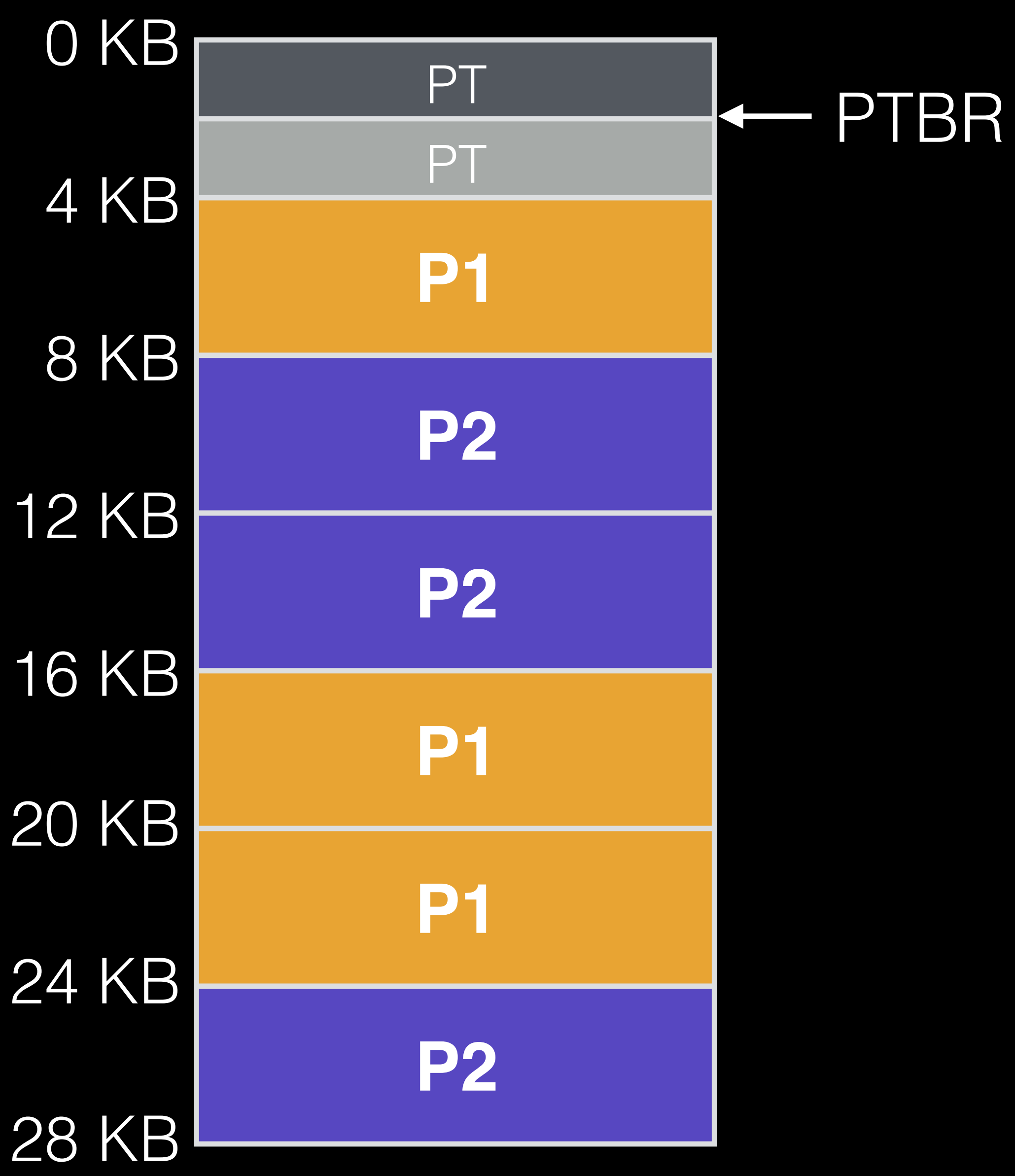
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)



1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

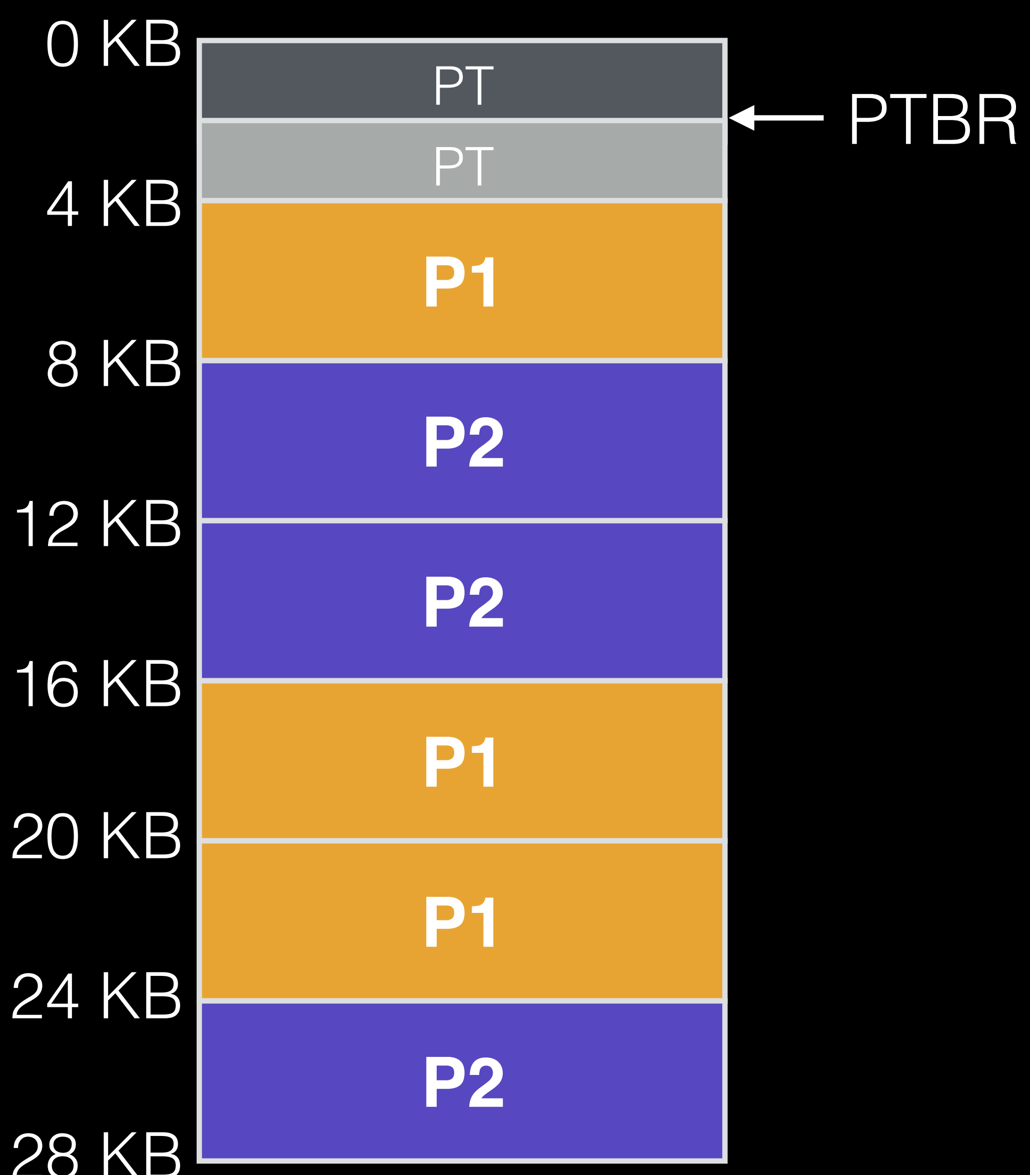
Virtual	**Physical**
load 0x0000	load 0x0800 (2KB)
	load 0x6000 (24KB)



1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	**Physical**
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	

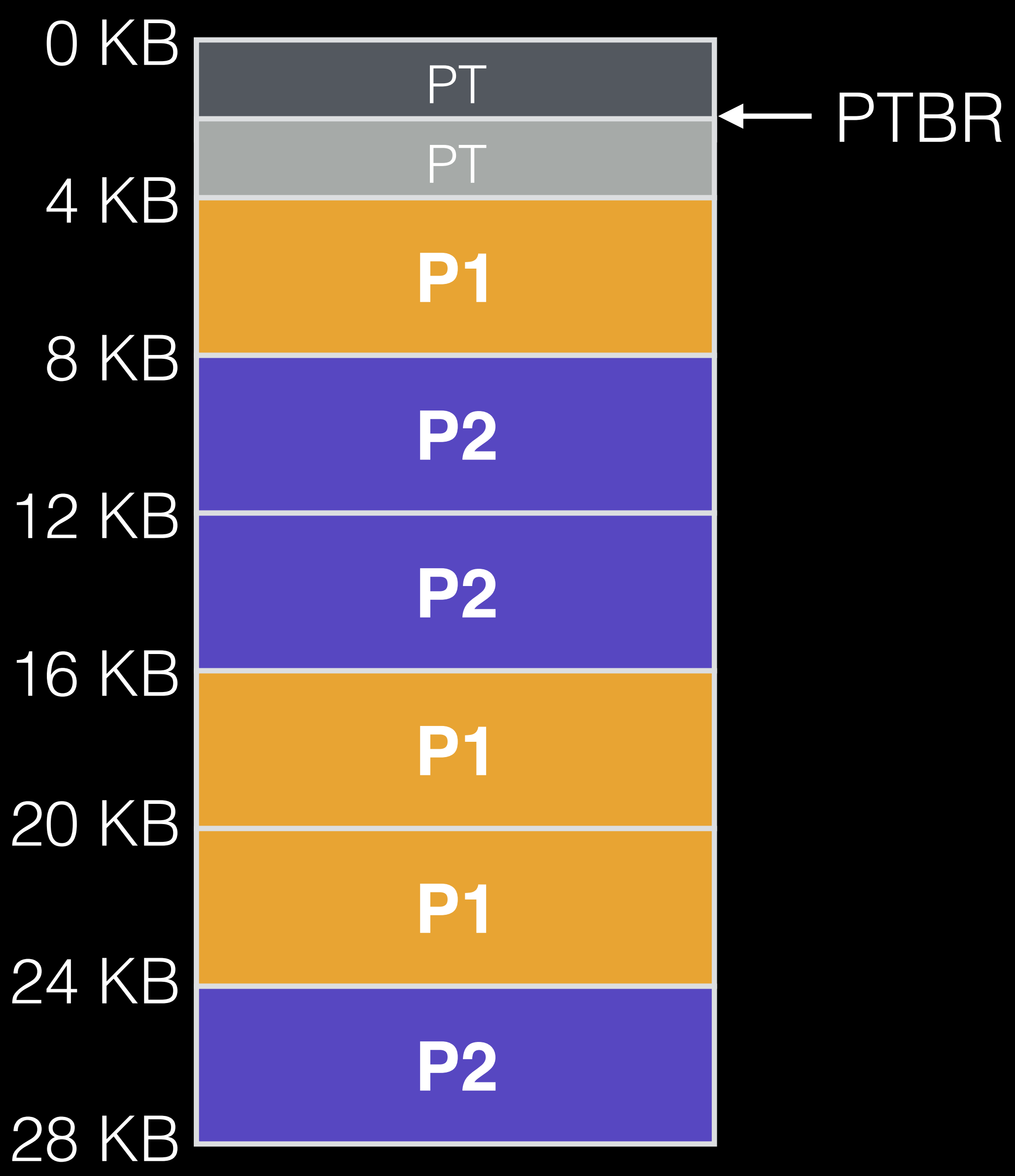
what must you know?



1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

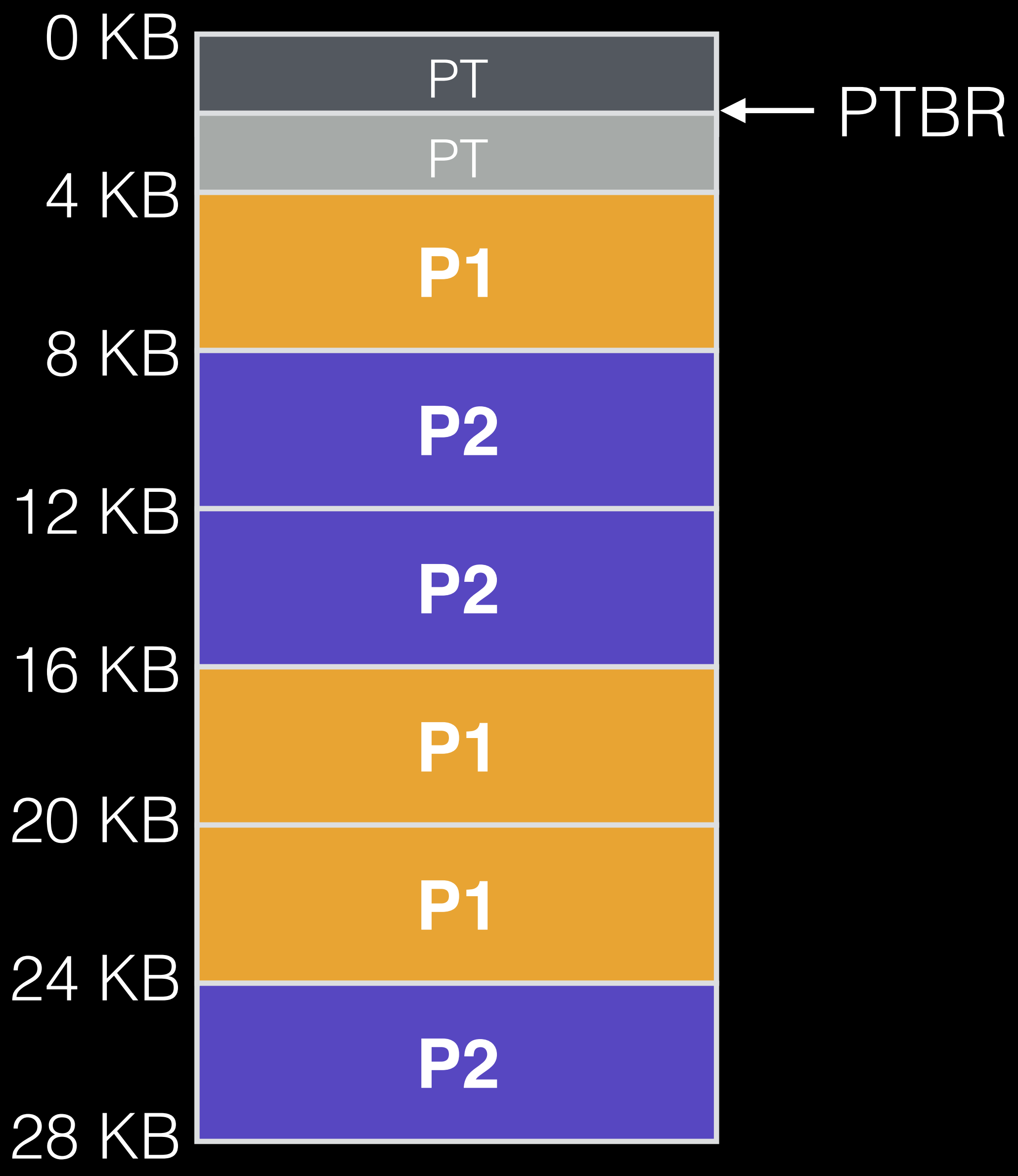
Virtual	**Physical**
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	

assume 8-byte PTEs



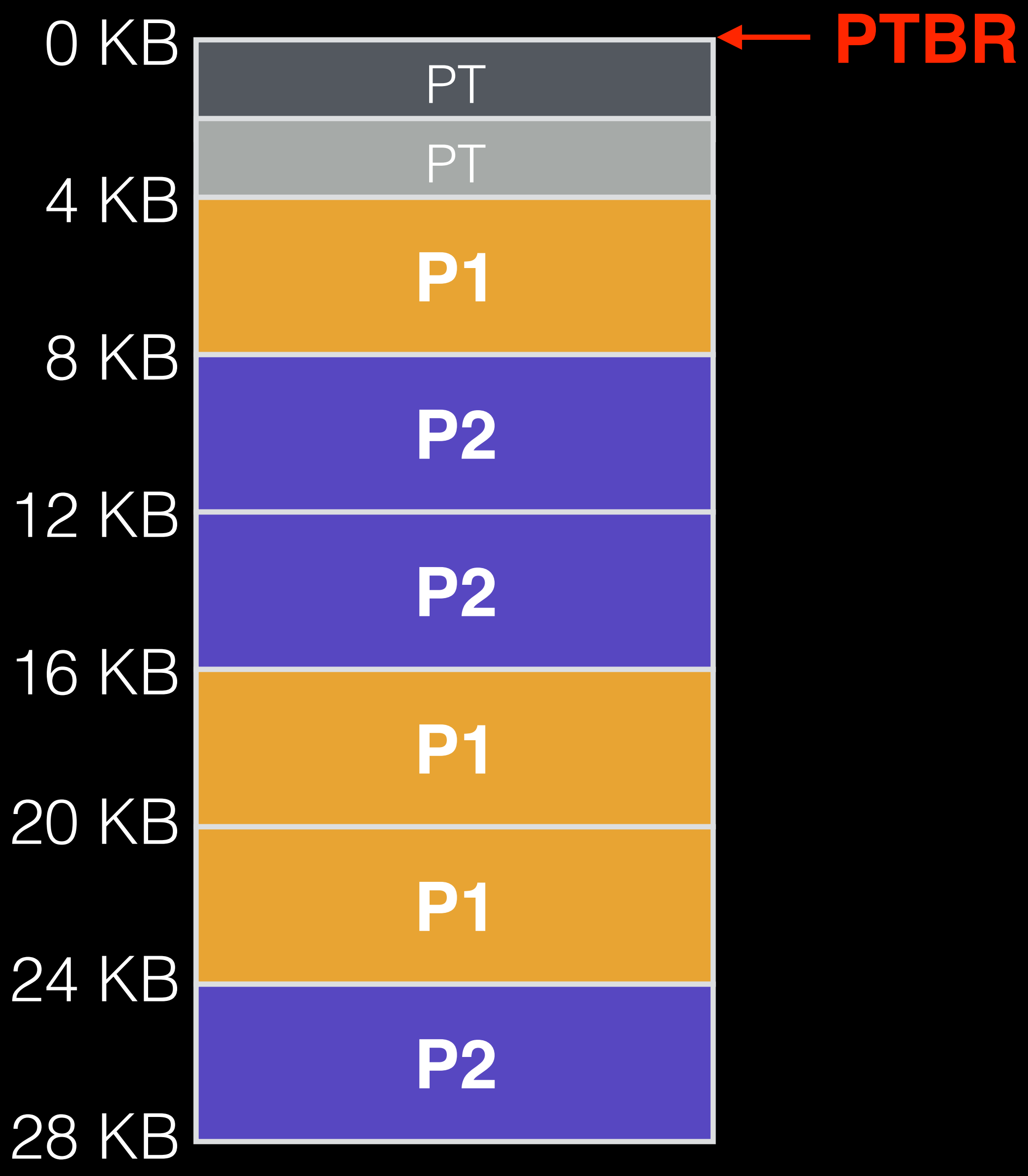
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808



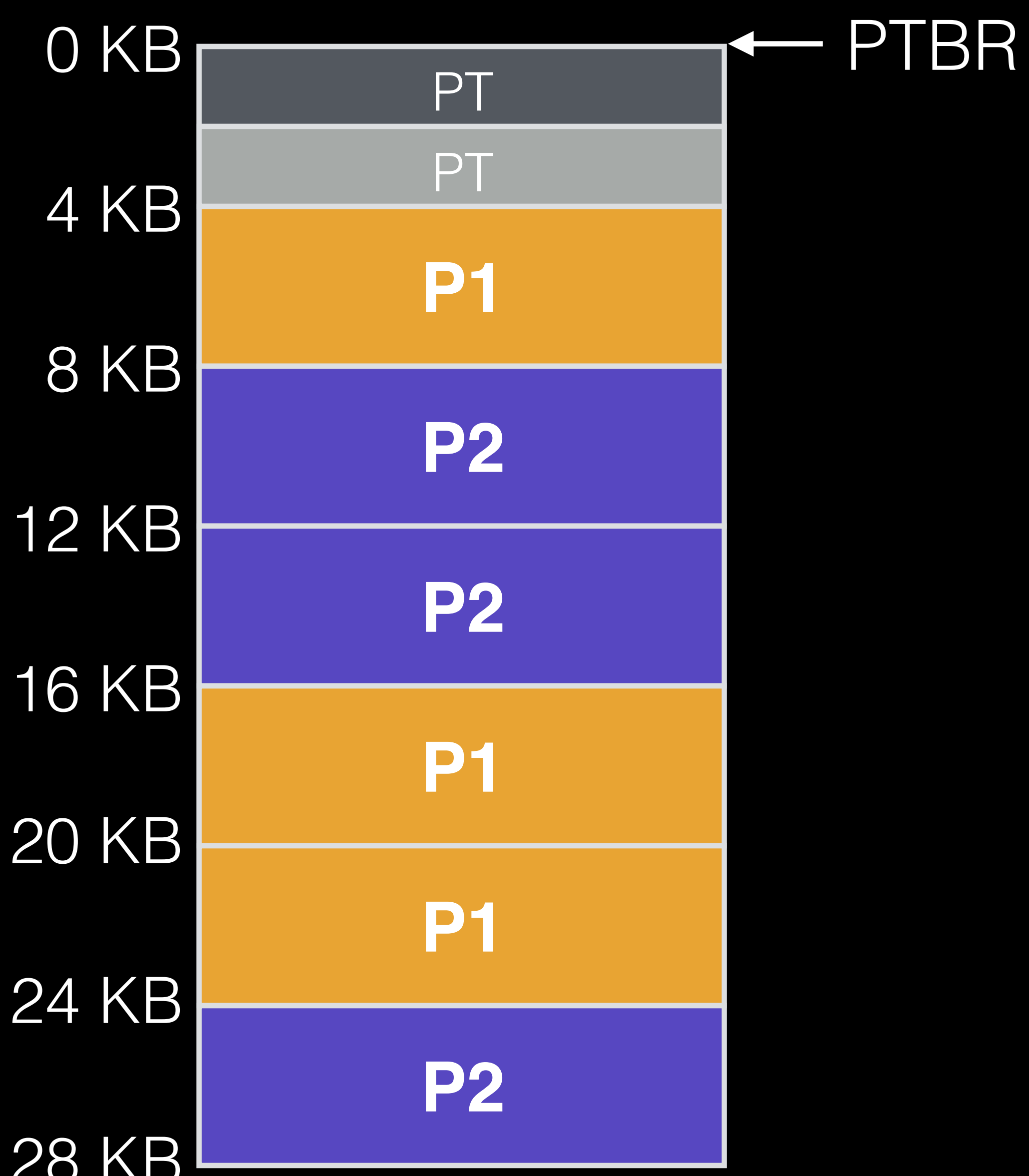
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444



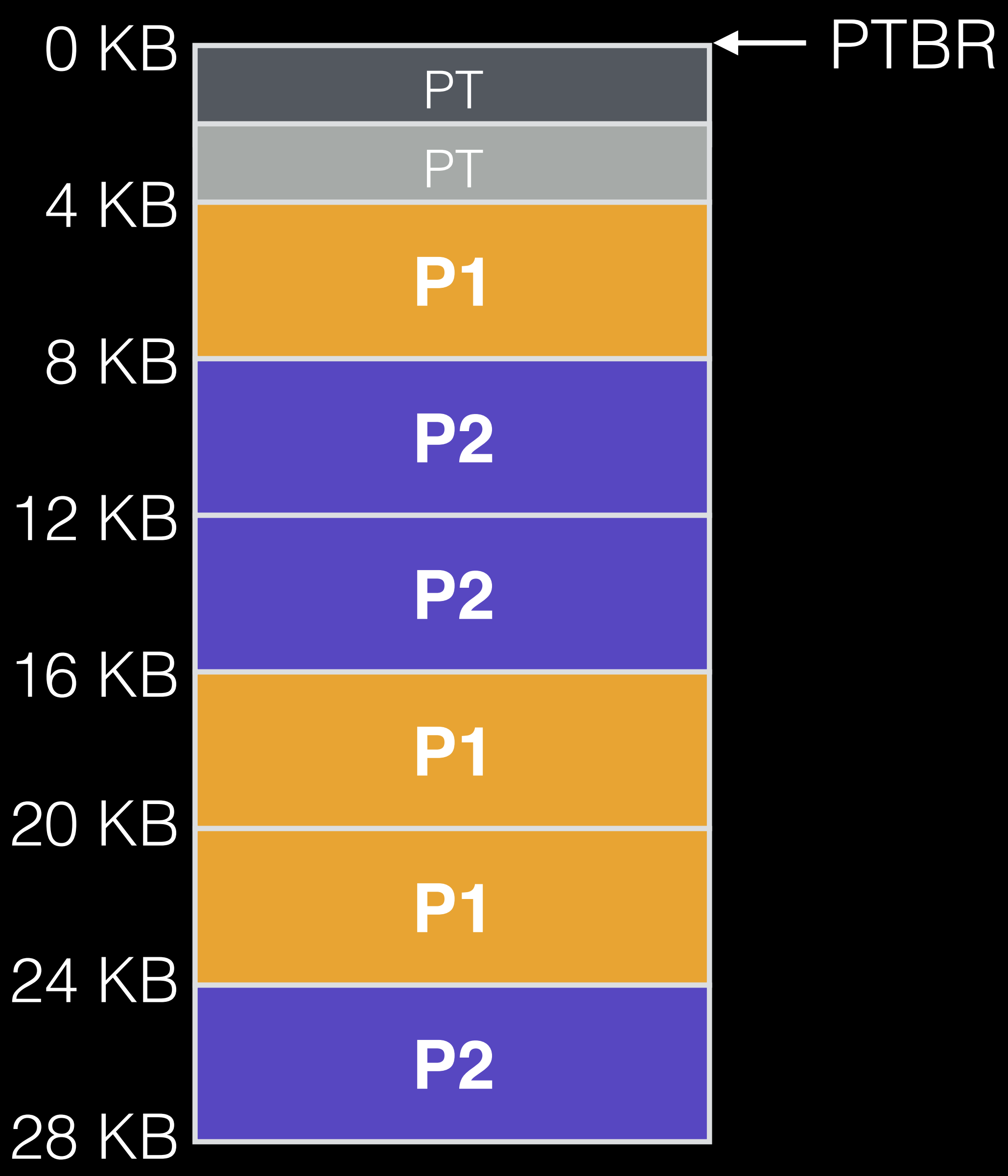
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	**Physical**
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444



