

CS 423

Operating System Design:

Paging and TLBs

Feb 12

Ram Kesavan

Logistics

MP0: Will release grades by 2/17

MP1: Due 2/25 mid-night

Will post a testing document next week

AI policy for MPs

Use Piazza & TA office hours for MP help

Ok to post publicly if not including large amounts of your code

Or post privately, and let staff decide to make it public

AGENDA / LEARNING OUTCOMES

Memory virtualization

Paging details

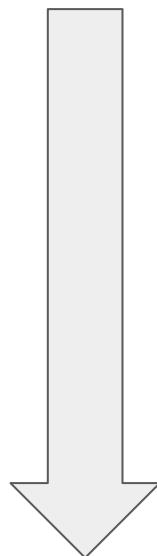
Challenges with implementing paging

Performance & TLBs

RECAP

Mechanisms for Virtualization

1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation



Limited practicality, has many problems

More practical, still has some problems

Segmentation

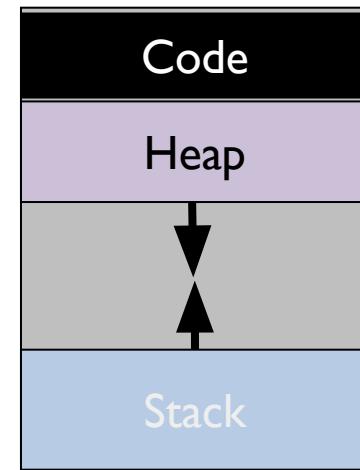
Divide address space into logical segments

- Each segment corresponds to logical entity in address space
(code, stack, heap)

Each segment has separate base + bounds register

Each segment can independently:

1. Be placed in physical memory
2. Grow and shrink
3. Be protected (rwx permissions)



0

$2^n - 1$

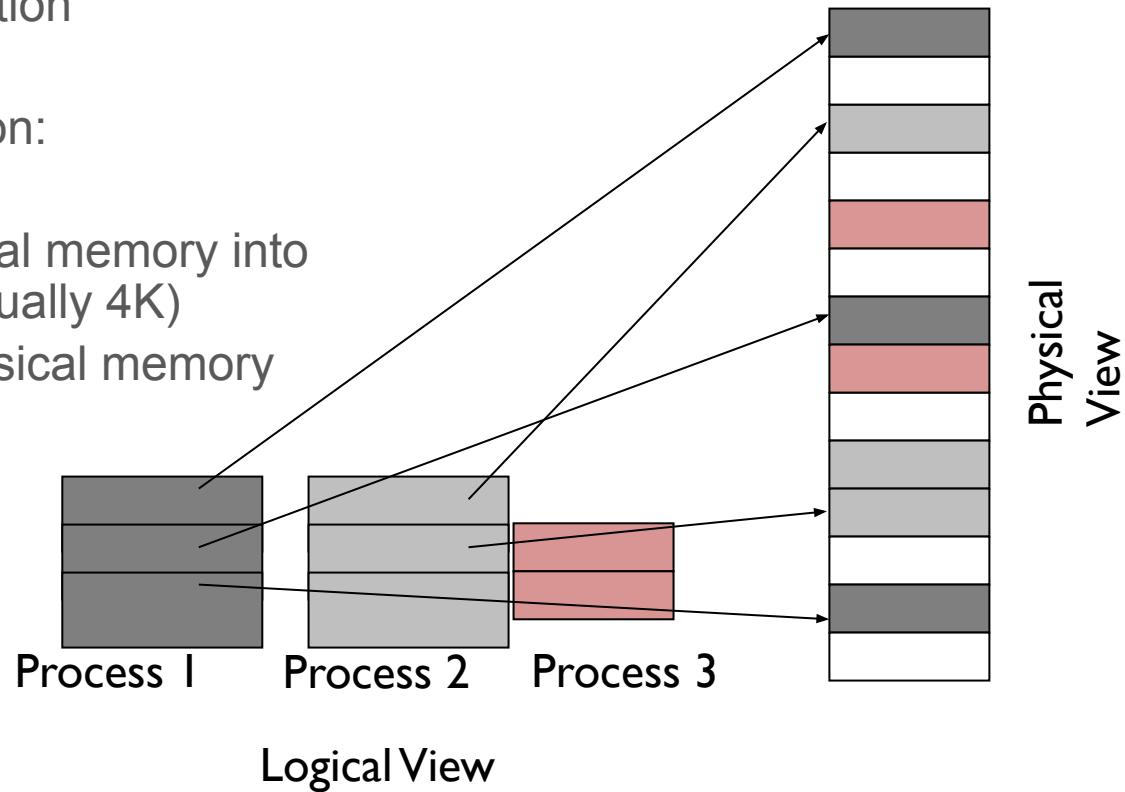
Paging

Goal: Eliminate requirement that segment is contiguous
Eliminate external fragmentation

Idea to avoid external fragmentation:

Divide address spaces and physical memory into fixed-sized pages/frames (usually 4K)

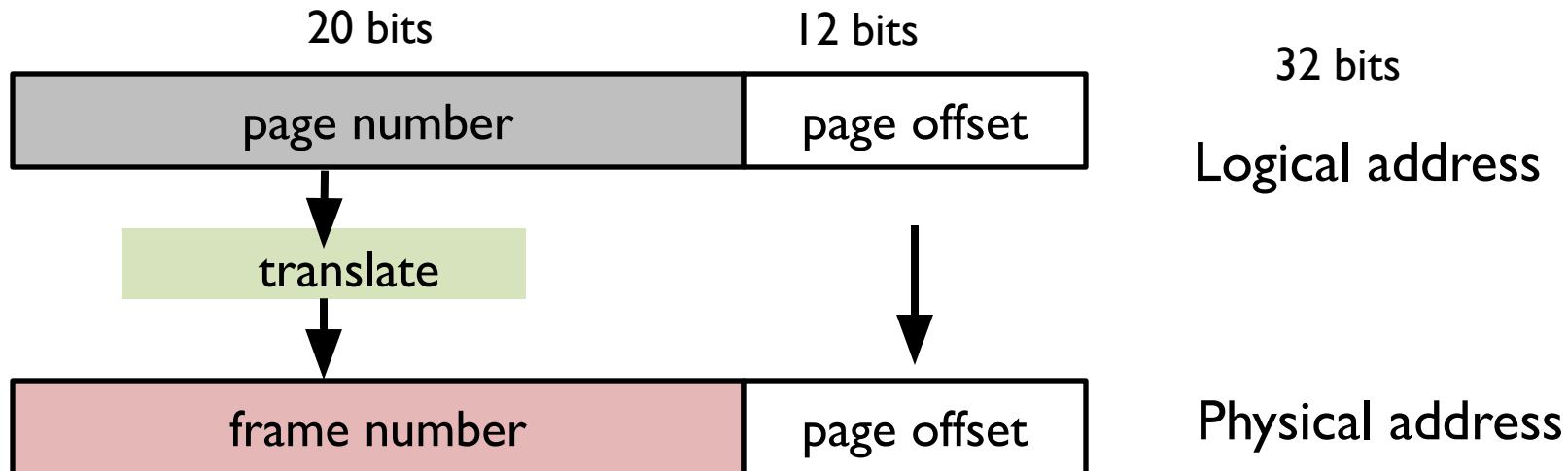
Can place pages anywhere in physical memory