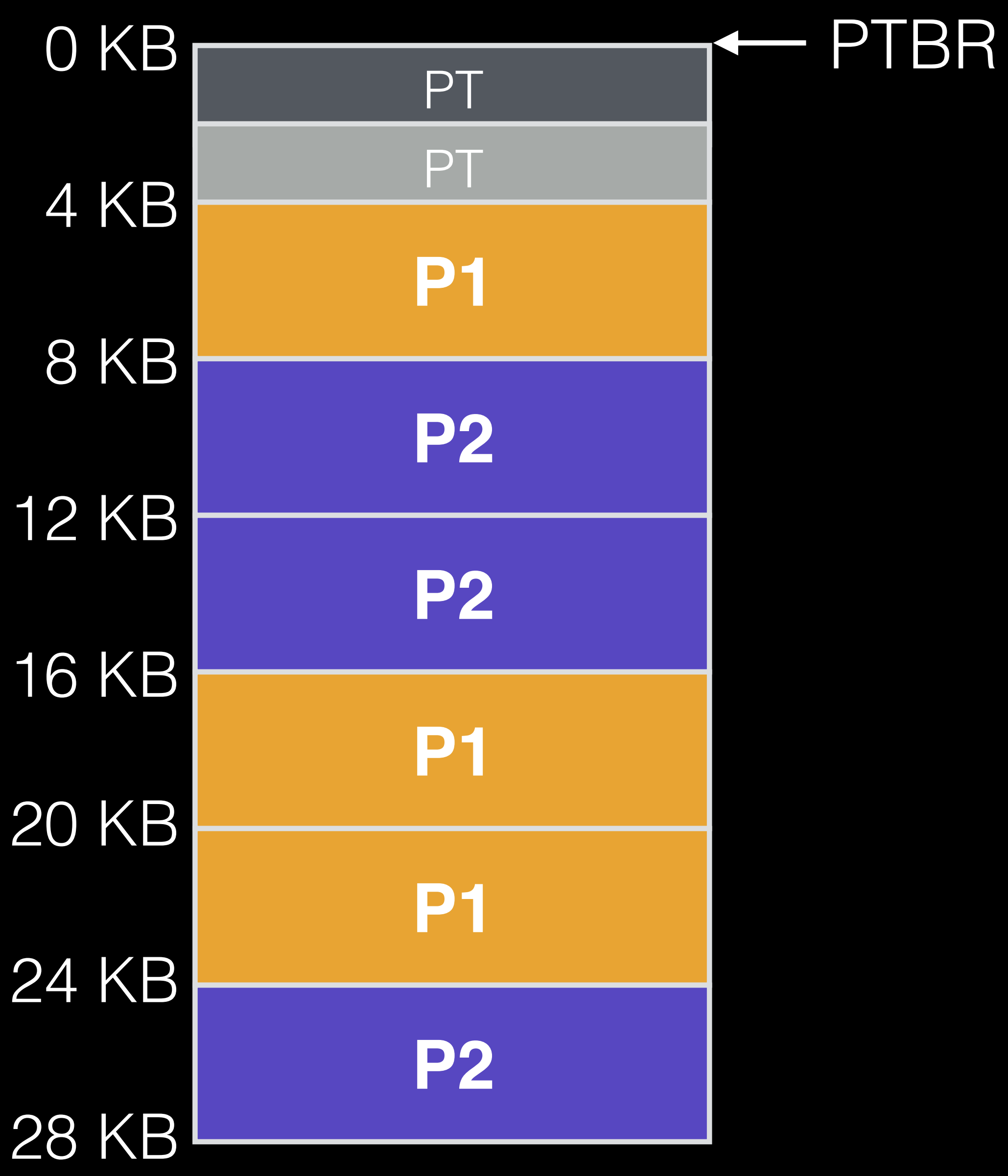
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444



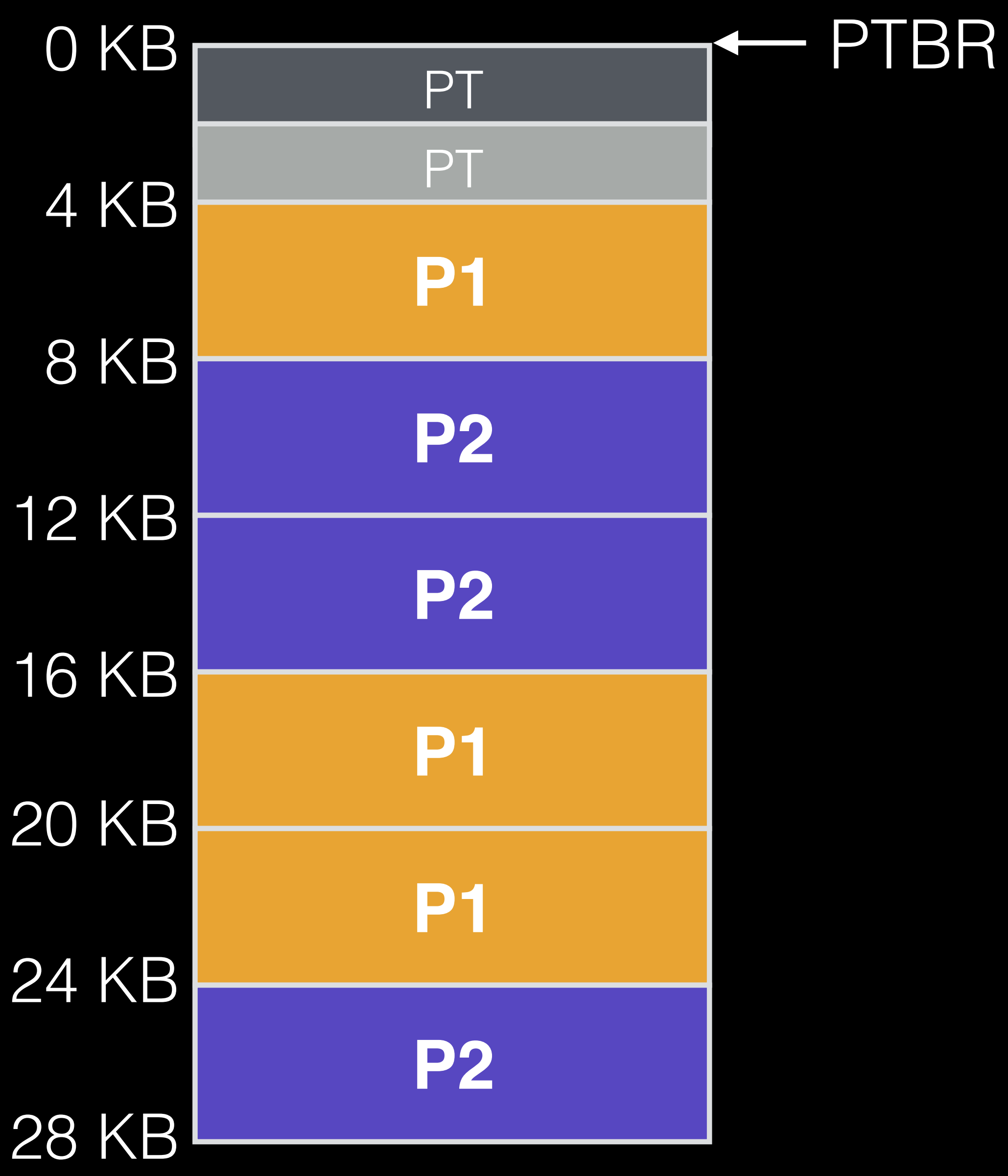
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444
load 0x1444	



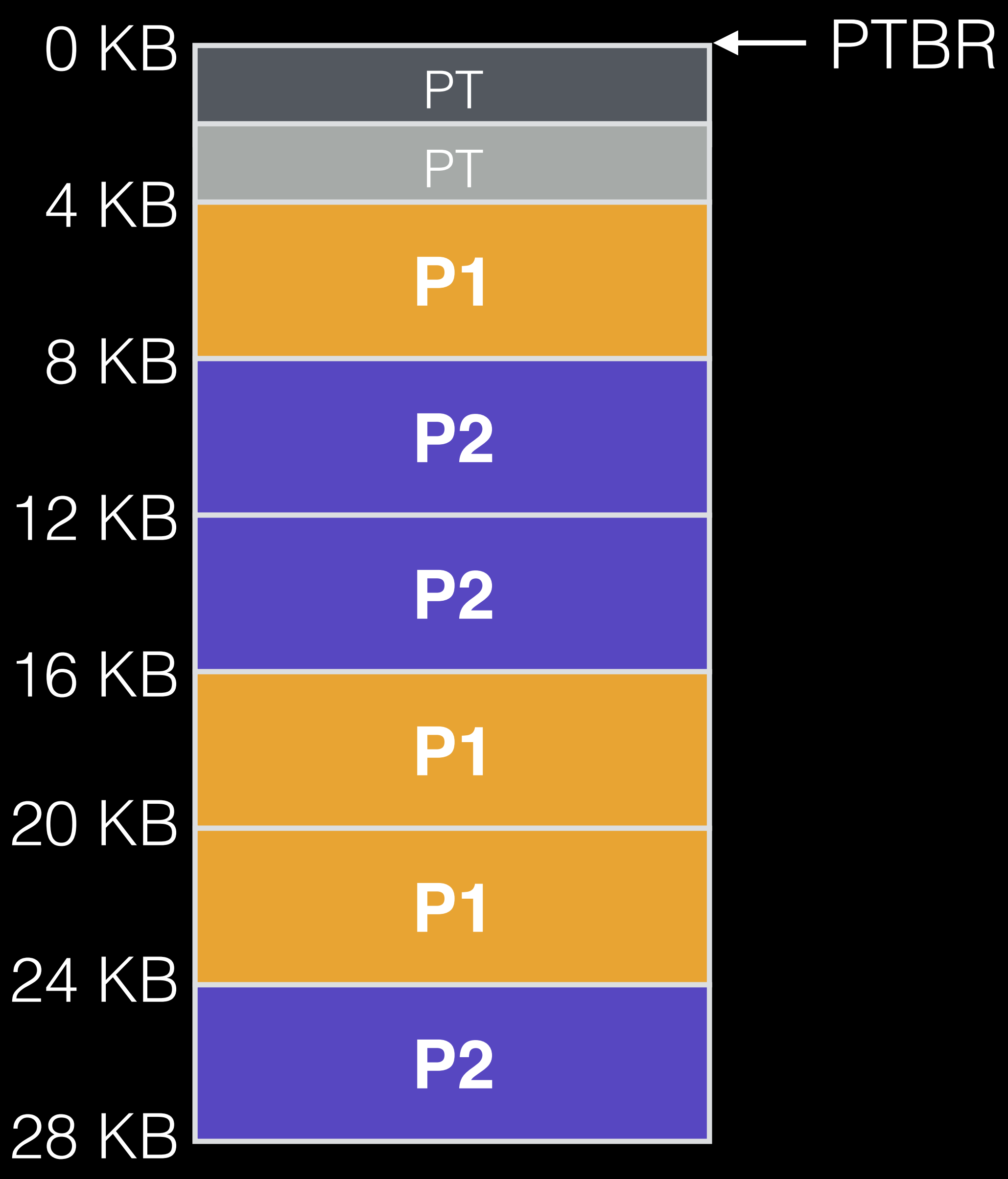
1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	**Physical**
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444
load 0x1444	load 0x0008



1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

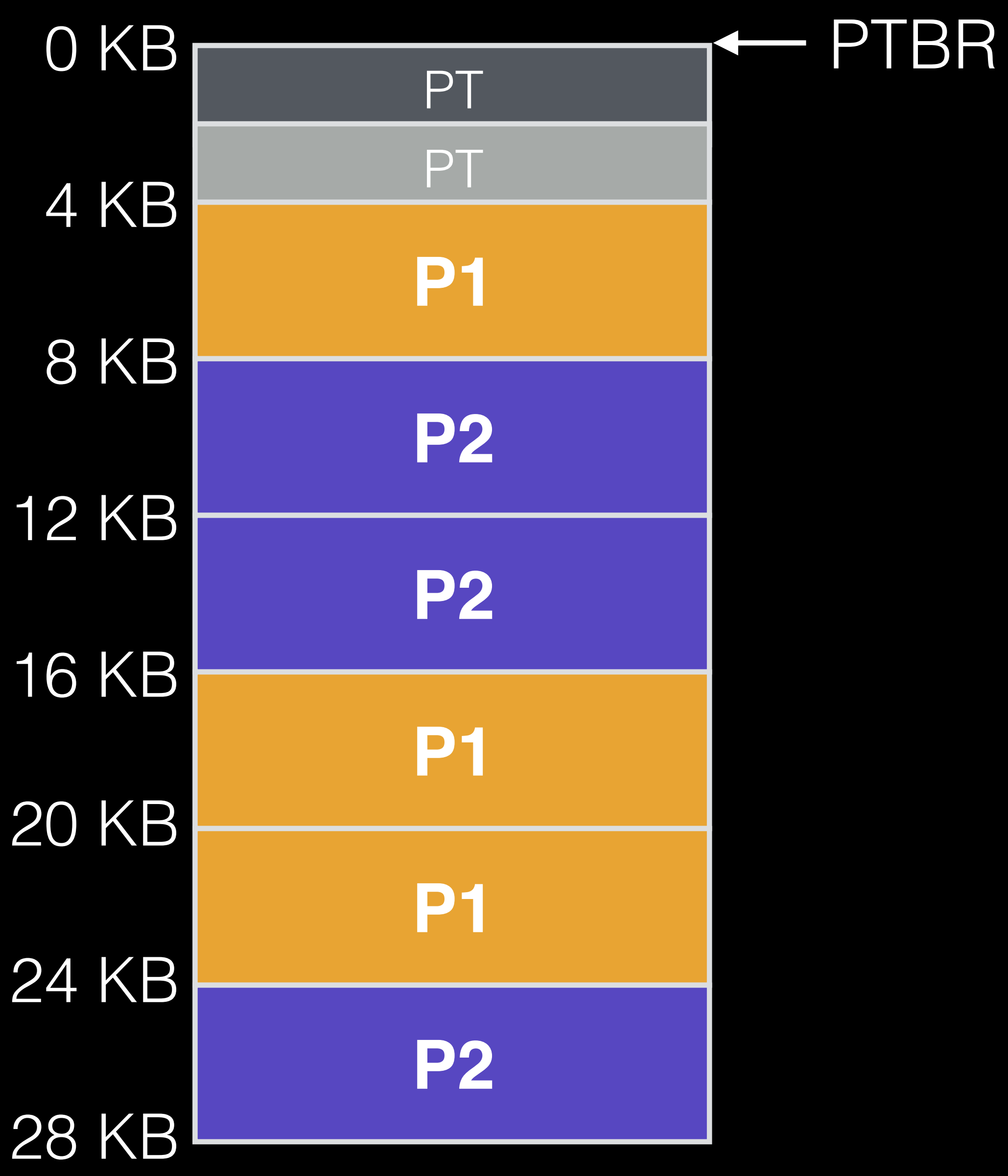
Virtual	**Physical**
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444
load 0x1444	load 0x0008 load 0x5444



1

1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	Physical
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444
load 0x1444	load 0x0008 load 0x5444



1	5	4	...	P1 pagetable
6	2	3	...	P2 pagetable

Virtual	**Physical**
load 0x0000	load 0x0800 (2KB) load 0x6000 (24KB)
load 0x1444	load 0x0808 load 0x2444
load 0x1444	load 0x0008 load 0x5444

Chapter 19: TLBs

Outline

What work can we eliminate?

Basic strategy.

Workloads, systems, metrics.

Context switching and security.

Paging Advantages

Flexible Addr Space
- don’t need to find contiguous RAM
- doesn’t waste whole data pages (valid bit)

Easy to manage
- fixed size pages
- simple free list for unused pages
- no need to coalesce

Paging Problems

Too big

Too slow

Paging Problems

Too big

Too slow [today's focus]

Translation Steps

H/W: for each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. fetch **PTE**
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. fetch **PA** to register

Translation Steps

H/W: for each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. fetch **PTE**
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. fetch **PA** to register

Which steps are expensive?

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. fetch **PTE**
(cheap) 4. extract **PFN** (page frame num)
(cheap) 5. build **PA** (phys addr)
(expensive) 6. fetch **PA** to register

Which steps are expensive?

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. fetch **PTE**

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

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

(expensive) 6. fetch **PA** to register

Which expensive step can we avoid?

Array Iterator

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

Array Iterator

Virt

load 0x3000

load 0x3004

load 0x3008

load 0x300C

…

Array Iterator

Virt	**Phys**
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Array Iterator

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Array Iterator

Virt	**Phys**
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Array Iterator

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Outline

What work can we eliminate?

Basic strategy.

Workloads, systems, metrics.

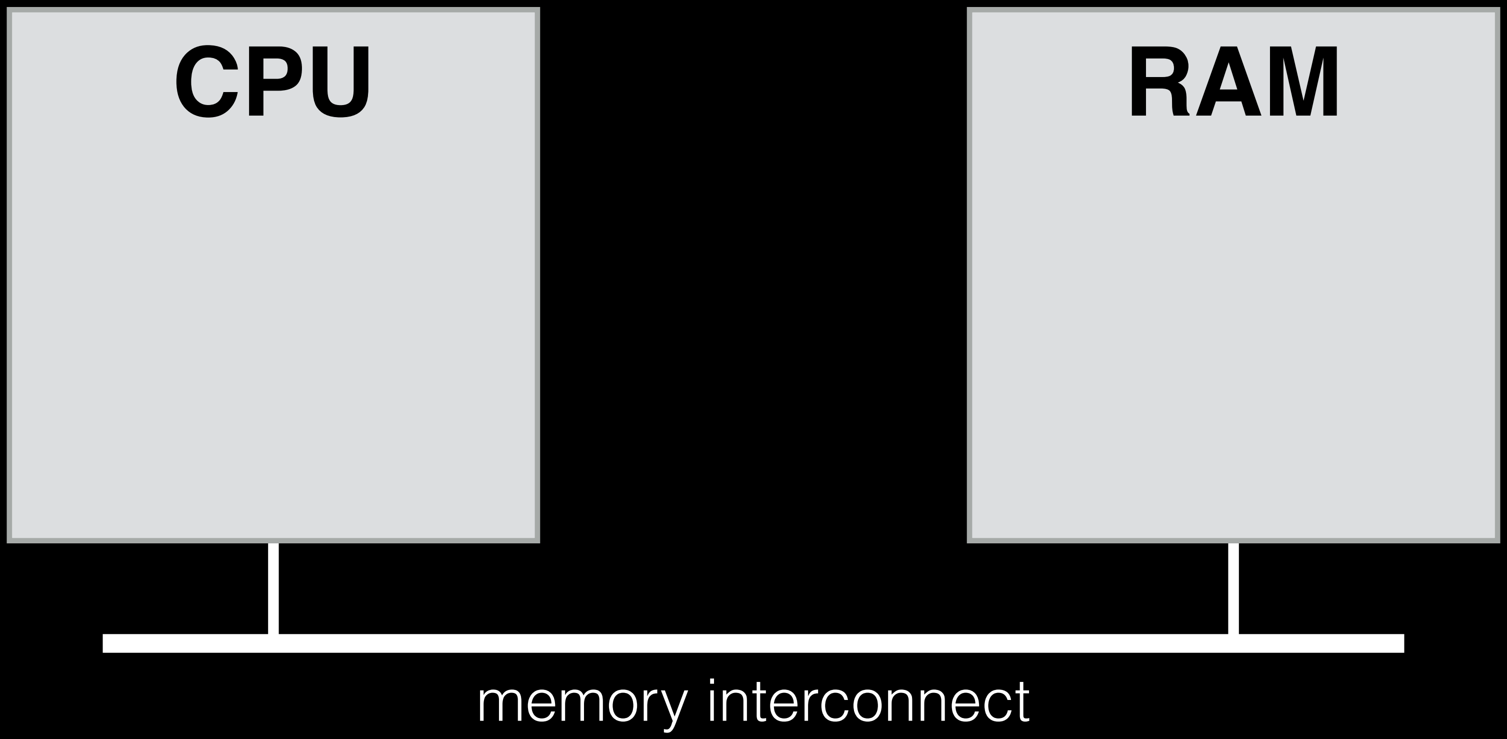
Context switching and security.

Strategy

Take advantage of repetition.
Use a CPU cache.

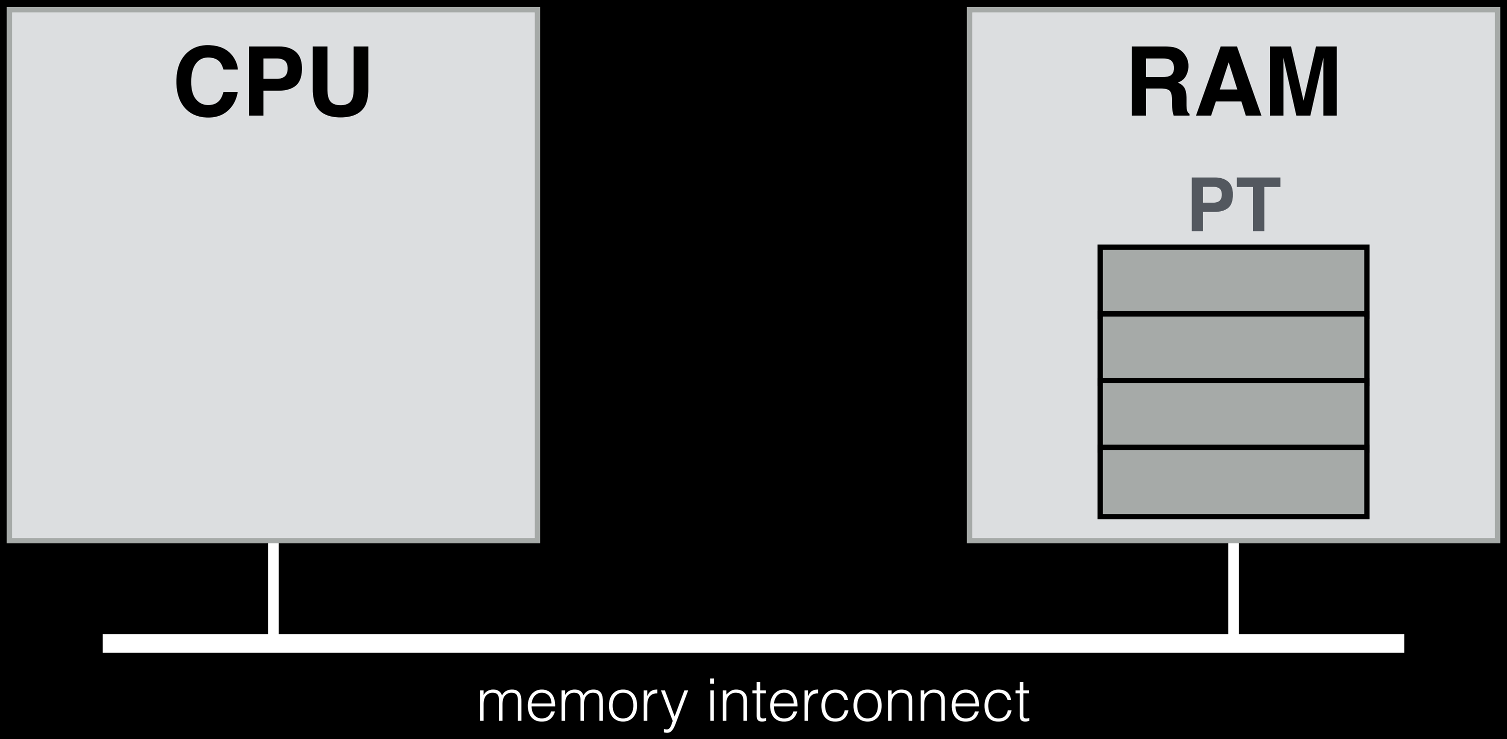
Strategy

Take advantage of repetition.
Use a CPU cache.



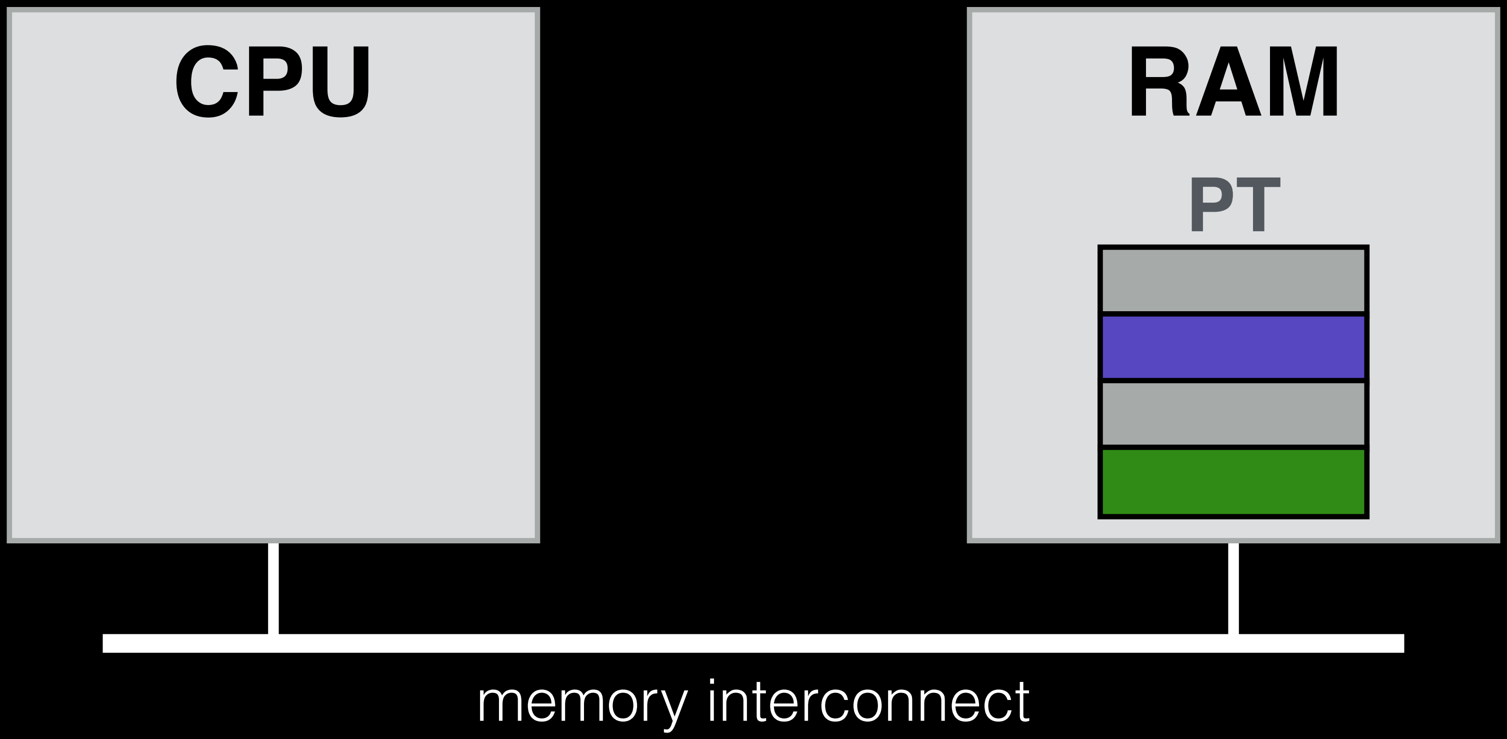
Strategy

Take advantage of repetition.
Use a CPU cache.



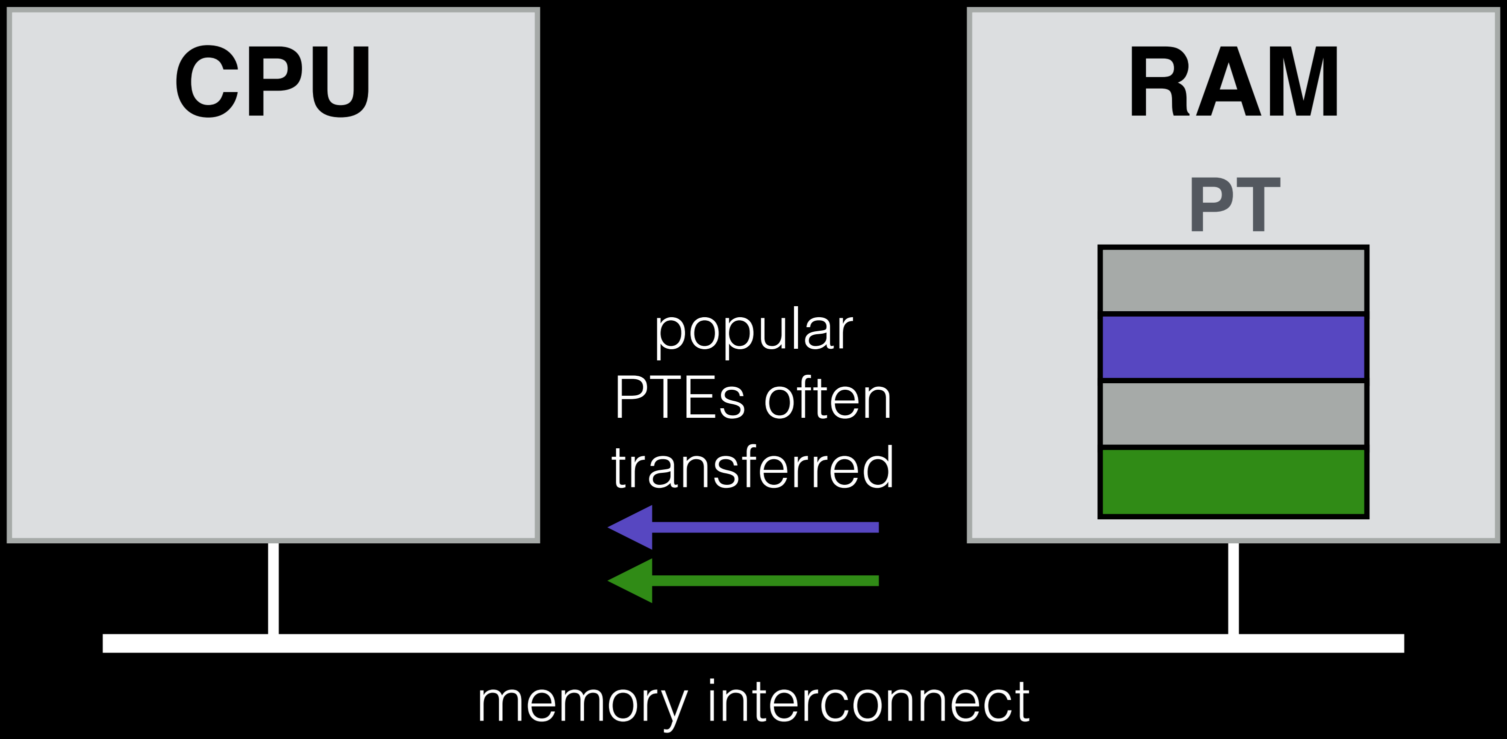
Strategy

Take advantage of repetition.
Use a CPU cache.



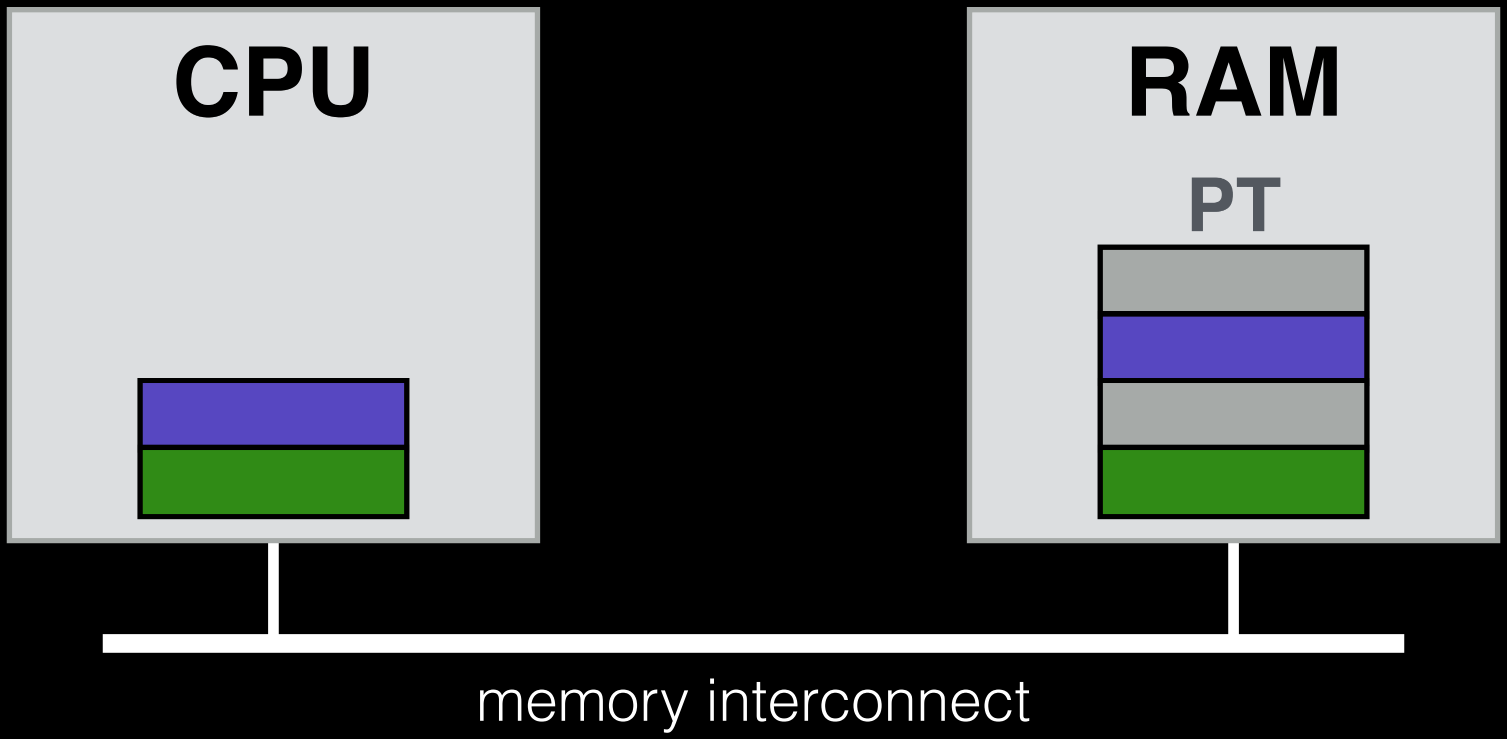
Strategy

Take advantage of repetition.
Use a CPU cache.



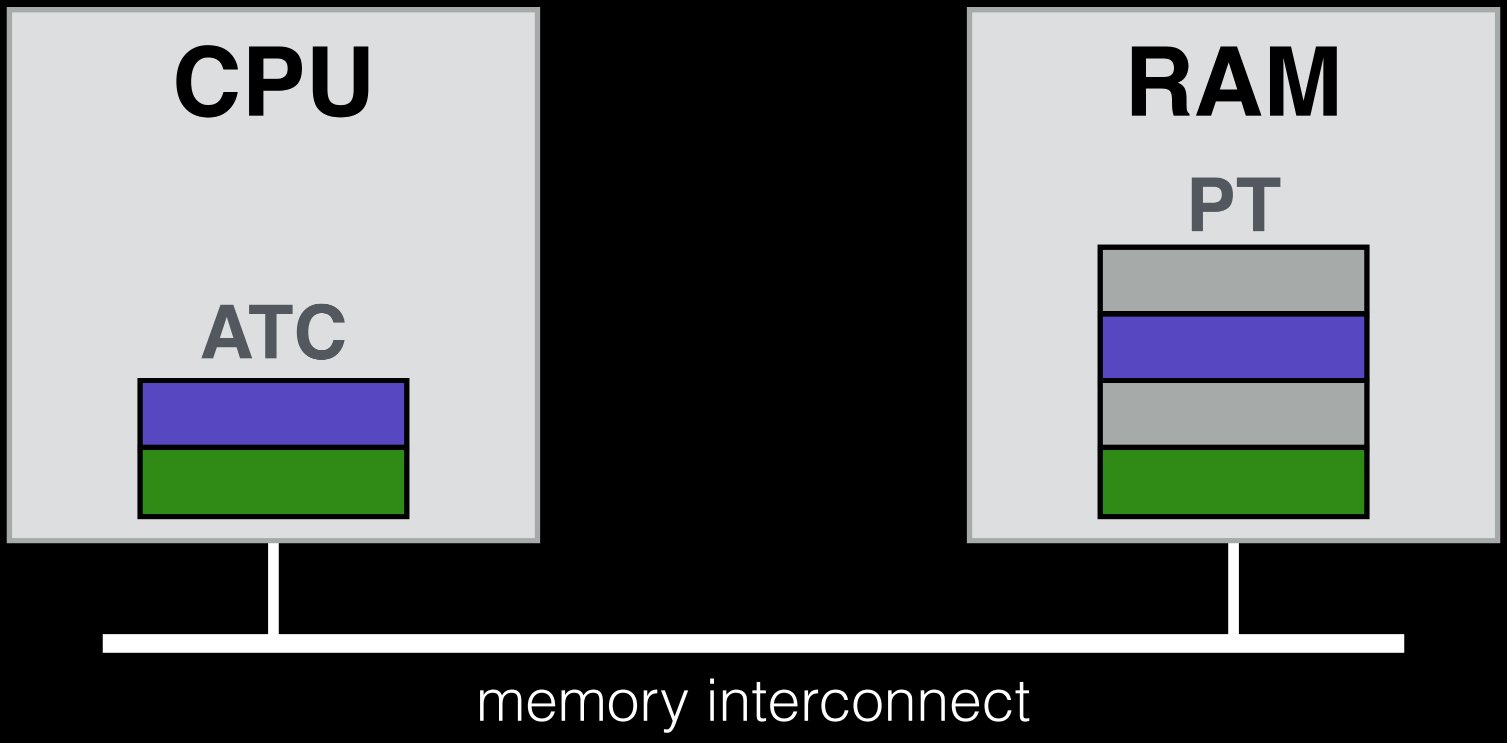
Strategy

Take advantage of repetition.
Use a CPU cache.



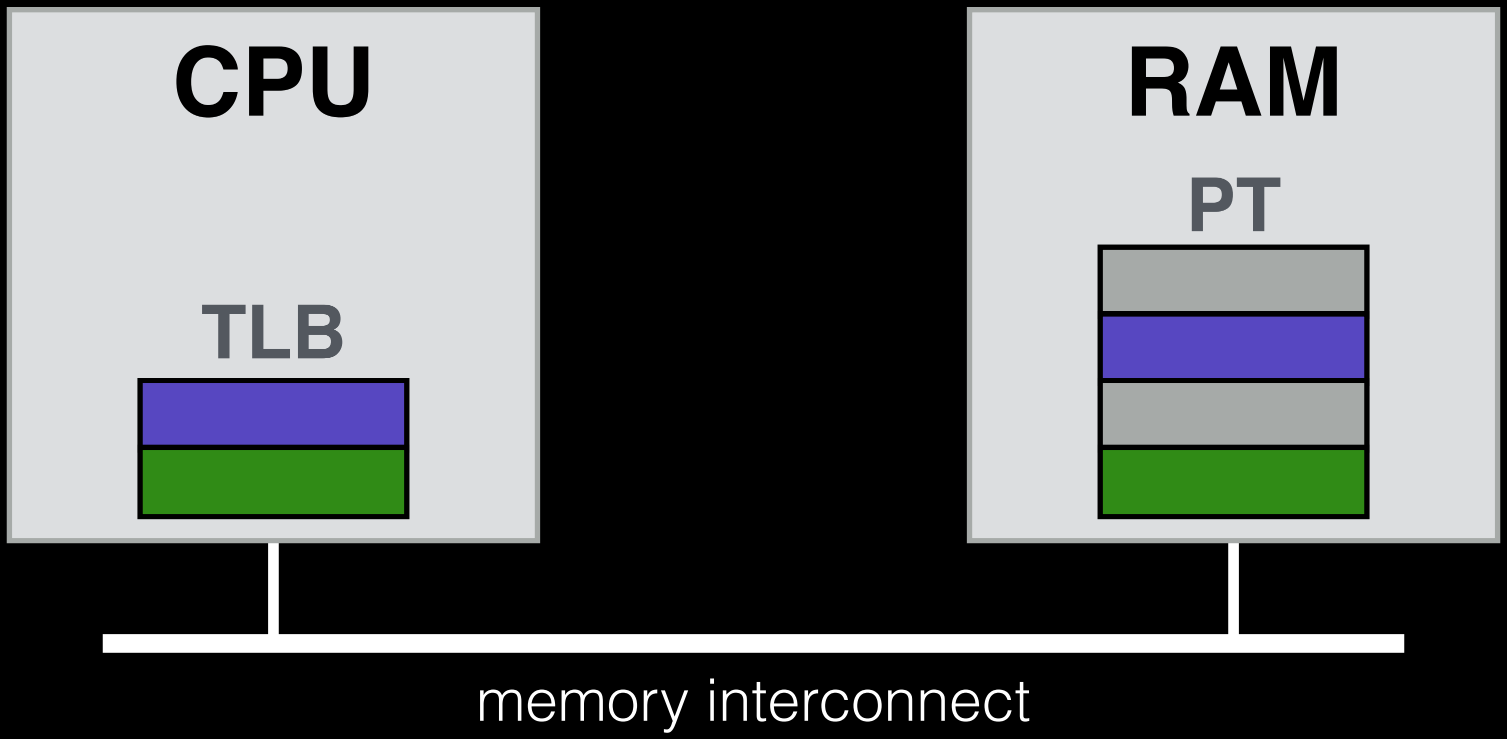
Strategy

Name? ATC: **A**ddress **T**ranslation **C**ache?   [OSTEP]



Strategy

Name? ATC: **A**ddress **T**ranslation **C**ache? [OSTEP]
Nope. TLB: **T**ranslation **L**ookaside **B**uffer



Strategy

Name? ATC: **A**ddress **T**ranslation **C**ache? [OSTEP]
Nope. TLB: **T**ranslation **L**ookaside **B**uffer

ATC

Strategy

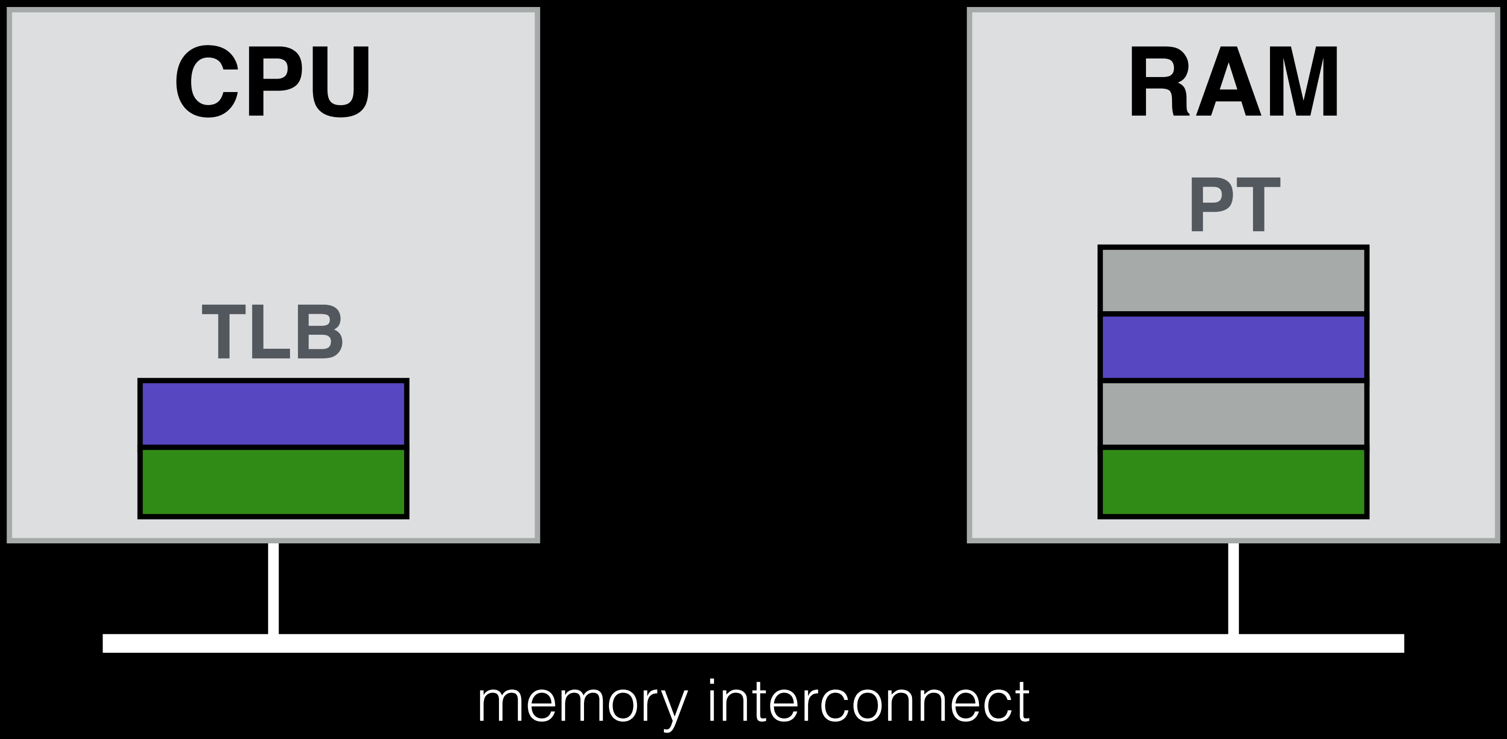
Name? ATC: **A**ddress **T**ranslation **C**ache? [OSTEP]
Nope. TLB: **T**ranslation **L**ookaside **B**uffer

Air **T**raffic **C**ontroller

Strategy

Name? ATC: **A**ddress **T**ranslation **C**ache? [OSTEP]

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



Cache Types (more in CS 552)

Direct-Mapped: only one place to put entries

Four-Way Set Associative: 4 options

Fully-Associative: entries can go anywhere

Cache Types (more in CS 552)

Direct-Mapped: only one place to put entries

Four-Way Set Associative: 4 options

Fully-Associative: entries can go anywhere
- most common for TLBs
- must store whole key/value in cache
- search all in parallel

Array Iterator (w/ TLB)