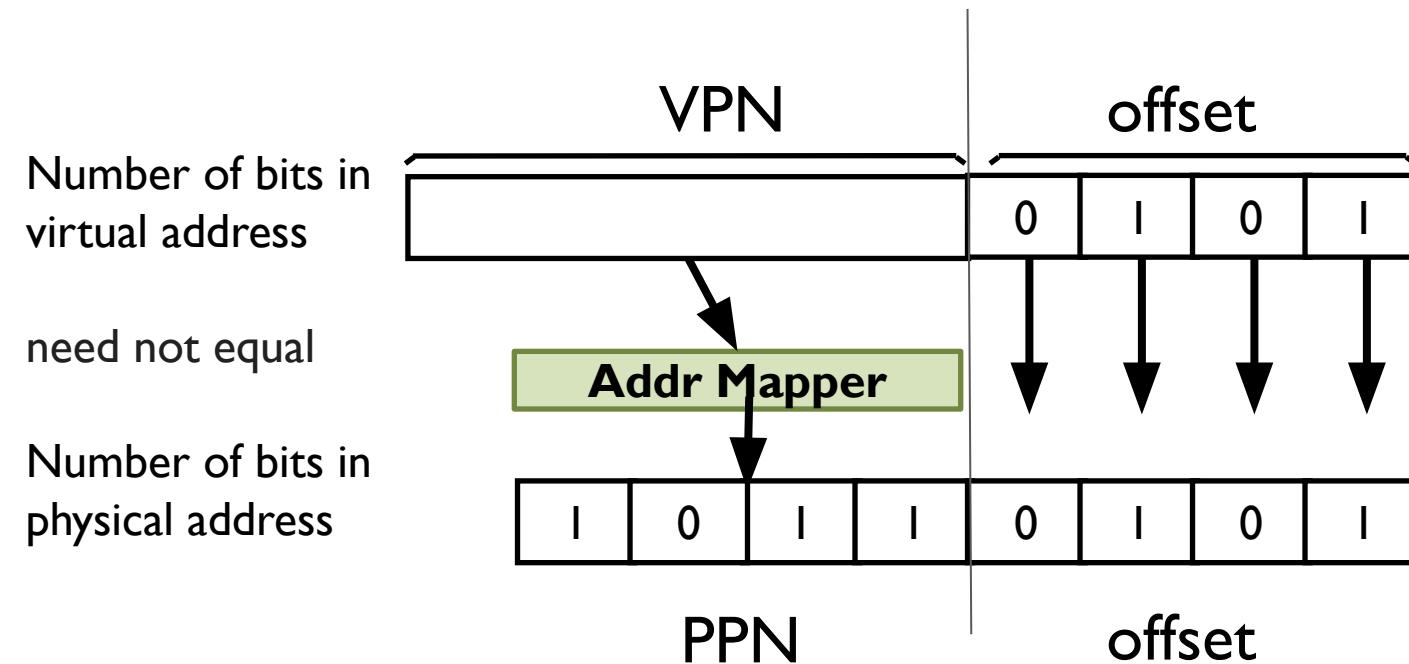
Translation of Page Addresses

How to translate logical address to physical address?

- High-order bits of address designate page number
- Low-order bits of address designate offset within page



VIRTUAL -> PHYSICAL PAGE MAPPING



How should OS translate VPN to PPN/PFN?

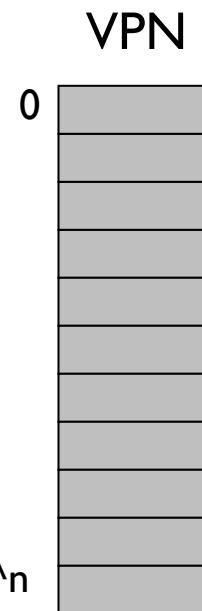
Linear Page Table

What is a good data structure ?

Simple solution: Linear page table aka array

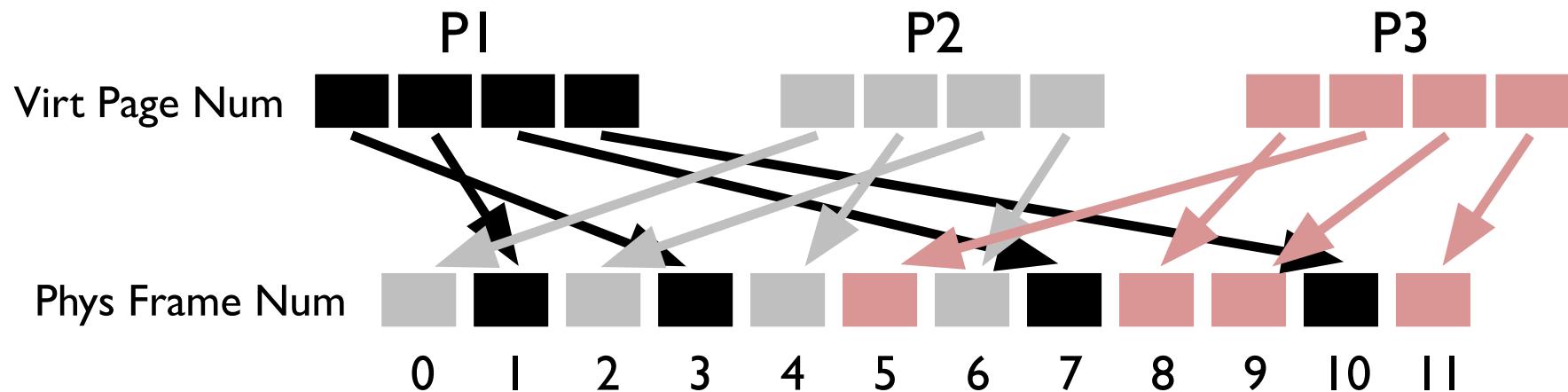
A: accessed, P: present, D: dirty

A Single PTE:



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																	G	PAT	D	A	PCD	PWT	U/S	R/W	P						

FILL IN THIS PAGE TABLE



Per-process
Page Tables

HOW BIG IS A PAGE TABLE?