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

Array Iterator

Virt

load 0x1000

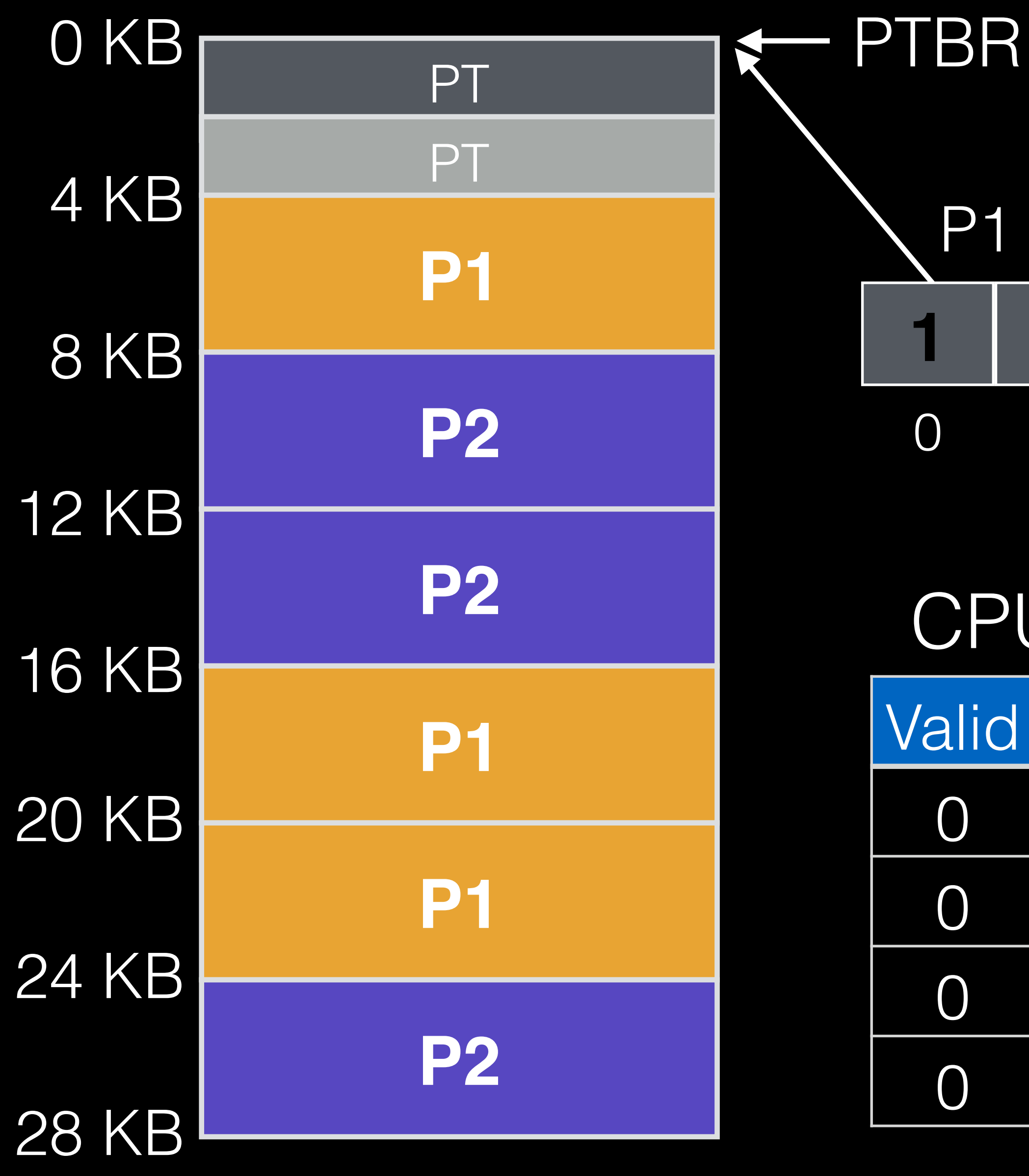
load 0x1004

load 0x1008

load 0x100C

…

Virt	Phys
load 0x1000	
load 0x1004	
load 0x1008	
load 0x100C	
…	



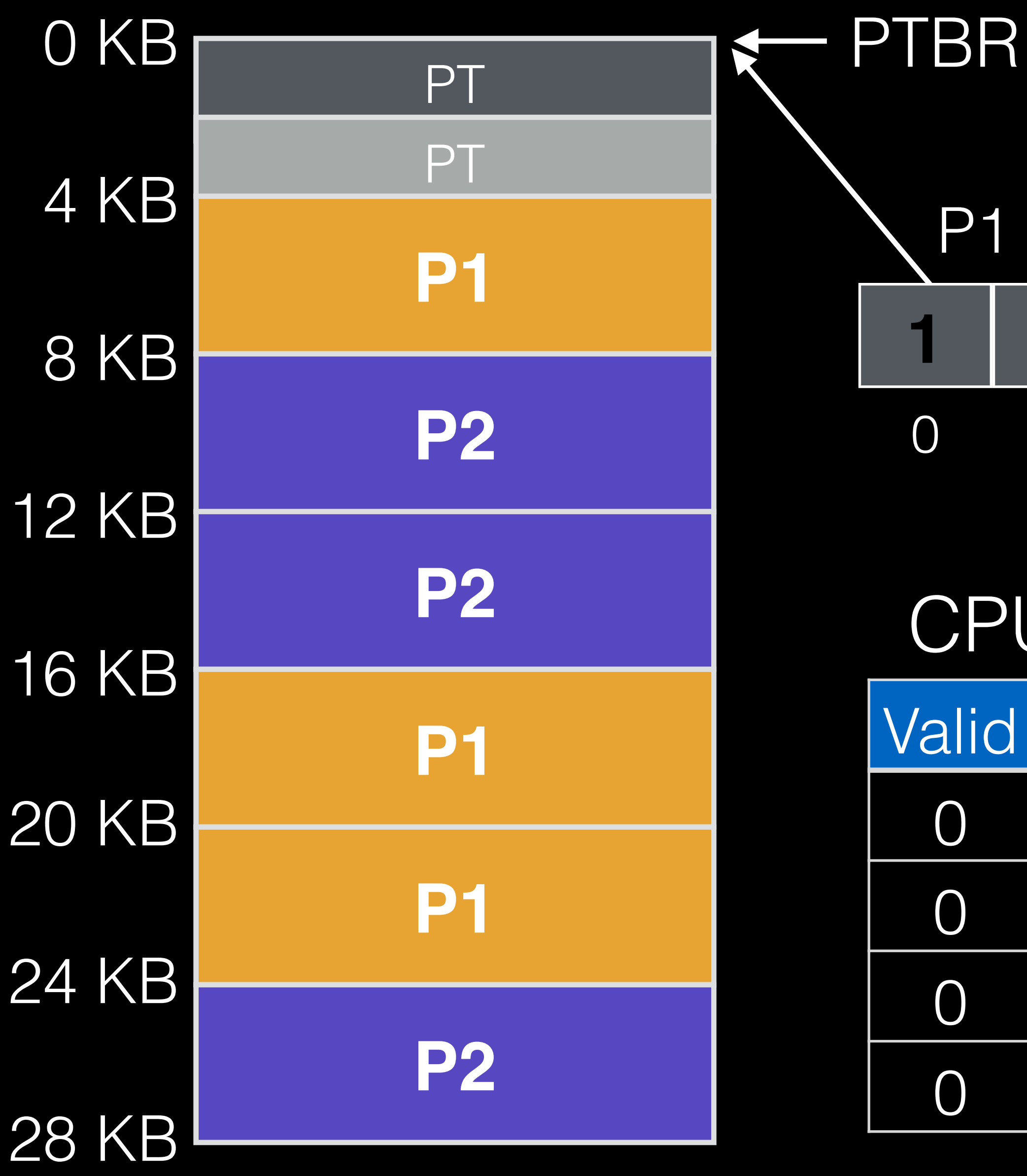
P1 pagetable

0	1	2	3
1	**5**	**4**	**…**

CPU’s TLB

Valid	Virt	Phys
0		
0		
0		
0		

Virt	**Phys**
load 0x1000	
load 0x1004	
load 0x1008	
load 0x100C	
…	



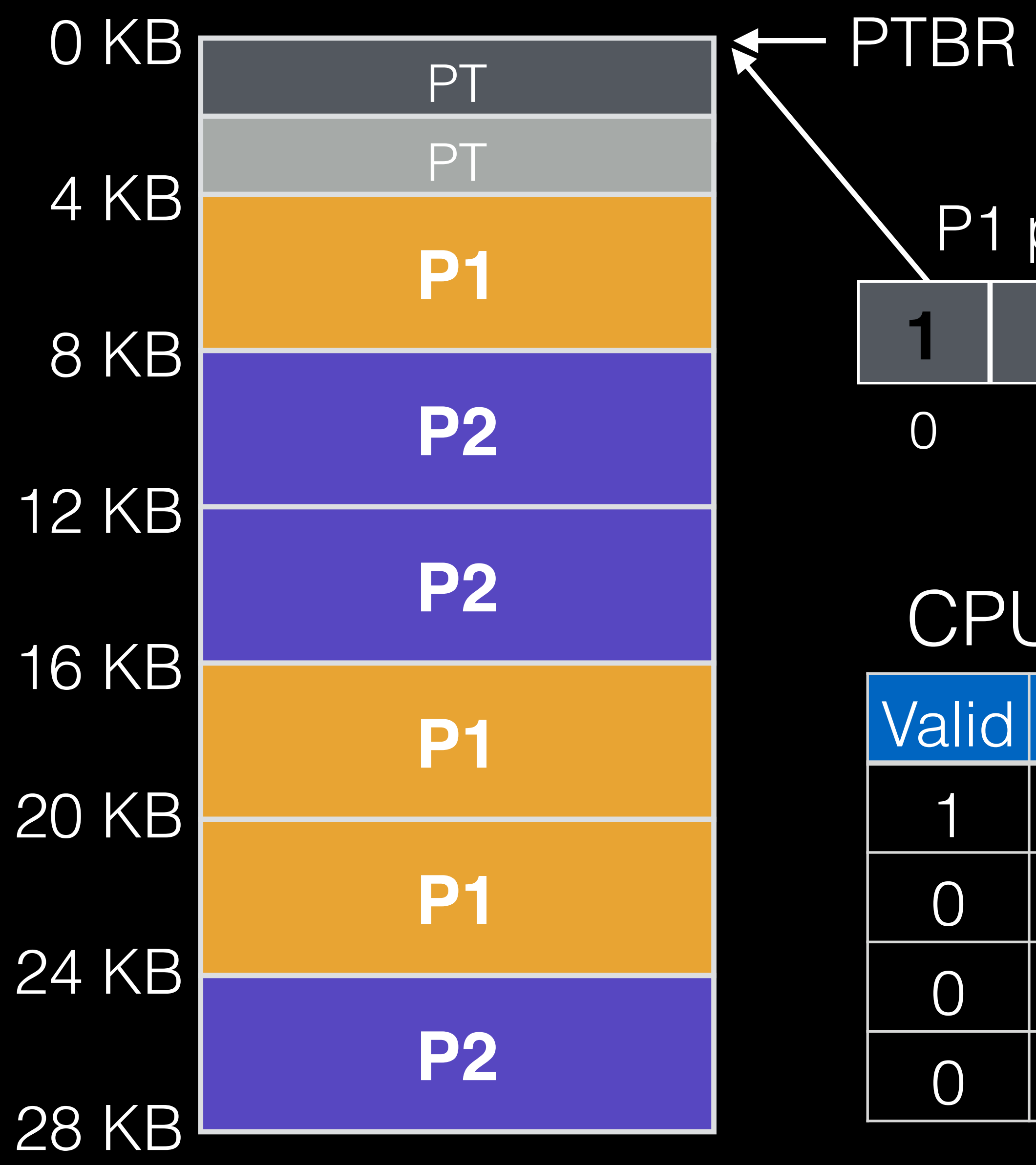
P1 pagetable

1	**5**	**4**	**...**
0	1	2	3

CPU's TLB

Valid	Virt	Phys
0		
0		
0		
0		

Virt	**Phys**
load 0x1000	
load 0x1004	
load 0x1008	
load 0x100C	
...	



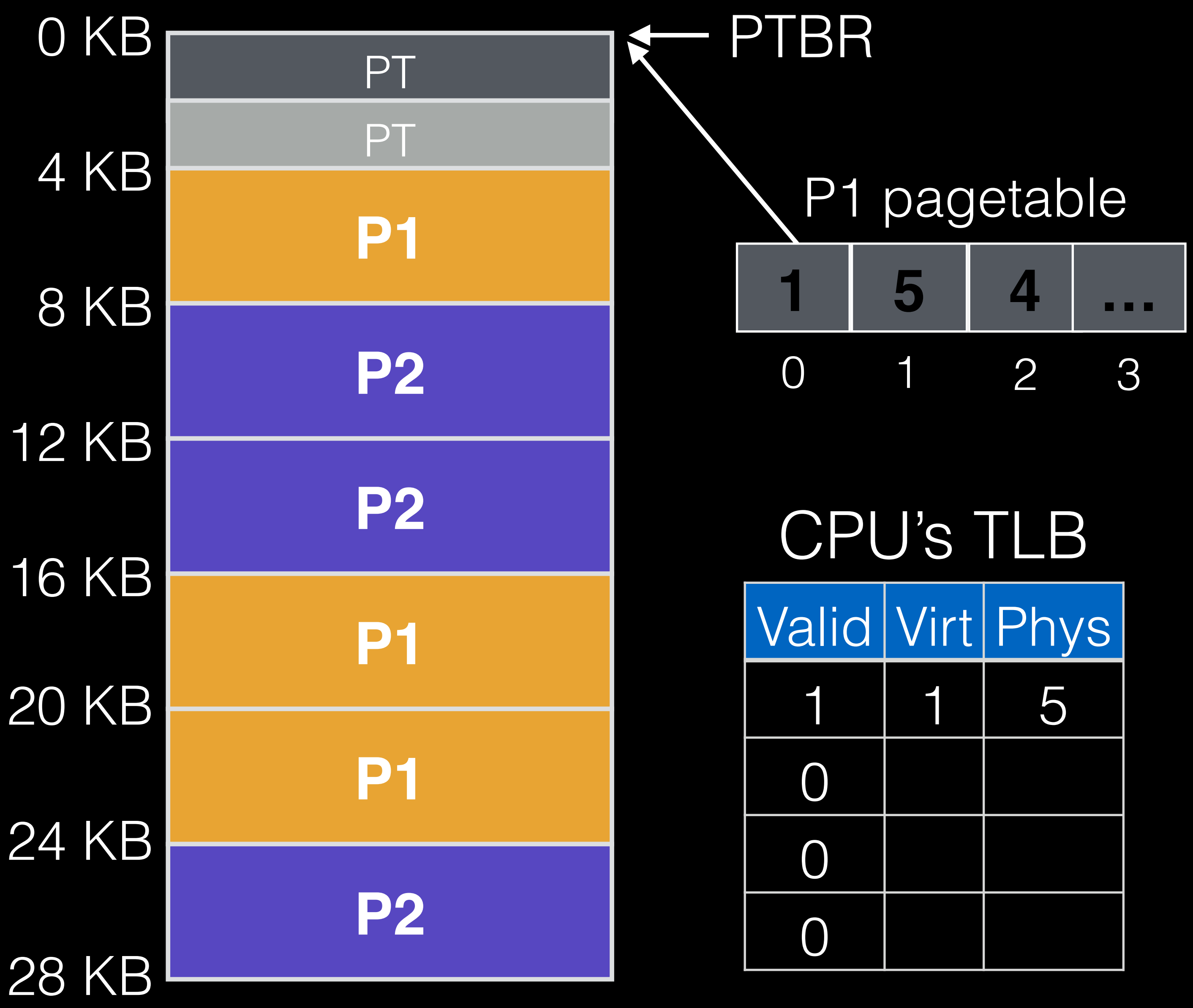
P1 pagetable

1	5	4	...
0	1	2	3

CPU’s TLB

Valid	Virt	Phys
1	1	5
0		
0		
0		

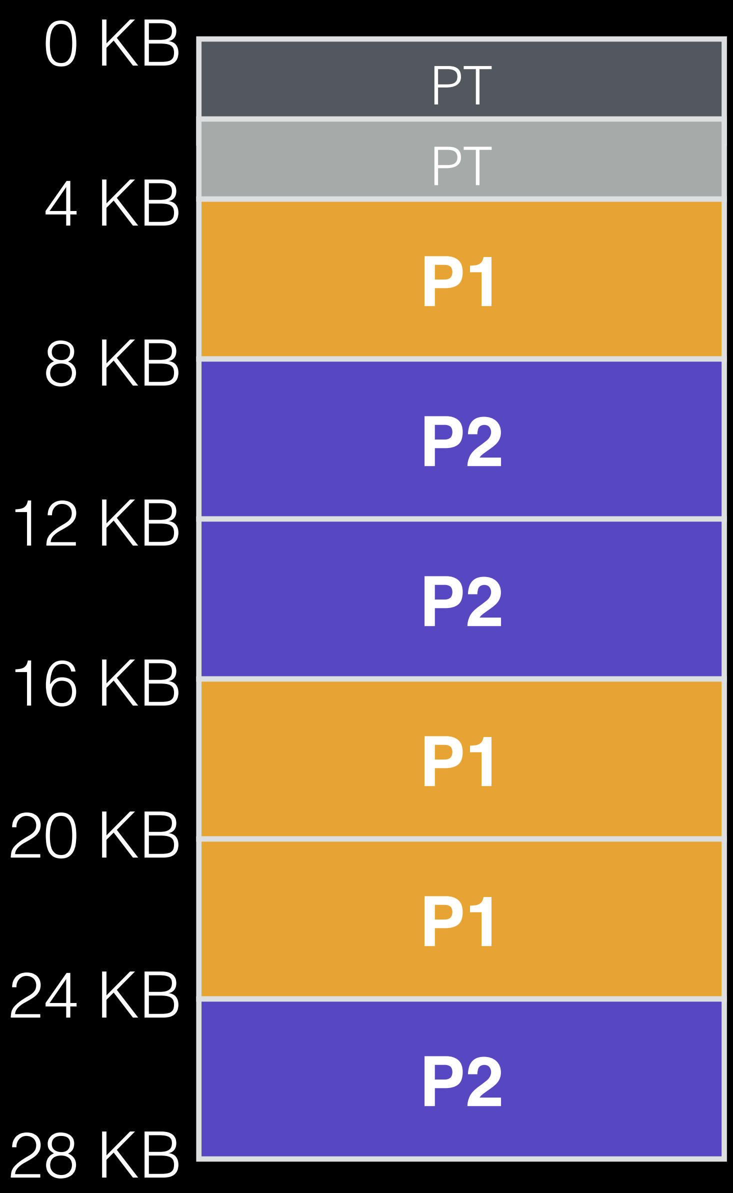
Virt	Phys
load 0x1000	load 0x0004
load 0x1004	
load 0x1008	
load 0x100C	
…	

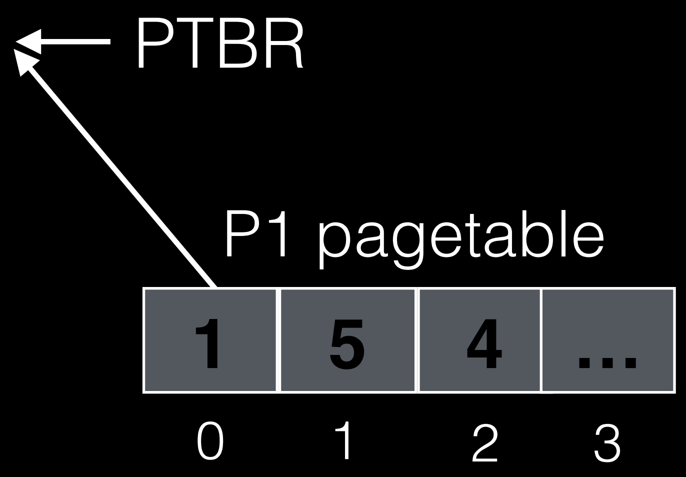


CPU's TLB

Valid	Virt	Phys
1	1	5
0		
0		
0		

Virt	**Phys**
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	
load 0x1008	
load 0x100C	
…	





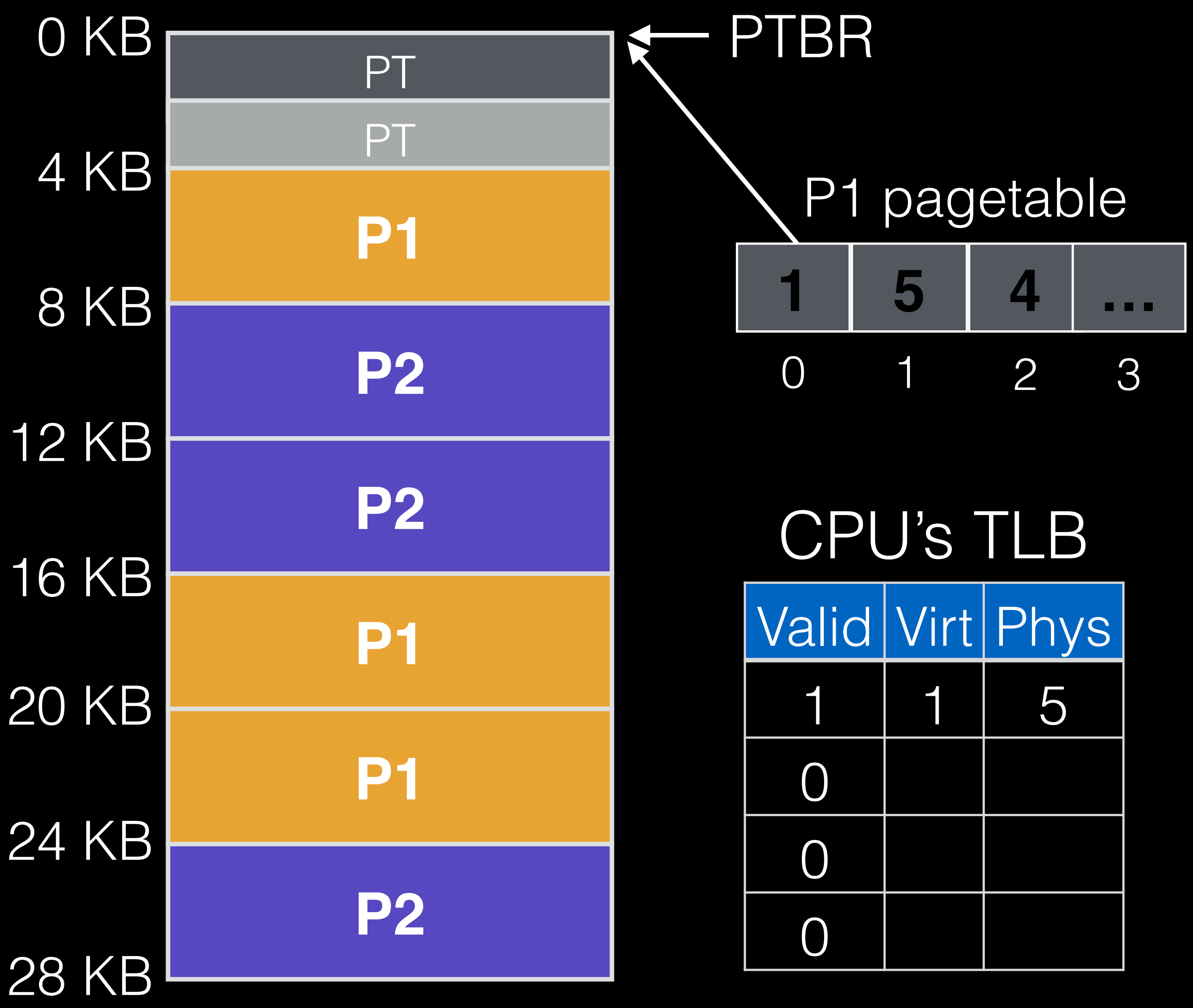
P1 pagetable

0	1	2	3
1	**5**	**4**	**…**

CPU’s TLB

Valid	Virt	Phys
1	1	5
0		
0		
0		

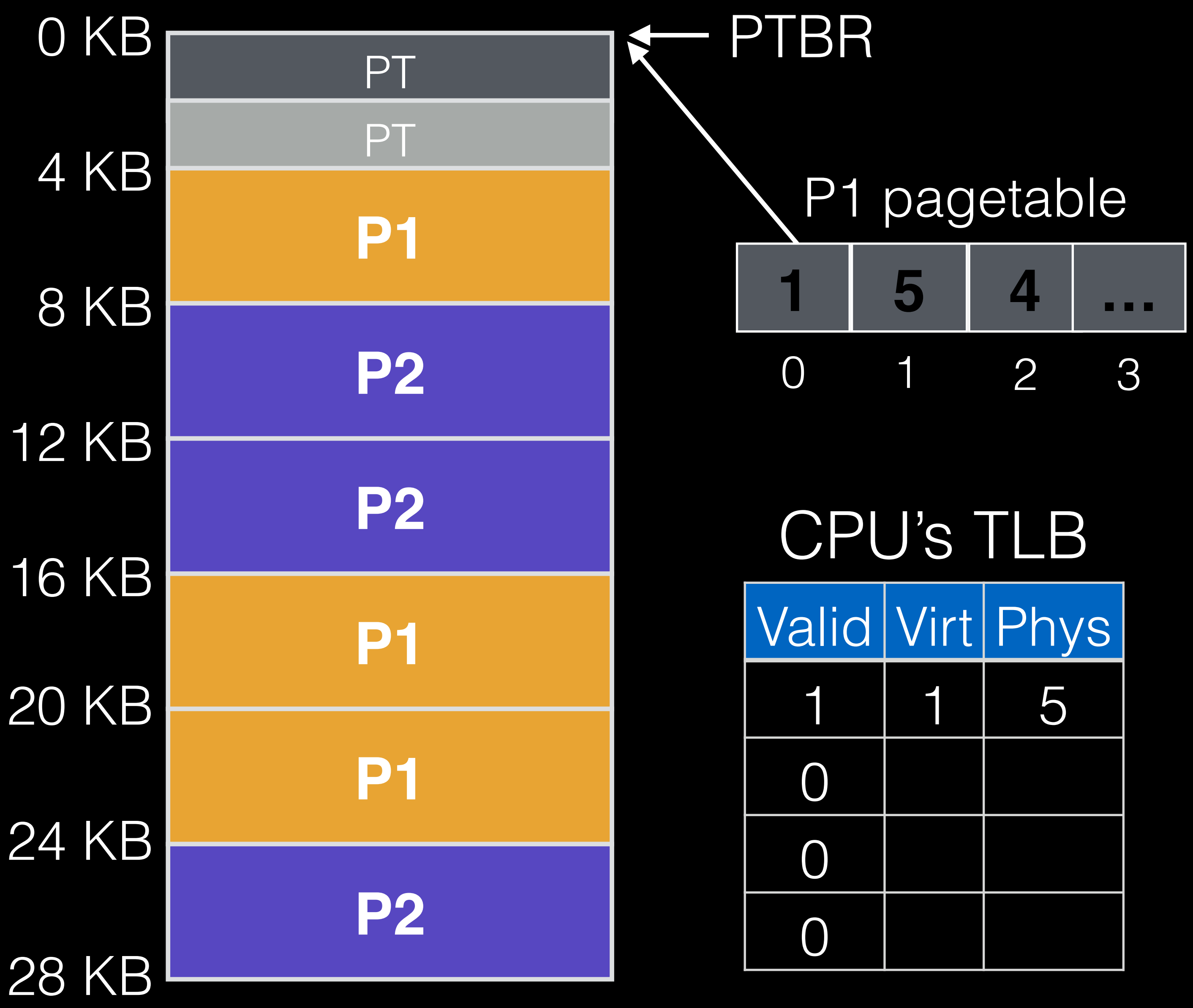
Virt	**Phys**
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	
load 0x1008	
load 0x100C	
…	



CPU’s TLB

Valid	Virt	Phys
1	1	5
0		
0		
0		

Virt	**Phys**
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB)
load 0x1008	
load 0x100C	
. . .	



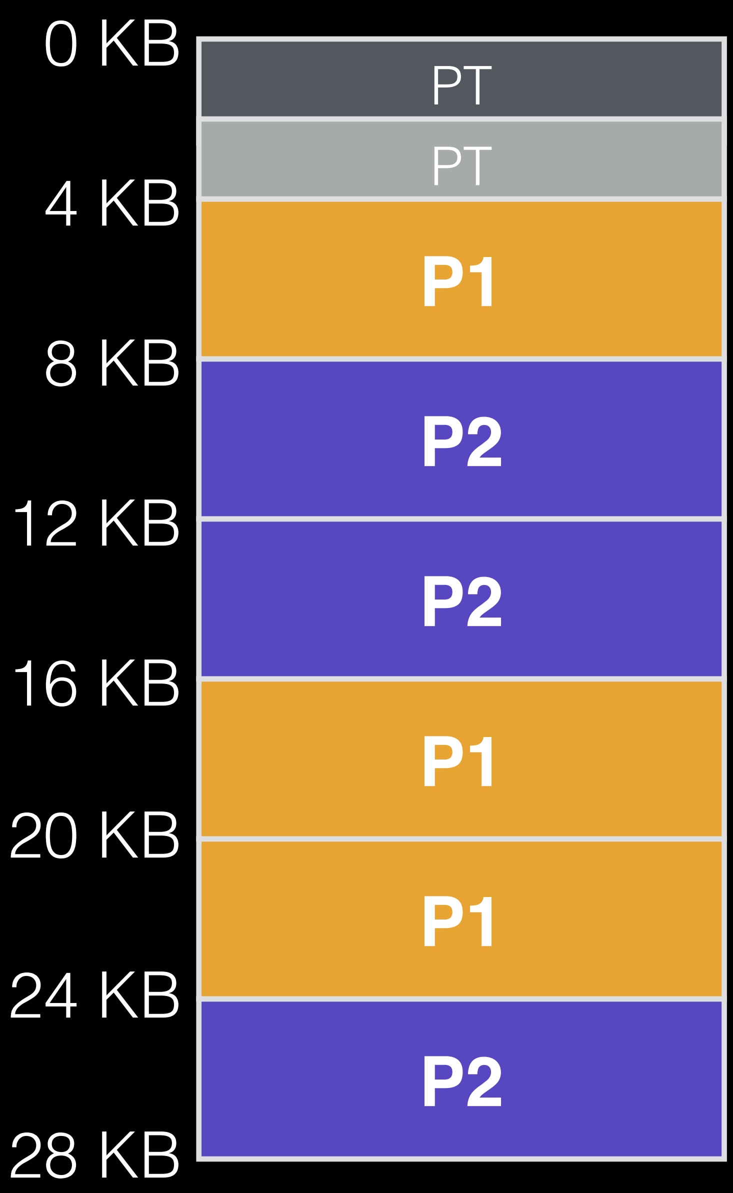
P1 pagetable

0	1	2	3
1	5	4	…

CPU's TLB

Valid	Virt	Phys
1	1	5
0		
0		
0		

Virt	Phys
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	
load 0x100C	
…	



PTBR

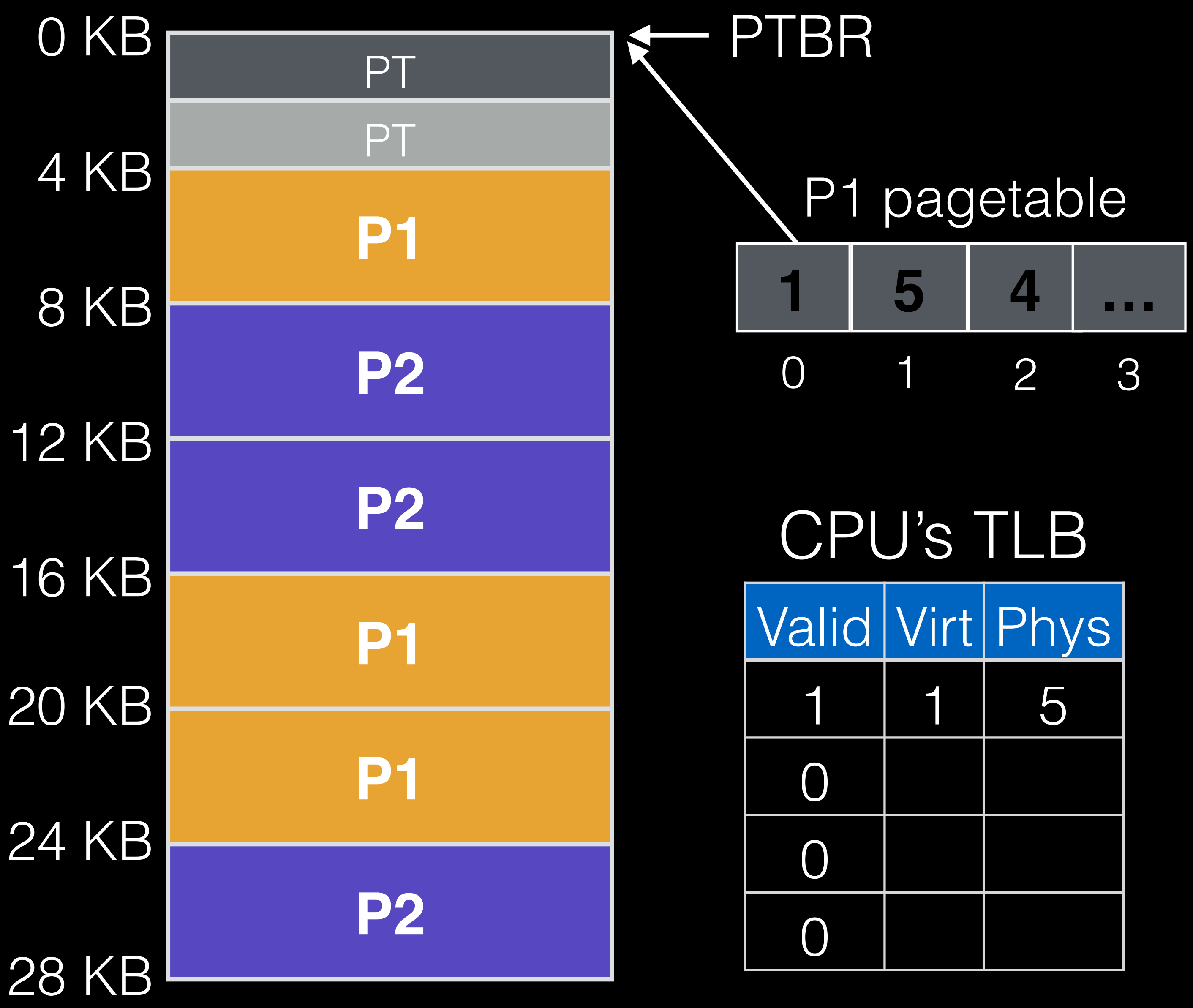
P1 pagetable

0	1	2	3
1	5	4	...

CPU's TLB

Valid	Virt	Phys
1	1	5
0		
0		
0		

Virt	Phys
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	(TLB)
	load 0x5008
load 0x100C	(TLB)
…	load 0x500C

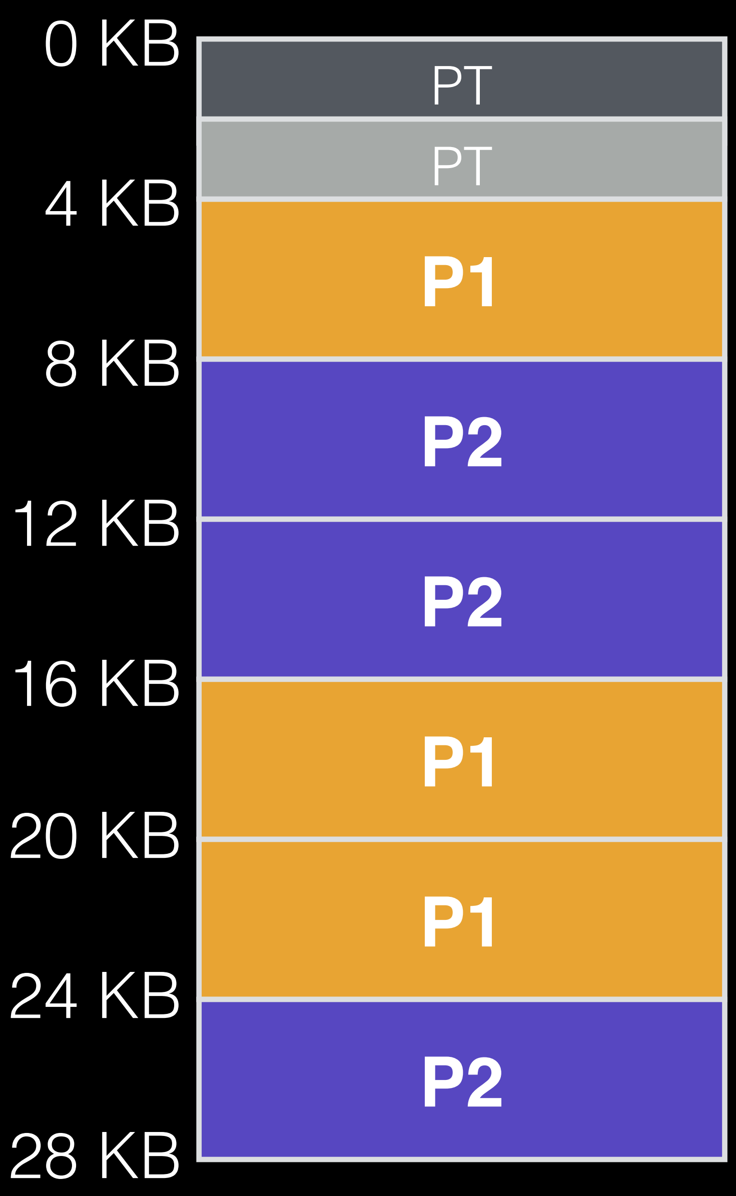


1	5	4	...
0	1	2	3

CPU's TLB

Valid	Virt	Phys
1	1	5
0		
0		
0		

Virt	Phys
load 0x1000	load 0x0004
	load 0x5000
load 0x1004	(TLB)
	load 0x5004
load 0x1008	(TLB)
	load 0x5008
load 0x100C	(TLB)
	load 0x500C
…	
load 0x2000	



PTBR

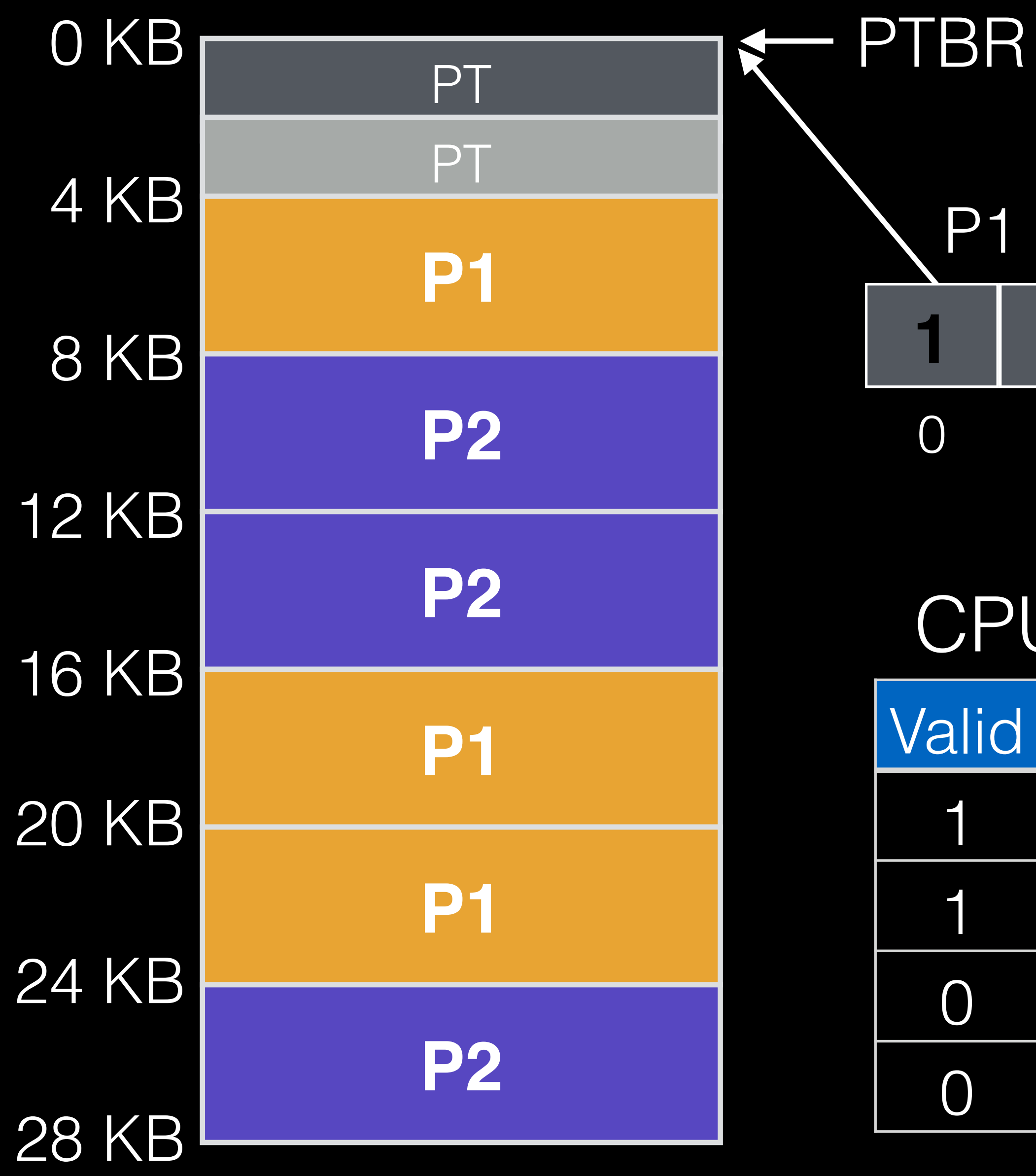
P1 pagetable

0	1	2	3
1	**5**	**4**	**...**

CPU's TLB

Valid	Virt	Phys
1	1	5
1	2	4
0		
0		

Virt	**Phys**
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB) load 0x5004
load 0x1008	(TLB) load 0x5008
load 0x100C	(TLB) load 0x500C
…	
load 0x2000	load 0x0008



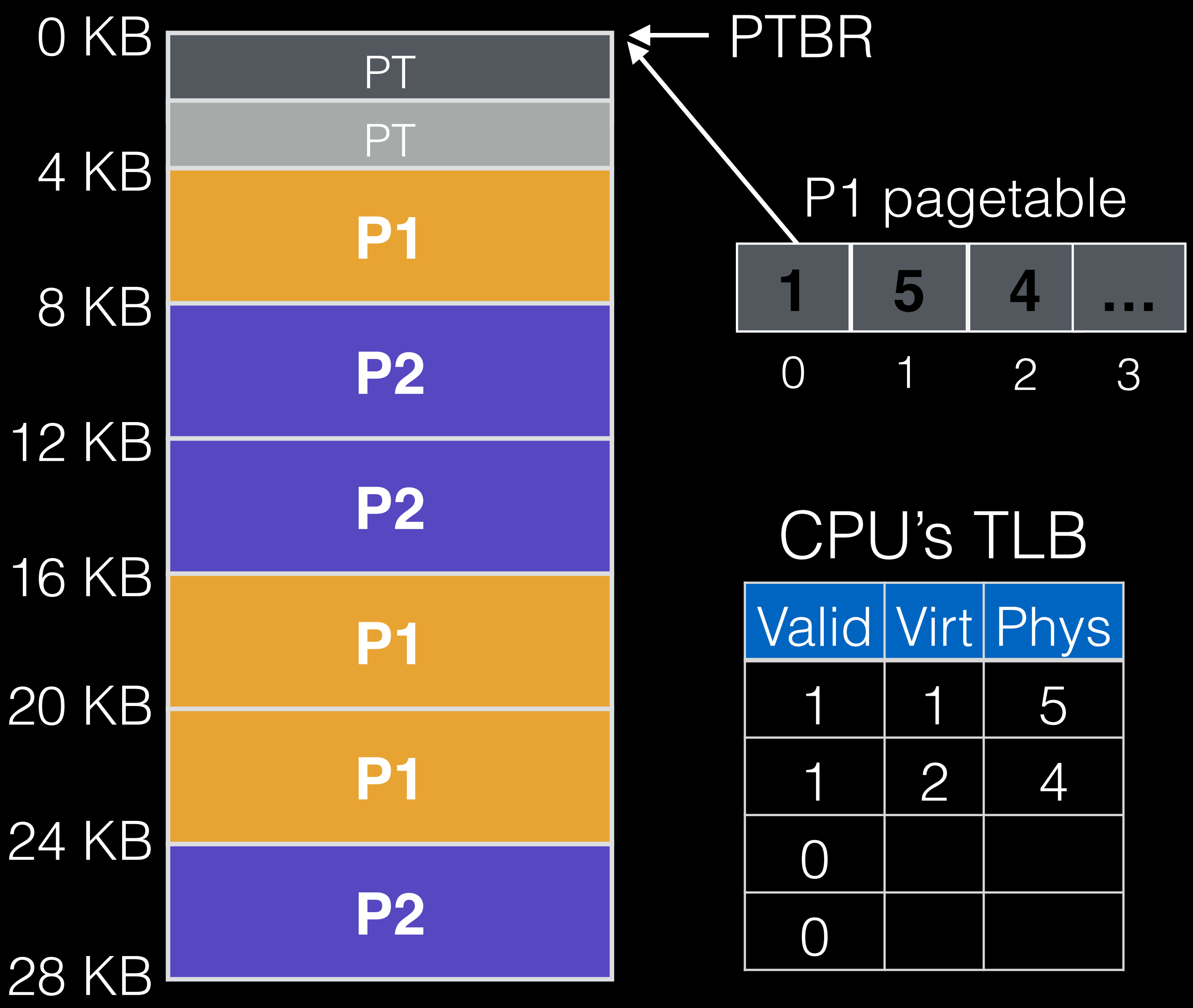
P1 pagetable

0	1	2	3
1	5	4	...

CPU's TLB

Valid	Virt	Phys
1	1	5
1	2	4
0		
0		

Virt	Phys
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB) load 0x5004
load 0x1008	(TLB) load 0x5008
load 0x100C	(TLB) load 0x500C
…	
load 0x2000	load 0x0008 load 0x4000



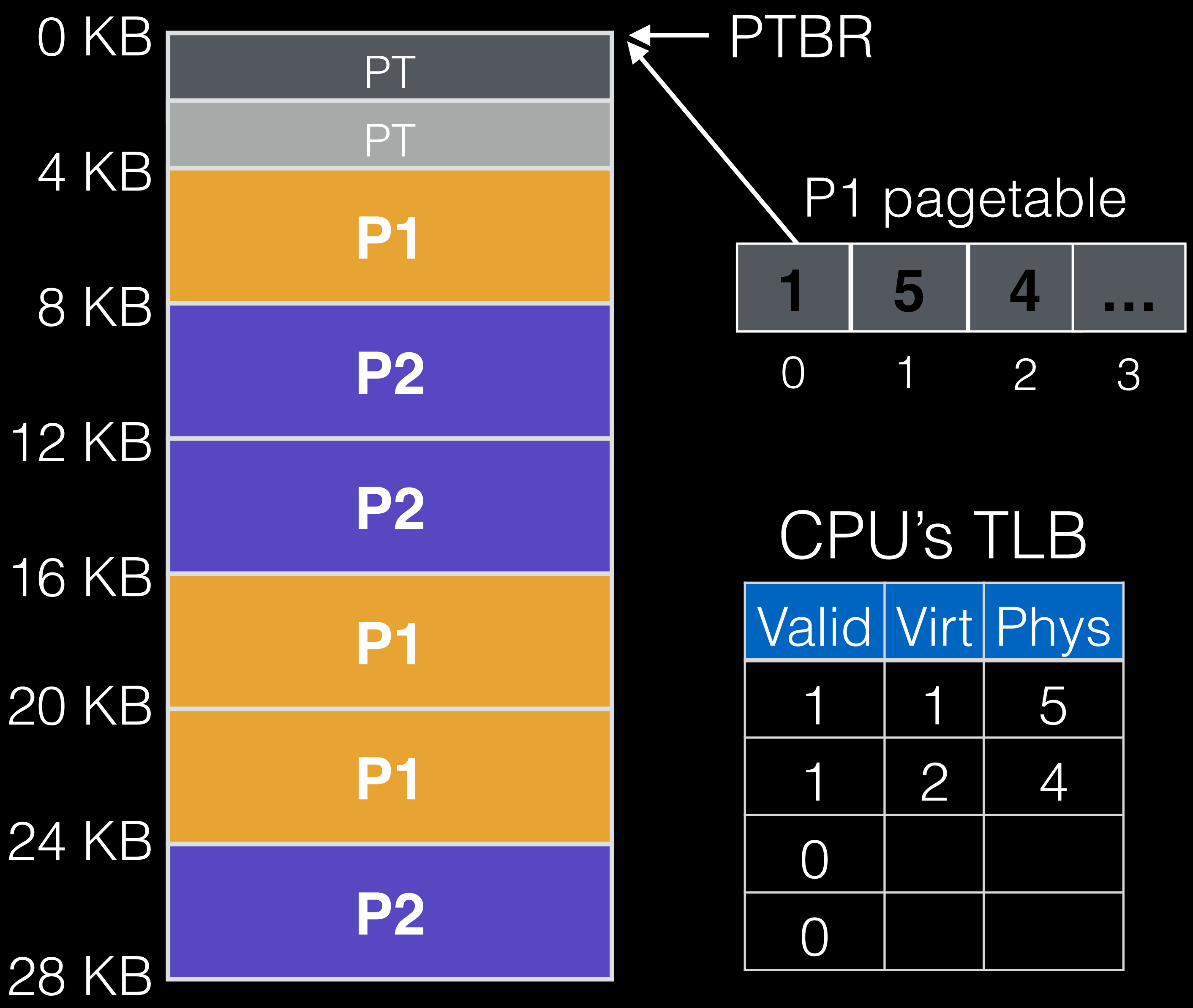
P1 pagetable

1	5	4	...
0	1	2	3

CPU's TLB

Valid	Virt	Phys
1	1	5
1	2	4
0		
0		

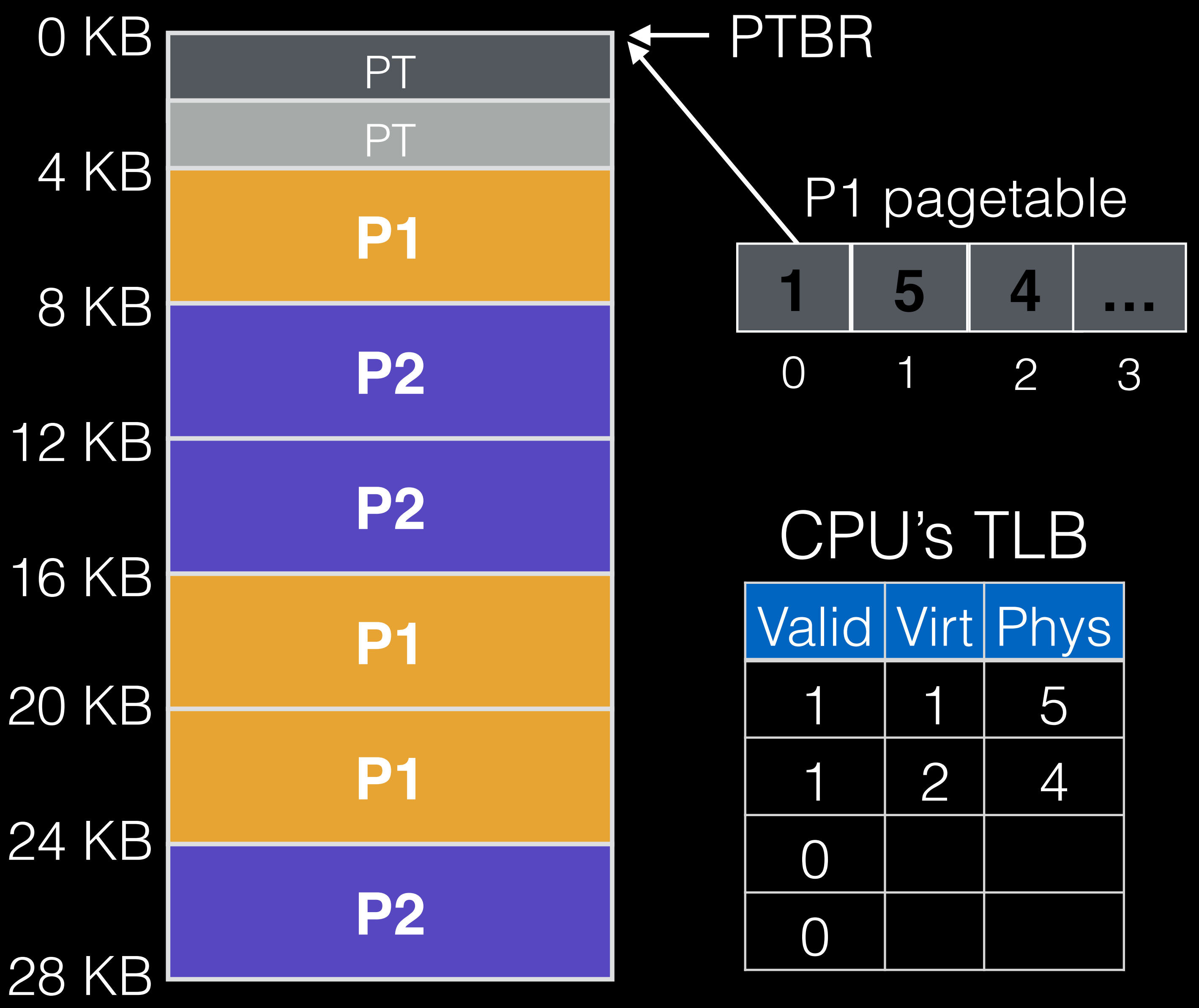
Virt	Phys
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB) load 0x5004
load 0x1008	(TLB) load 0x5008
load 0x100C	(TLB) load 0x500C
...	
load 0x2000	load 0x0008 load 0x4000
load 0x2004	



CPU's TLB

Valid	Virt	Phys
1	1	5
1	2	4
0		
0		

Virt	Phys
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB) load 0x5004
load 0x1008	(TLB) load 0x5008
load 0x100C	(TLB) load 0x500C
…	
load 0x2000	load 0x0008 load 0x4000
load 0x2004	(TLB)



P1 pagetable

0	1	2	3
1	5	4	…

CPU’s TLB

Valid	Virt	Phys
1	1	5
1	2	4
0		
0		

Virt	Phys
load 0x1000	load 0x0004 load 0x5000
load 0x1004	(TLB) load 0x5004
load 0x1008	(TLB) load 0x5008
load 0x100C	(TLB) load 0x500C
…	
load 0x2000	load 0x0008 load 0x4000
load 0x2004	(TLB) 0x4004

How many TLB lookups?

(assume 1KB pages)

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

How many TLB lookups?

(assume 1KB pages)

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

2048/sizeof(int) = **512**

How many TLB “misses”?

(assume 1KB pages)

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

How many TLB “misses”?

(assume 1KB pages)

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

if a%4096 is 0, then **2** else **3**

Miss rate?

(assume 1KB pages)

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

2/512 = **0.4%** or 3/512 = **0.6%**

Hit rate?

(assume 1KB pages)

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

510/512 = **99.6%** or 509/512 = **99.4%**

Outline

What work can we eliminate?

Basic strategy.

Workloads, systems, metrics.

Context switching and security.

Reasoning about TLB

Workload: series of loads/stores to accesses

TLB: chooses entries to store in CPU

Metric: performance (i.e., hit rate)

TLB “algebra”, given 2 variables, find the 3rd:

$$f(\mathbf{W}, \mathbf{T}) = \mathbf{M}$$

Reasoning about TLB

Workload: series of loads/stores to accesses

TLB: chooses entries to store in CPU

Metric: performance (i.e., hit rate)

TLB “algebra”, given 2 variables, find the 3rd:

$$f(\mathbf{W}, \mathbf{T}) = \mathbf{M}$$

TLB Workloads

Sequential array accesses can almost always hit in the TLB, and so are very fast!

What pattern would be slow?

TLB Workloads

Sequential array accesses can almost always hit in the TLB, and so are very fast!

What pattern would be slow?
- highly random, with no repeat accesses

Workload Characteristics

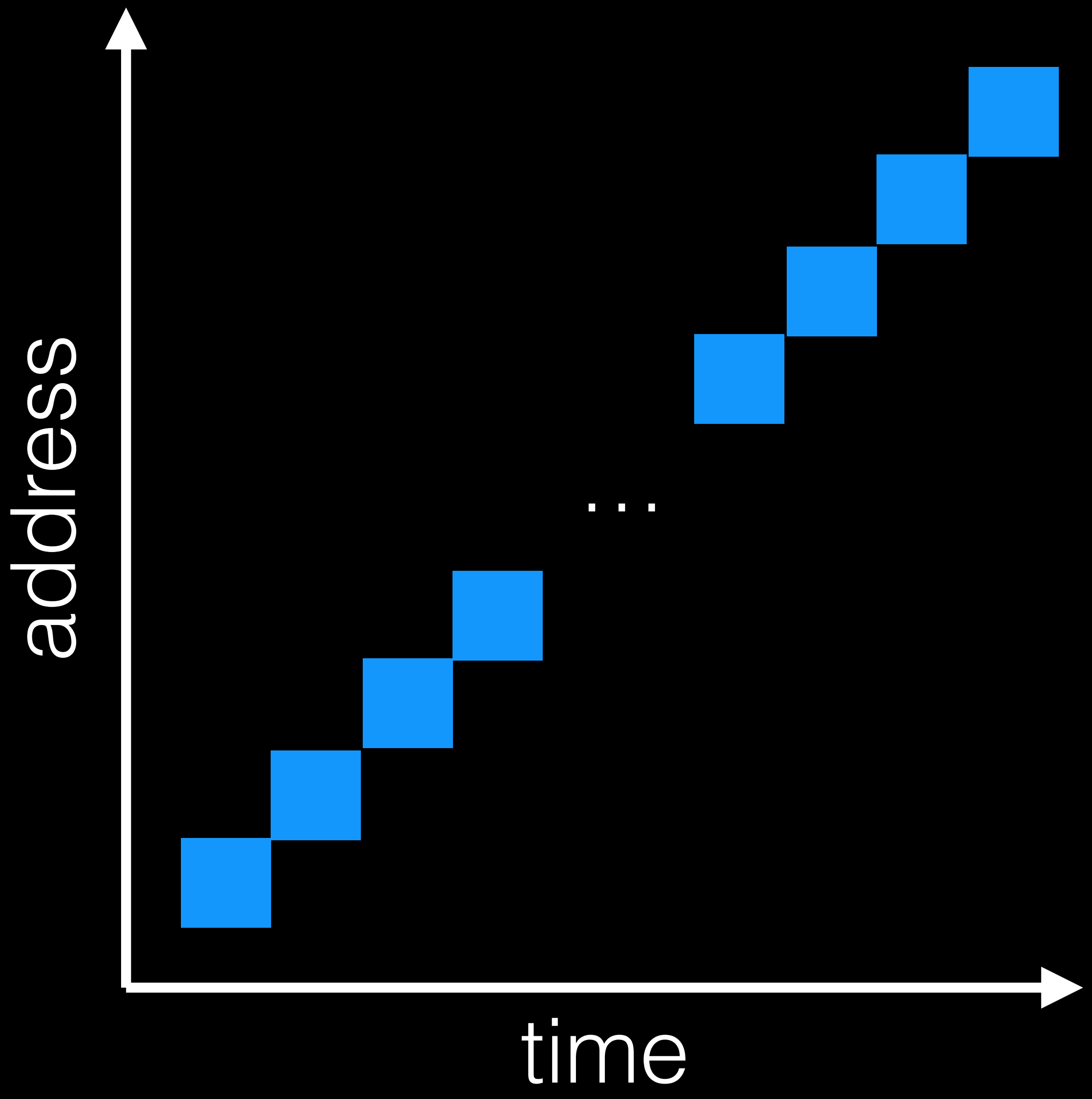
Workload A

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

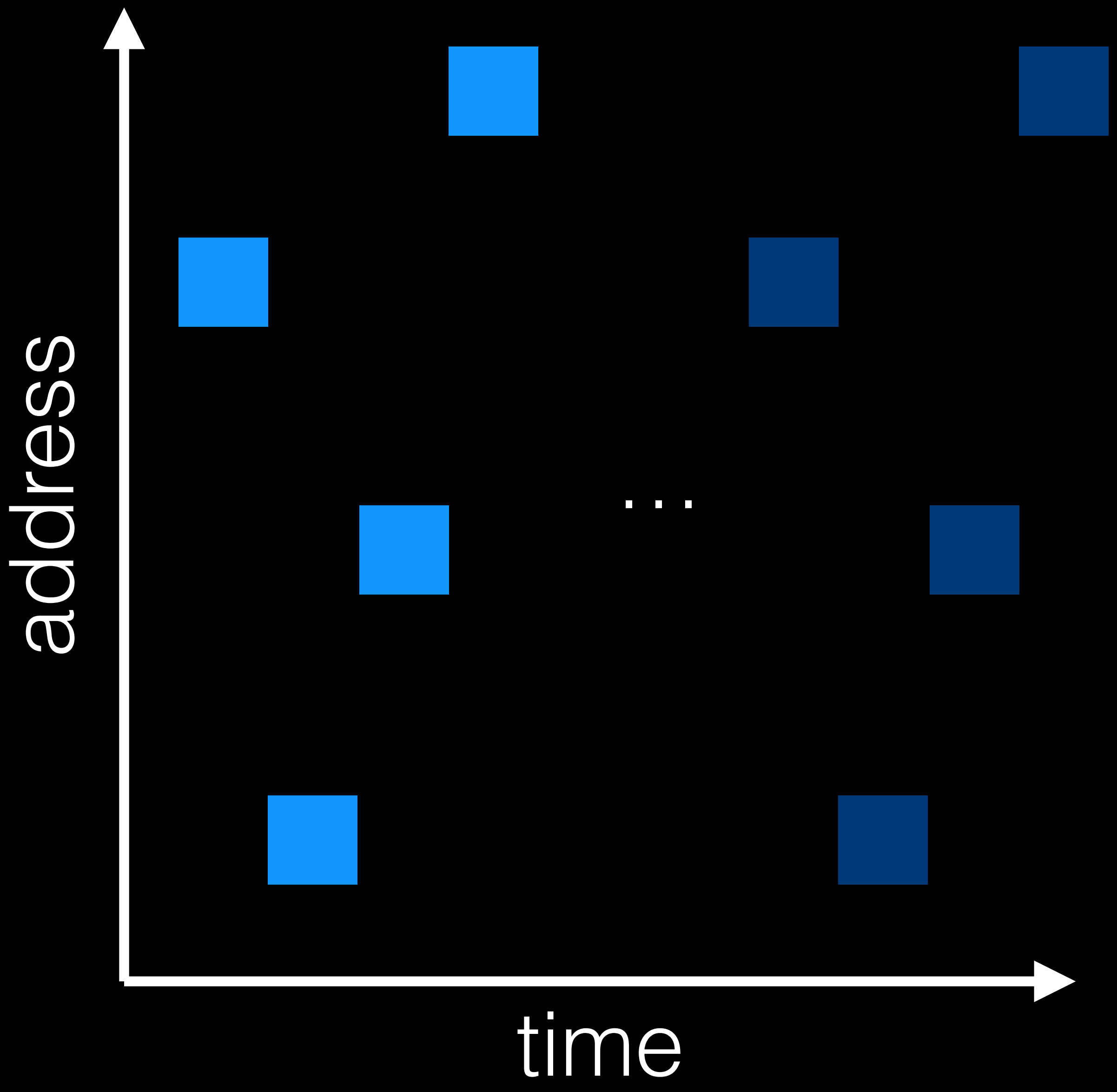
Workload B

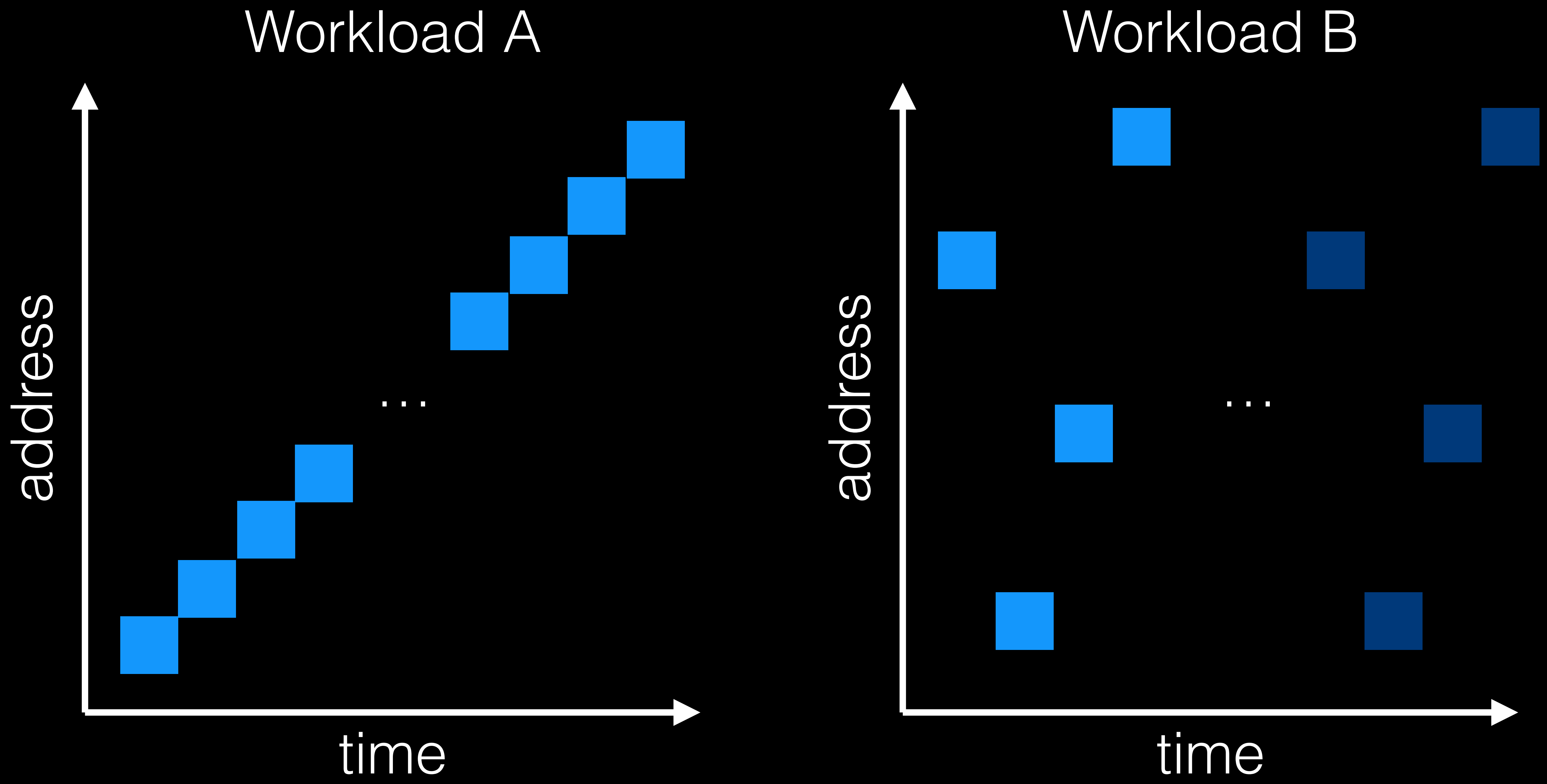
```
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
   sum += a[rand() % N];
}
```

?


address
...
time

?

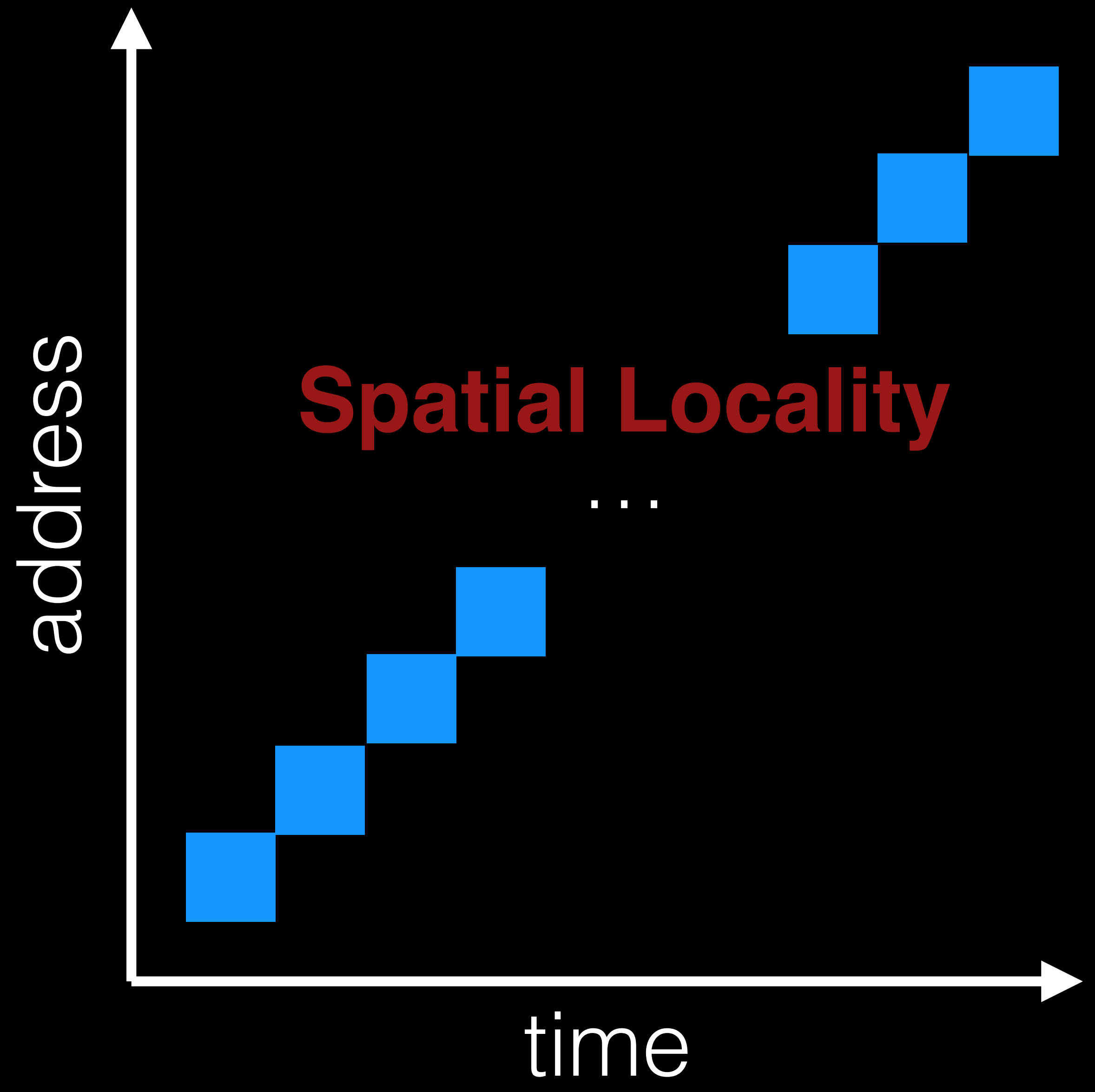

address
...
time



Workload A

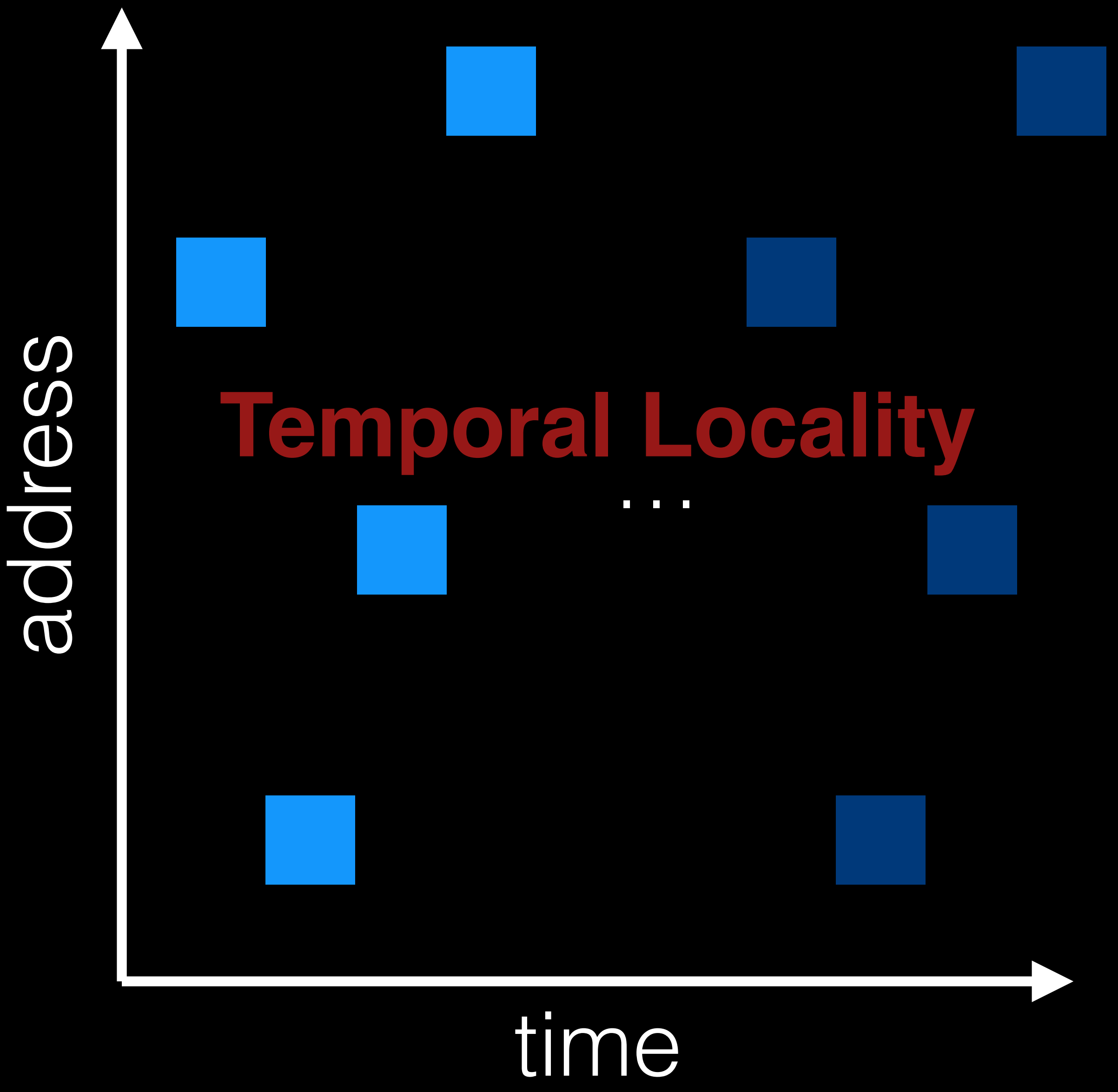
Workload B

Workload A



Spatial Locality

Workload B



Temporal Locality

Workload Locality

Spatial Locality: future access will be to nearby addresses

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

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?

A couple policies

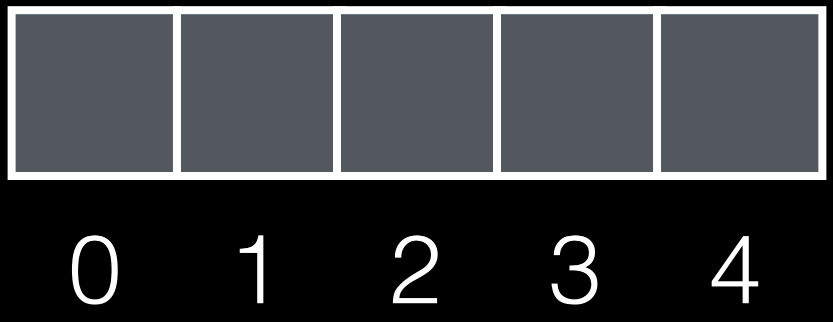
LRU: evict least-recently used a TLB slot is needed

Random: randomly choose entries to evict

When is each better?

LRU Troubles

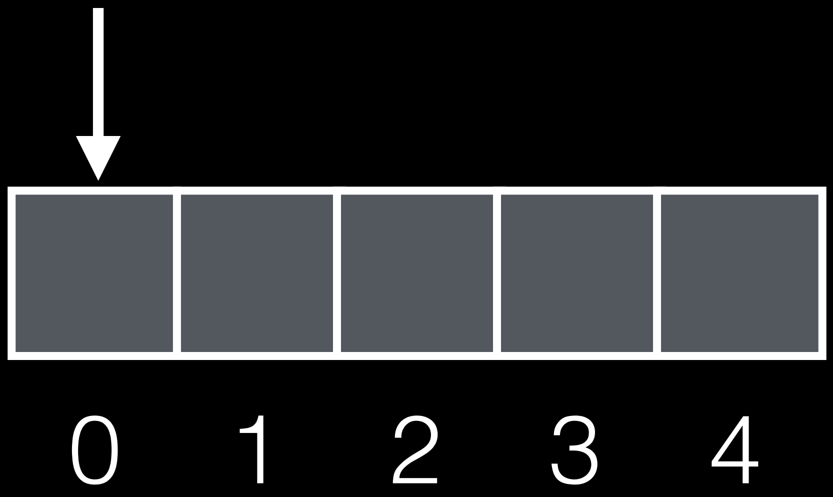
virtual addresses:



Valid	Virt	Phys
0		
0		
0		
0		

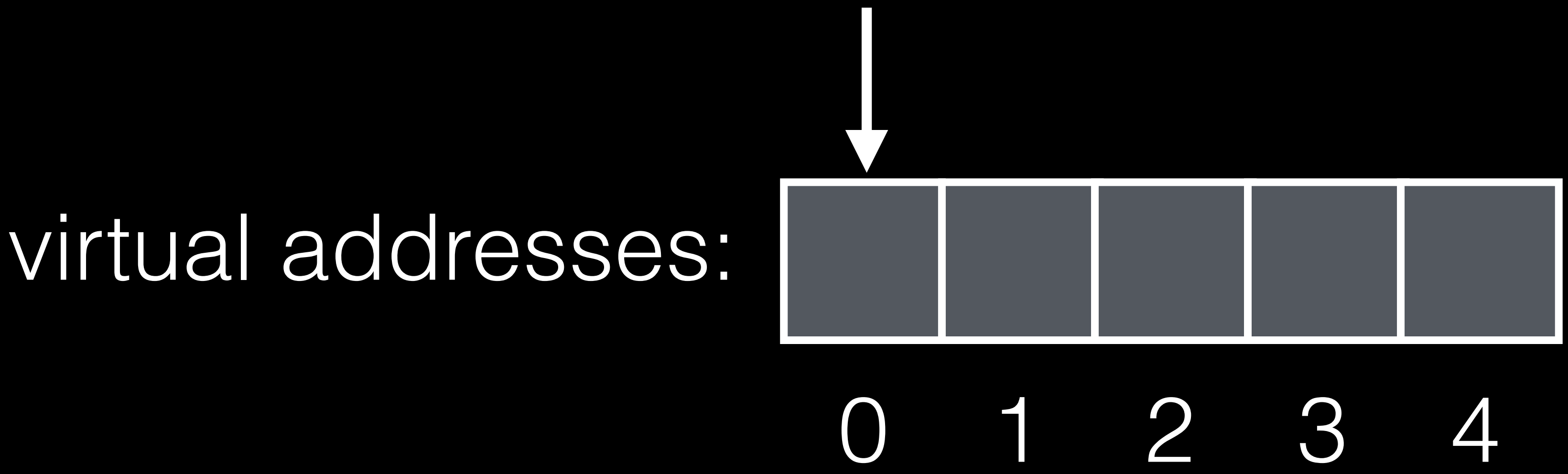
LRU Troubles

virtual addresses:



Valid	Virt	Phys
0		
0		
0		
0		

LRU Troubles

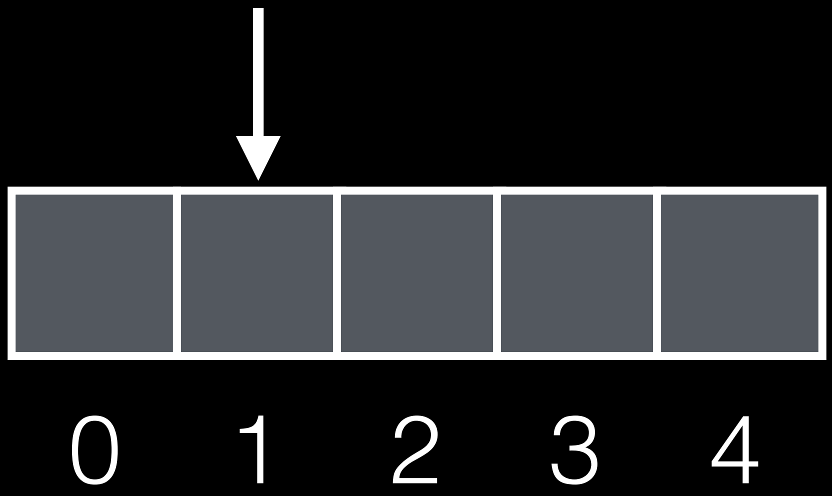


Valid	Virt	Phys
1	0	?
0		
0		
0		

miss!

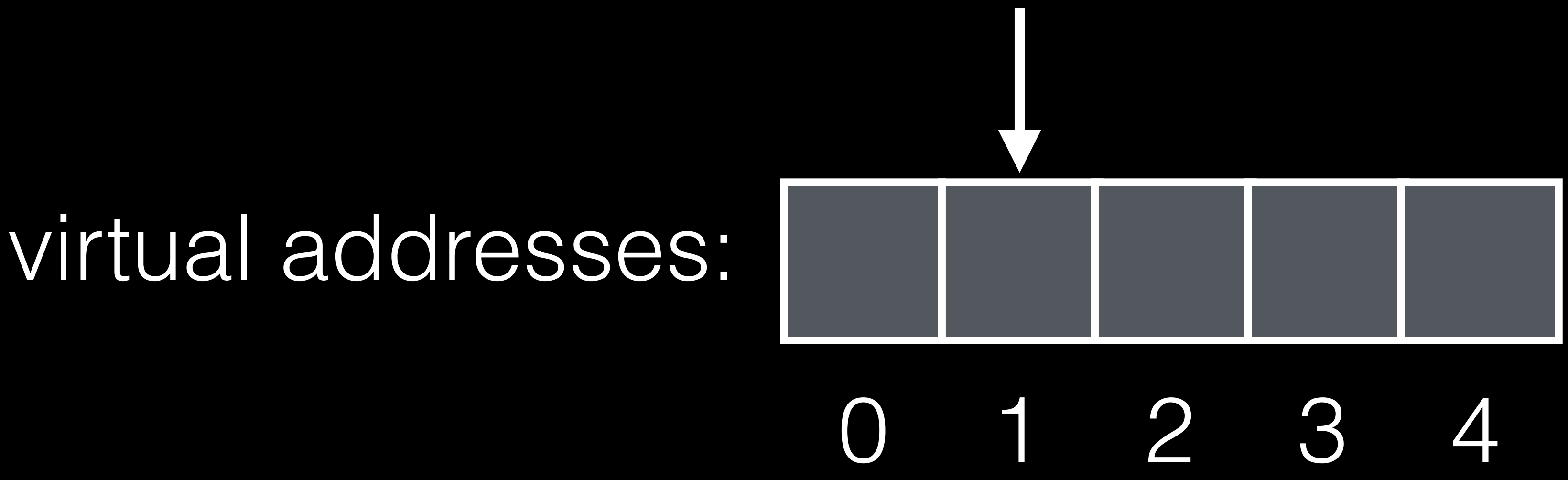
LRU Troubles

virtual addresses:



Valid	Virt	Phys
1	0	?
0		
0		
0		

LRU Troubles



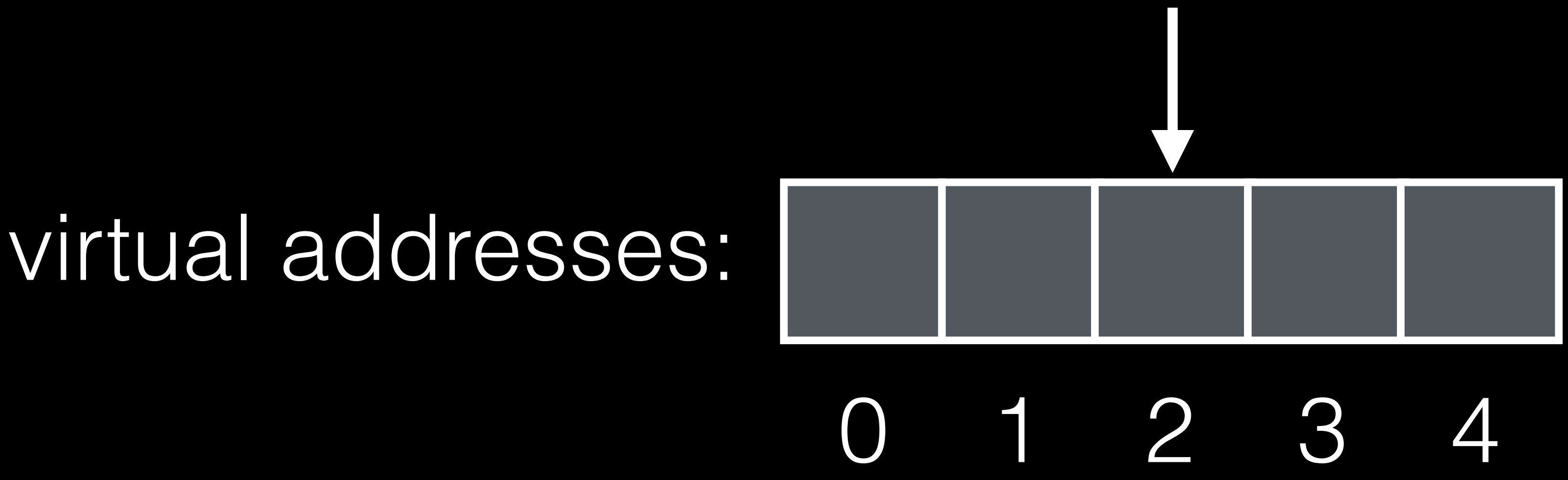
virtual addresses:

0 1 2 3 4

miss!

Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

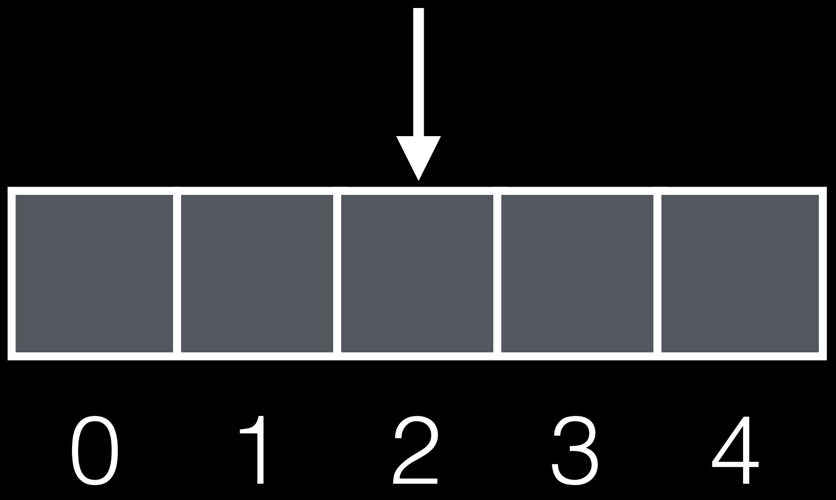
LRU Troubles



Valid	Virt	Phys
1	0	?
1	1	?
0		
0		

LRU Troubles

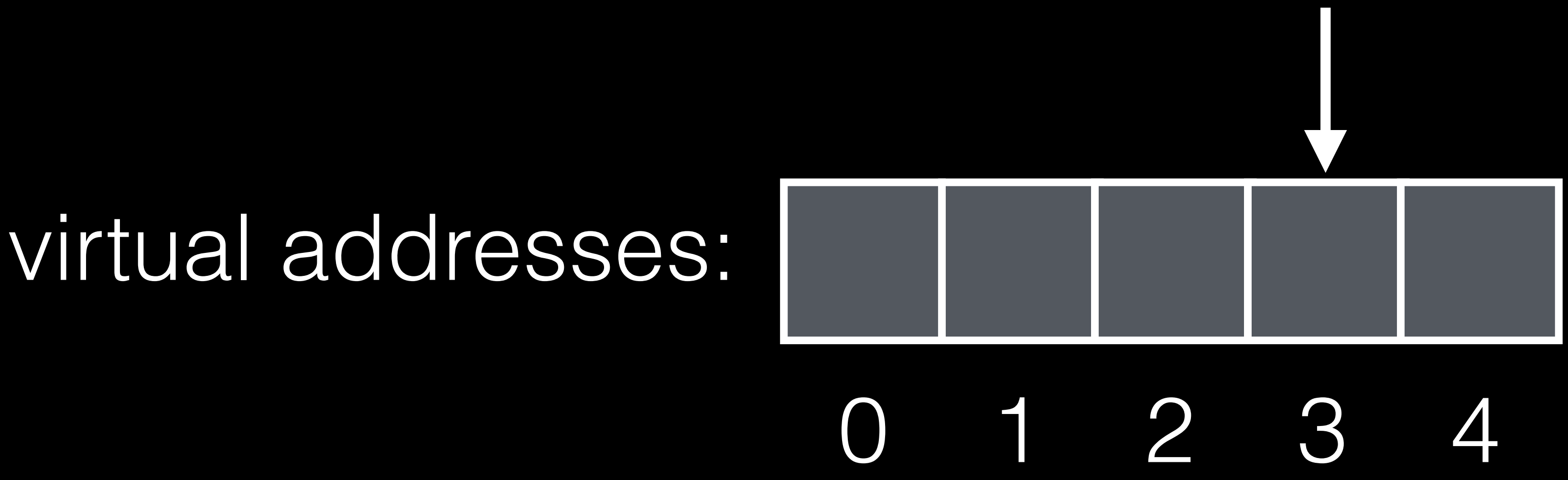
virtual addresses:



miss!

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

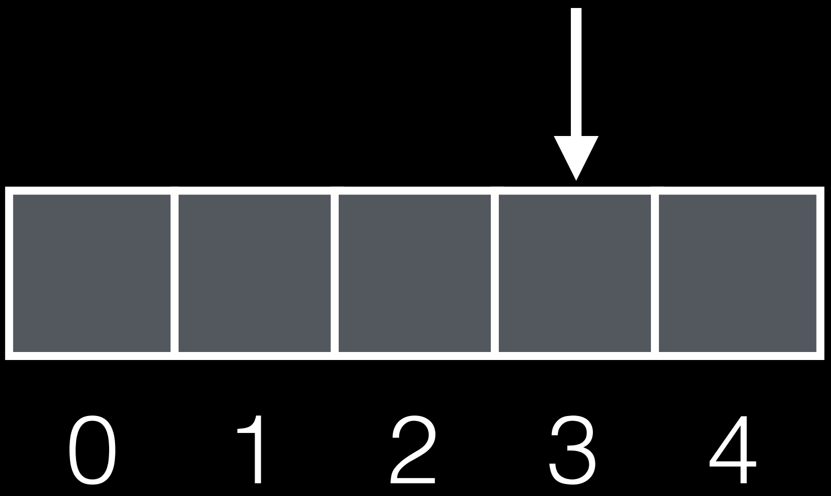
LRU Troubles



Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0		

LRU Troubles

virtual addresses:

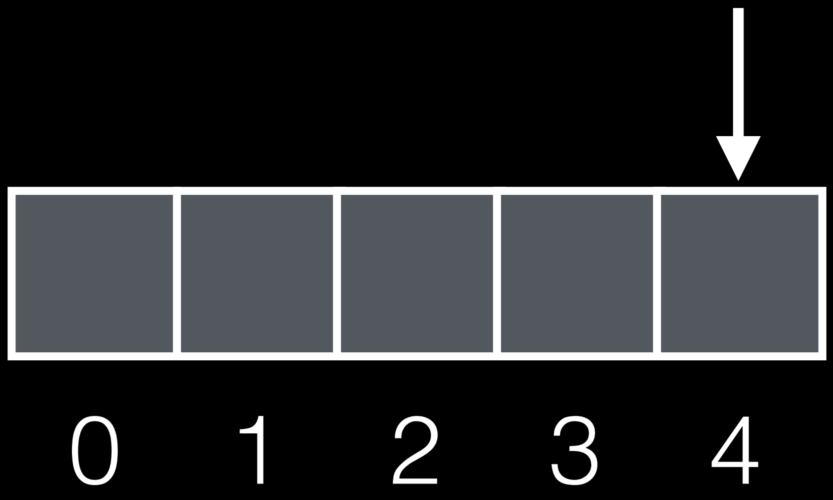


miss!

Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0	3	?

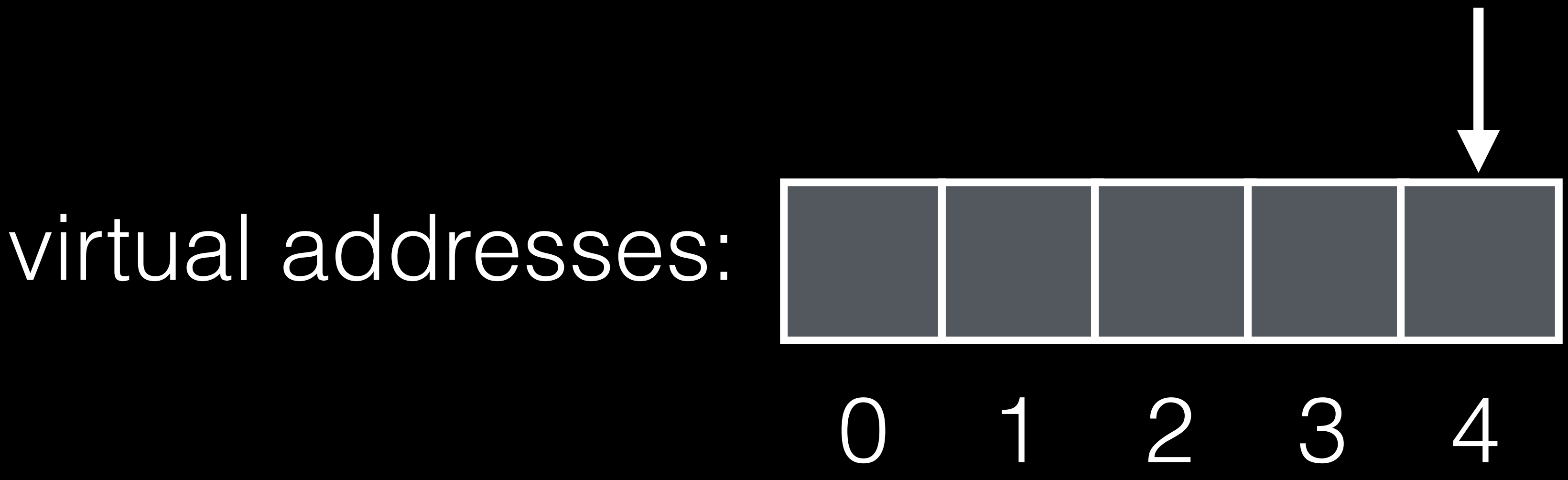
LRU Troubles

virtual addresses:



Valid	Virt	Phys
1	0	?
1	1	?
1	2	?
0	3	?

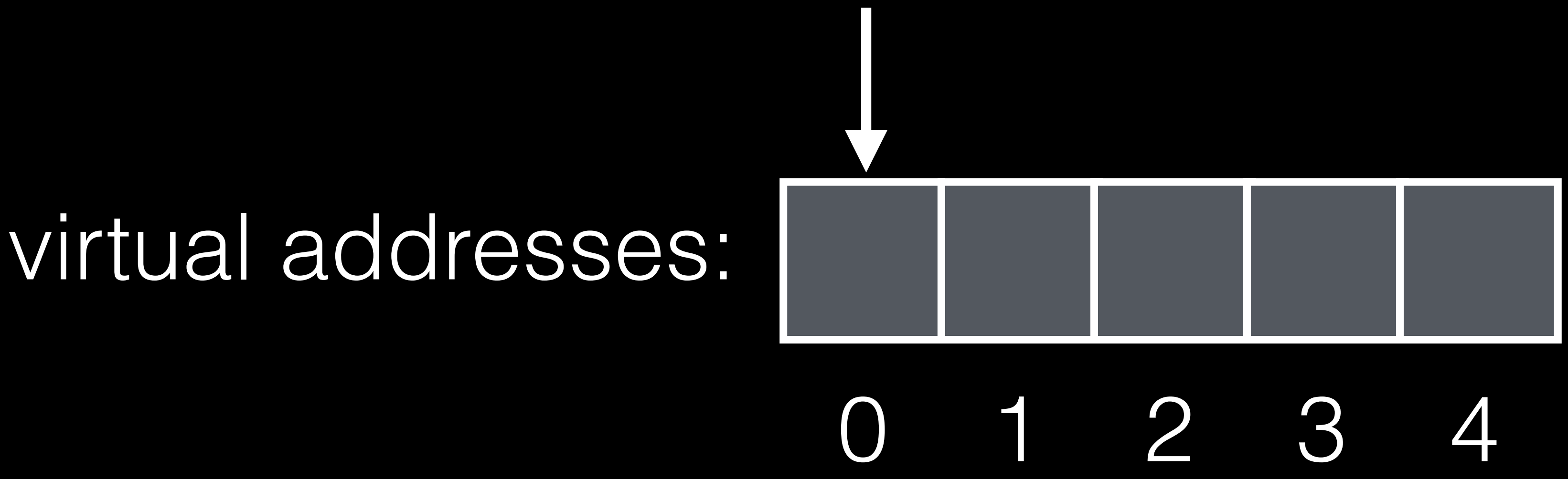
LRU Troubles



Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
0	3	?

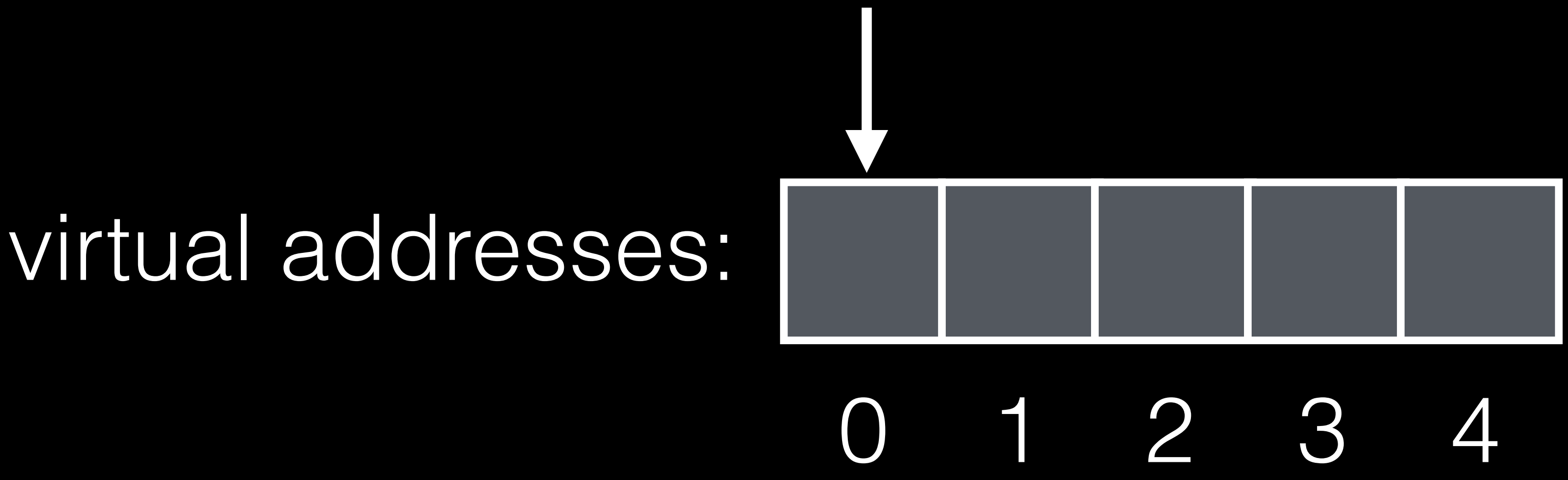
miss!

LRU Troubles



Valid	Virt	Phys
1	4	?
1	1	?
1	2	?
0	3	?

LRU Troubles


virtual addresses:

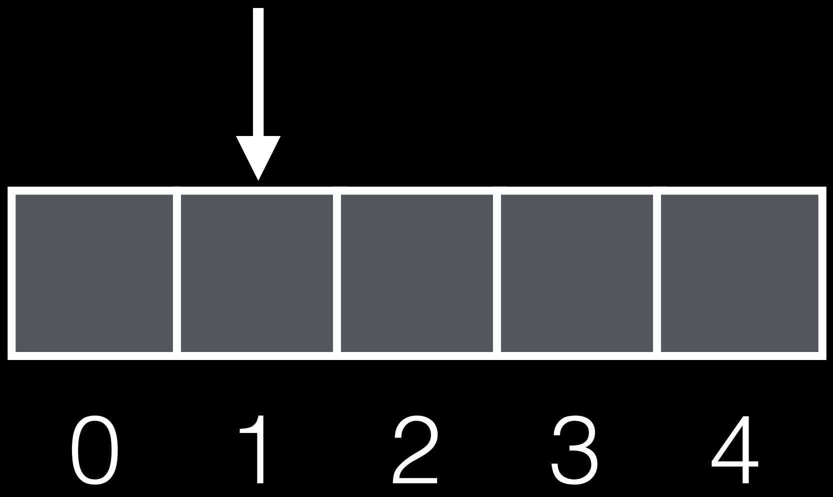
0 1 2 3 4

miss!

Valid	Virt	Phys
1	4	?
1	0	?
1	2	?
0	3	?

LRU Troubles

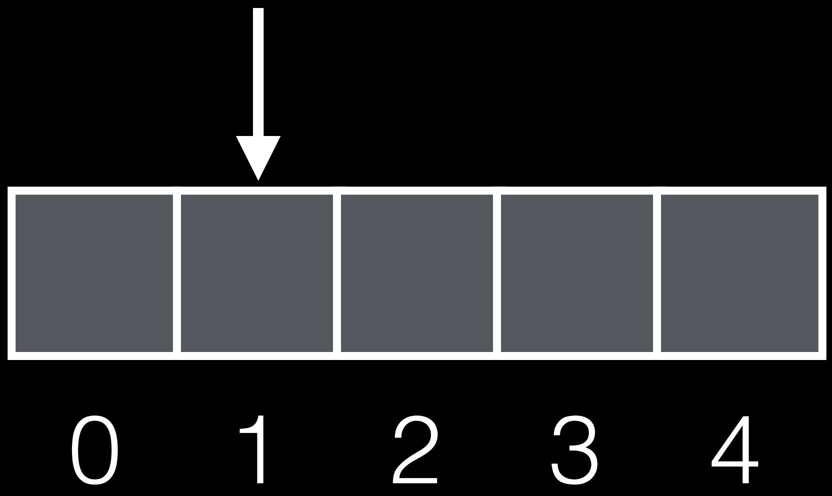
virtual addresses:



Valid	Virt	Phys
1	4	?
1	0	?
1	2	?
0	3	?

LRU Troubles

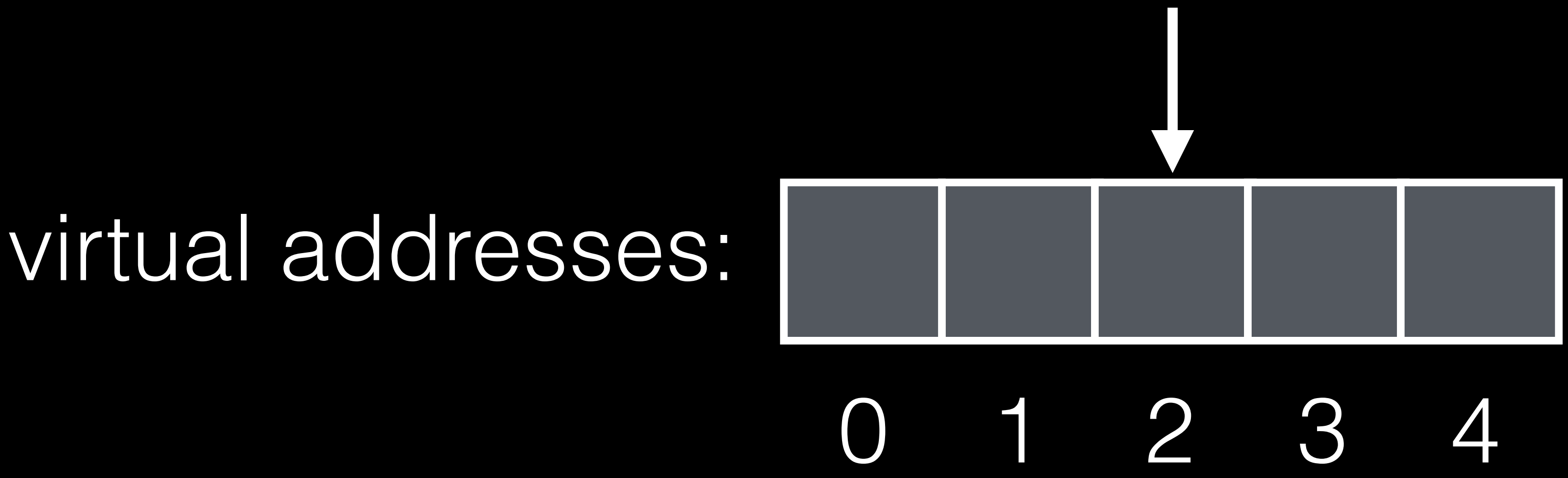
virtual addresses:



miss!

Valid	Virt	Phys
1	4	?
1	0	?
1	1	?
0	3	?

LRU Troubles



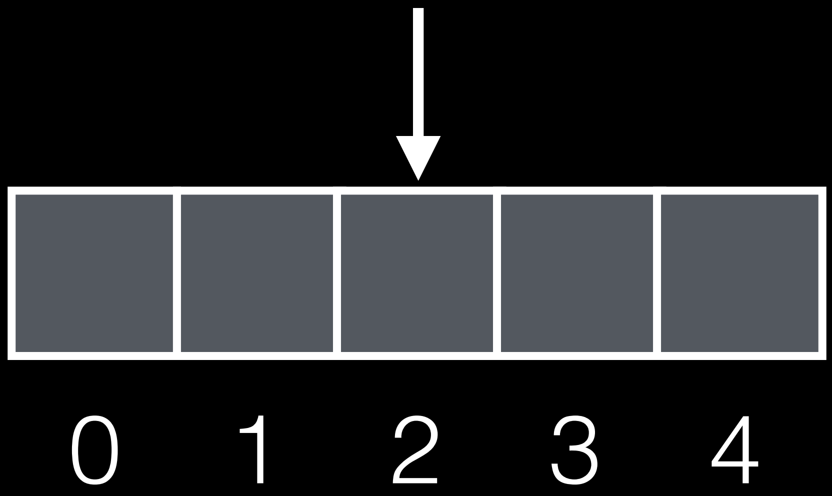
virtual addresses:

0 1 2 3 4

Valid	Virt	Phys
1	4	?
1	0	?
1	1	?
0	3	?

LRU Troubles

virtual addresses:



Valid	Virt	Phys
1	4	?
1	0	?
1	1	?
0	2	?

miss!

A couple policies

LRU: evict least-recently used a TLB slot is needed

Random: randomly choose entries to evict

When is each better?
Sometimes random is better than a “smart” policy!

Outline

What work can we eliminate?

Basic strategy.

Workloads, systems, metrics.

Context switching and security.

Context Switches

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

Context Switches

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

Solutions?

Context Switches

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

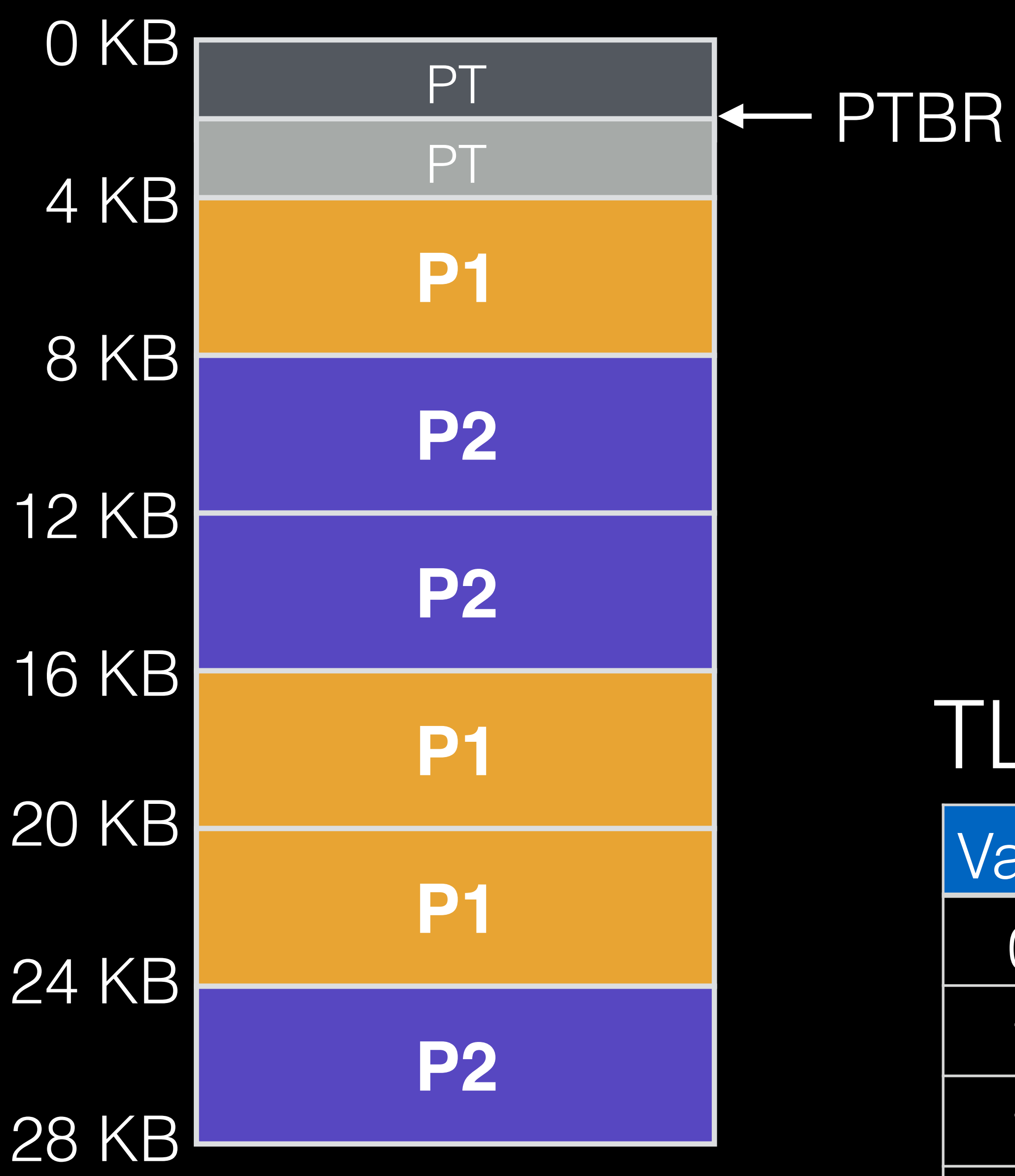
Solutions?

- flush TLB on each switch
- remember which entries are for each process

Address **S**pace **Id**entifier

Tag each TLB entry with an 8-bit ASID

- how many ASIDs to we get?
- why not use PIDs?
- what if there are more PIDs than ASIDs?

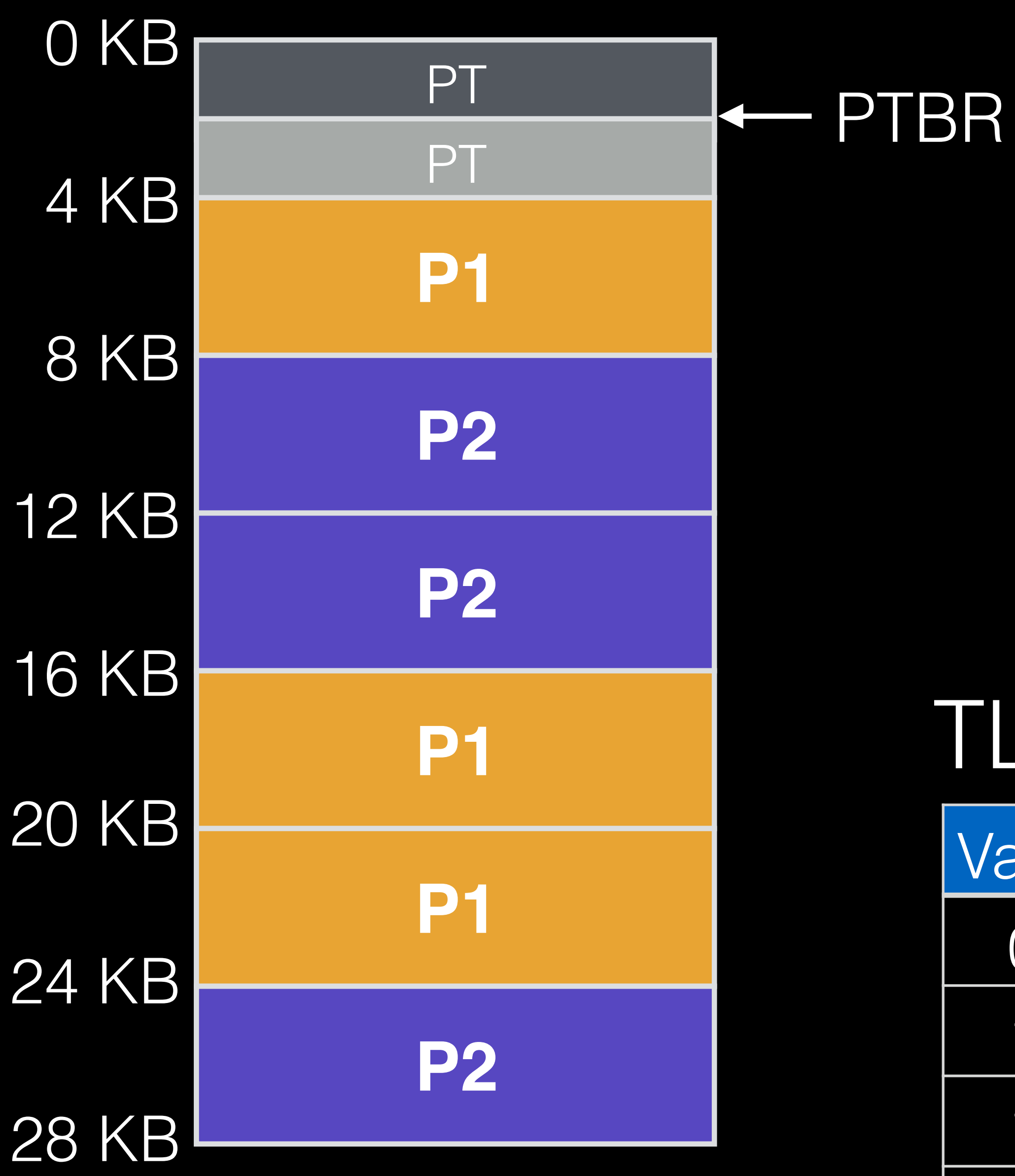


1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual	Physical

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

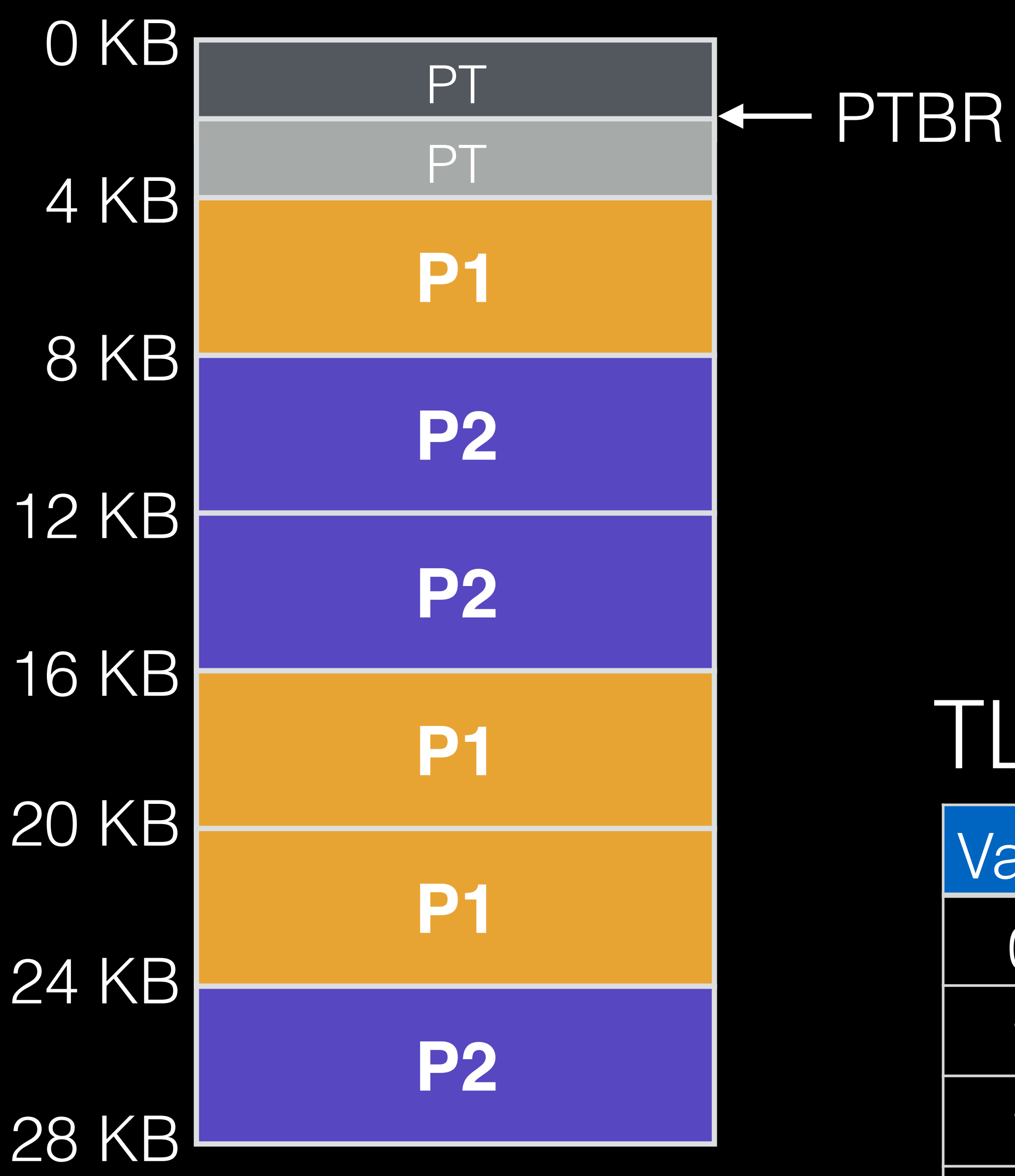


1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual	Physical
load 0x1444	

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11



1	5	4	...

P1 pagetable (ASID 11)

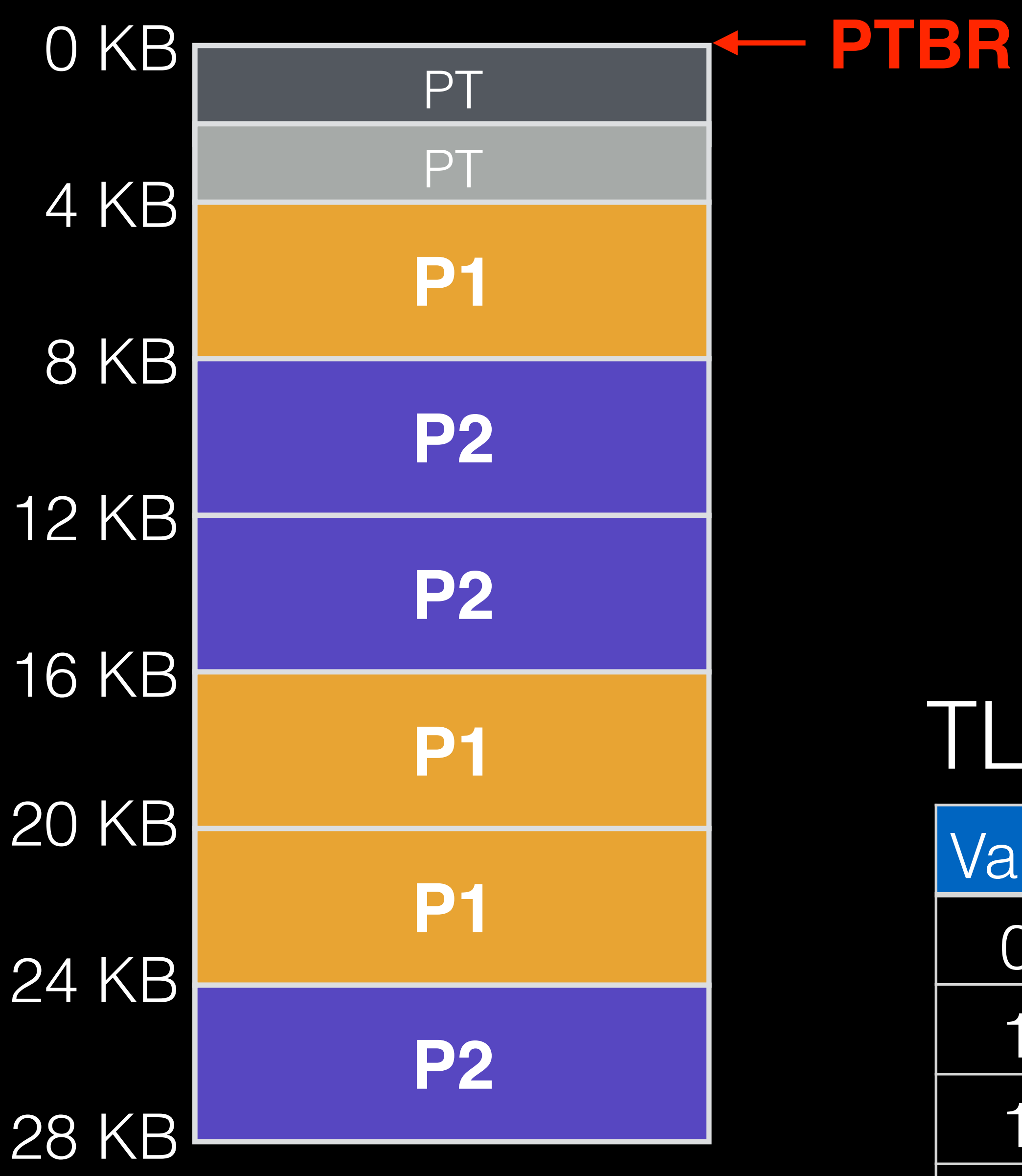
6	2	3	...

P2 pagetable (ASID 12)

Virtual	**Physical**
load 0x1444	load 0x2444

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

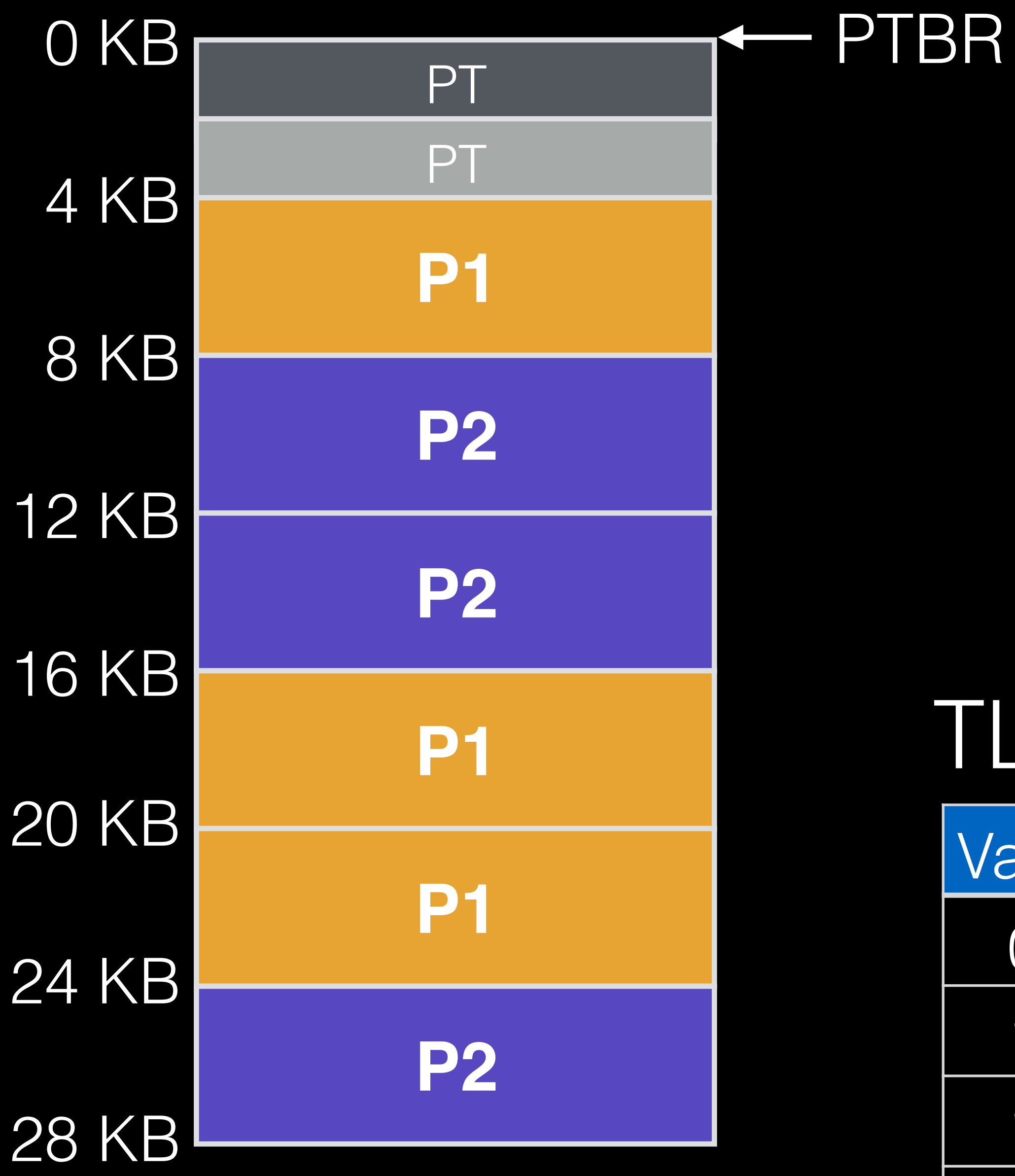


1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual	**Physical**
load 0x1444	load 0x2444

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

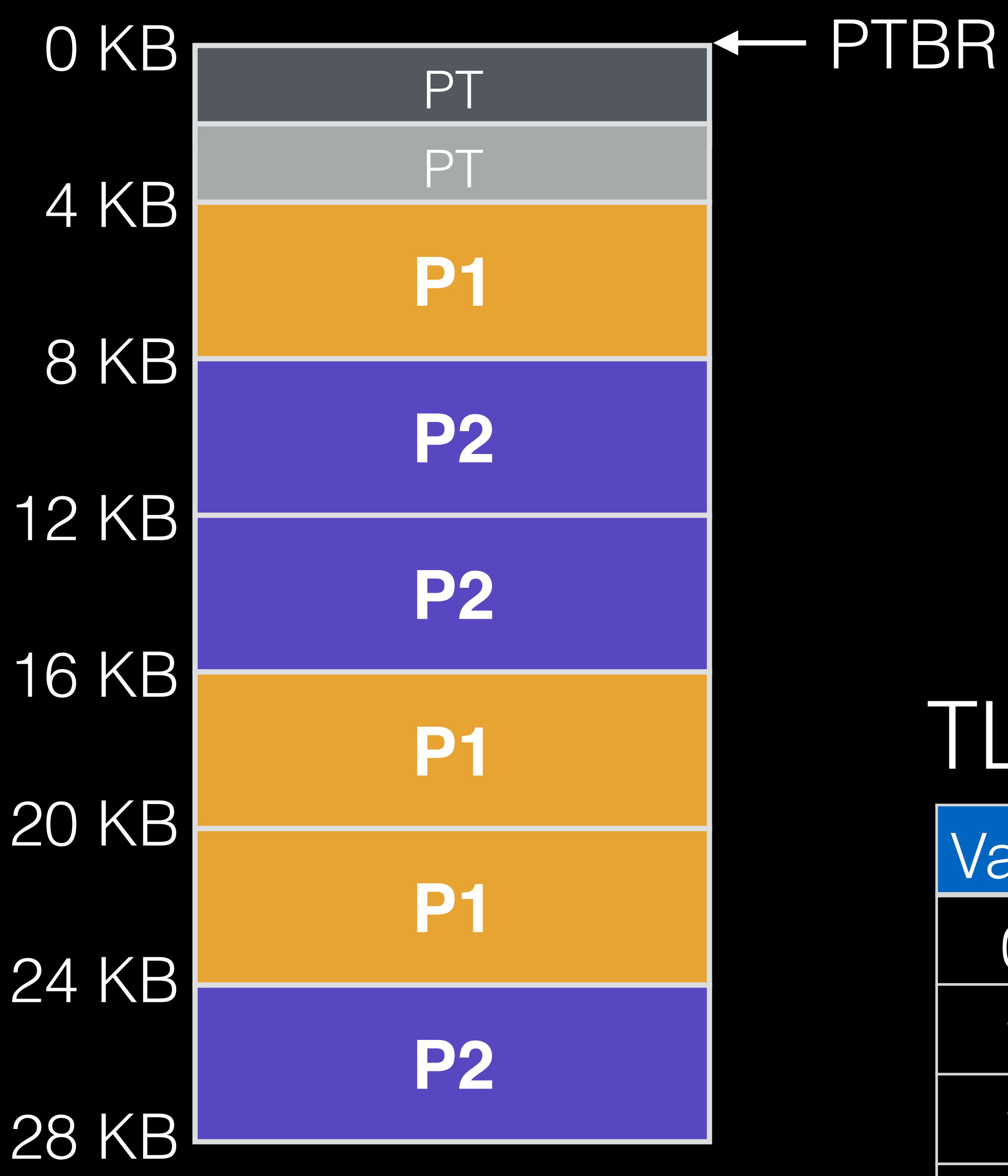


1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual	Physical
load 0x1444	load 0x2444

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

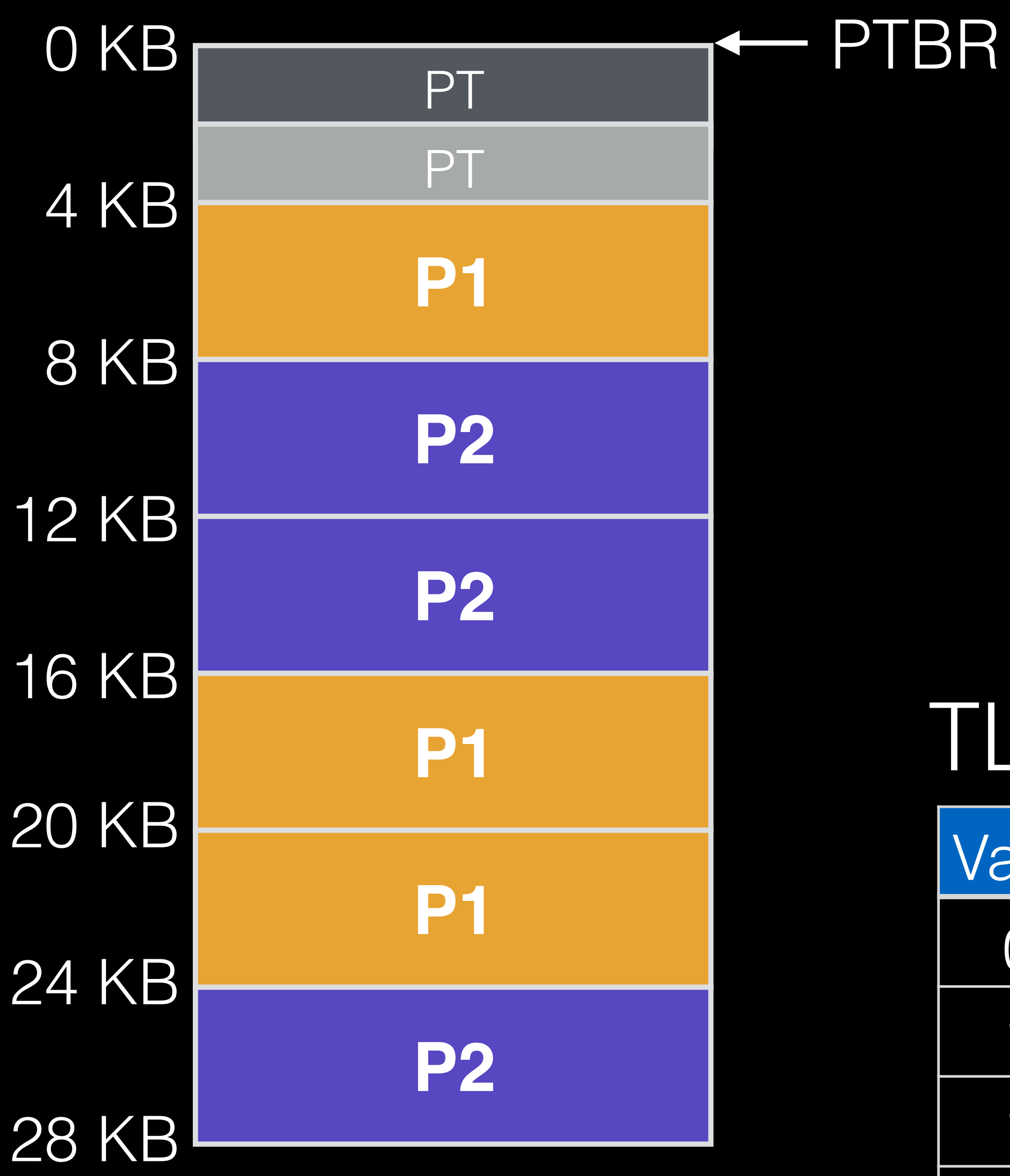


1	5	4	...	P1 pagetable (ASID 11)
6	2	3	...	P2 pagetable (ASID 12)

Virtual	**Physical**
load 0x1444	load 0x2444
load 0x1444	

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11



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

Virtual	**Physical**
load 0x1444	load 0x2444
load 0x1444	load 0x5444

TLB:

Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

Address **S**pace **Id**entifier

Context switches are expensive.

Even with ASID, other processes “pollute” the TLB.

Who changes the TLB?

H/W or **OS**?

Who changes the TLB?

H/W or **OS**?

H/W: CPU must know where pagetables are
- CR3 on x86
- pagetable structure not flexible
- “walk” the pagetable

OS: CPU traps into OS upon TLB miss
- how to avoid double traps?
- more modern

Security

Modifying TLB entries is privileged
 - otherwise what could you do?

Need same protection bits in TLB as pagetable
 - rwx

Measurement Demo

(if enough time)