Assume 32-bit address

Assume 4KB pages => 12 bits for offset
20 bits for VPN

Assume 4 byte page table entries (PTE)

How large is PT for each process? $2^{20} * 4 = 4 \text{ MB}$

Implications? For 1K processes, we need 4GB of PTs!

END OF RECAP

WHERE TO STORE PAGE TABLES?

Obvious: store each page table in memory

For now: let's imagine we have sufficient memory for this!

Hardware finds page table base with register

What is this register in x86?

On a context-switch:

Store contents of this register in PCB of old process

Load contents of this register with new process' page table base

MEMORY ACCESSES WITH PAGING

14 bit virtual address

0x0010: movl 0x1100, %edi

Assume PT is at phys addr 0x50000

Assume each PTE is 4 bytes

Assume 4KB pages

i.e., 12 bits for offset

How many memory accesses?

To which physical addresses?

Chat with neighbors for 2 mins...

Simplified view
of page table

2
0
80
99

MEMORY ACCESSES WITH PAGING

14 bit virtual address

0x0010: movl 0x1100, %edi

Assume PT is at phys addr 0x50000

Assume each PTE is 4 bytes

Assume 4KB pages

i.e., 12 bits for offset

Simplified view
of page table

2
0
80
99

Disadvantages of Paging

Additional memory reference to page table entry

- Very inefficient, so...
- ...page table must be stored in memory
- MMU stores only base address of page table
 - Solution: TLBs

Space needed for page tables is too large

- Simple page table: Requires PTE for each virtual page number
PTE needed even if page not allocated
- Page tables must be contiguously allocated
 - Solution: paging the page tables!

PAGE 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. use **PA** to read from memory

Which steps are expensive?

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' and 'sum'

What virtual addresses?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

Assume 4-byte PTE, what can you infer?

What is the PT base address?

What is the PPN for VPN 3?

What physical addresses?

load 0x100C

load 0x7000

load 0x100C

load 0x7004

load 0x100C

load 0x7008

load 0x100C

load 0x700C

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' and 'sum'

What virtual addresses?

load 0x3000

load 0x3004

load 0x3008

load 0x300C

What physical addresses?

load 0x100C

load 0x7000

load 0x100C

load 0x7004

load 0x100C

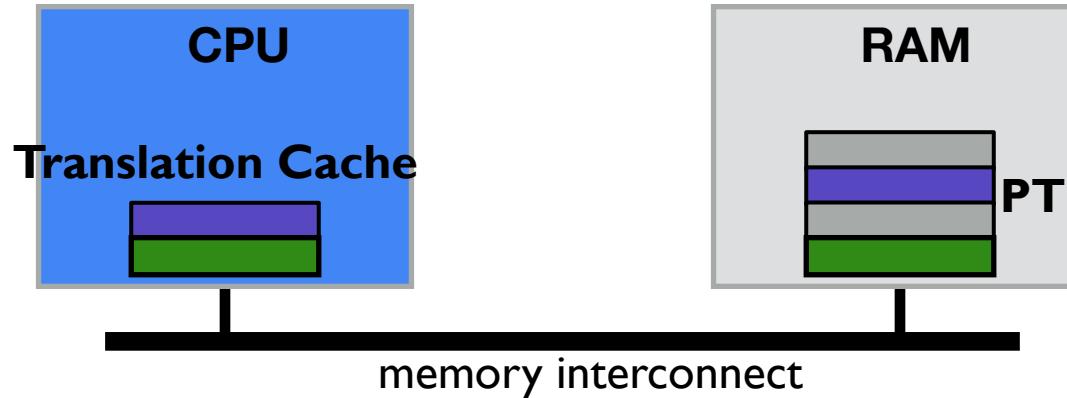
load 0x7008

load 0x100C

load 0x700C

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

STRATEGY: CACHE PAGE TRANSLATIONS



TLB: TRANSLATION LOOKASIDE BUFFER
Each core has its own TLB

TLB ORGANIZATION

TLB Entry

Tag (virtual page number)	Physical frame number (from PTE)	other bits
---------------------------	----------------------------------	------------

TLB housed in the MMU

Fully Associative

A given translation can be anywhere in the TLB

Hardware searches the entire TLB in parallel

Takes only 0.5 to 1 clock cycle to search

Typical size of TLB: 16 to 1024 entries

Other bits: valid, rwx (protection), dirty

ARRAY ITERATOR (W/ TLB)

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

Assume 'a' starts at 0x1000.
Ignore load & stores of 'i' and 'sum'

Assume following virtual address stream:
load 0x1000

load 0x1004

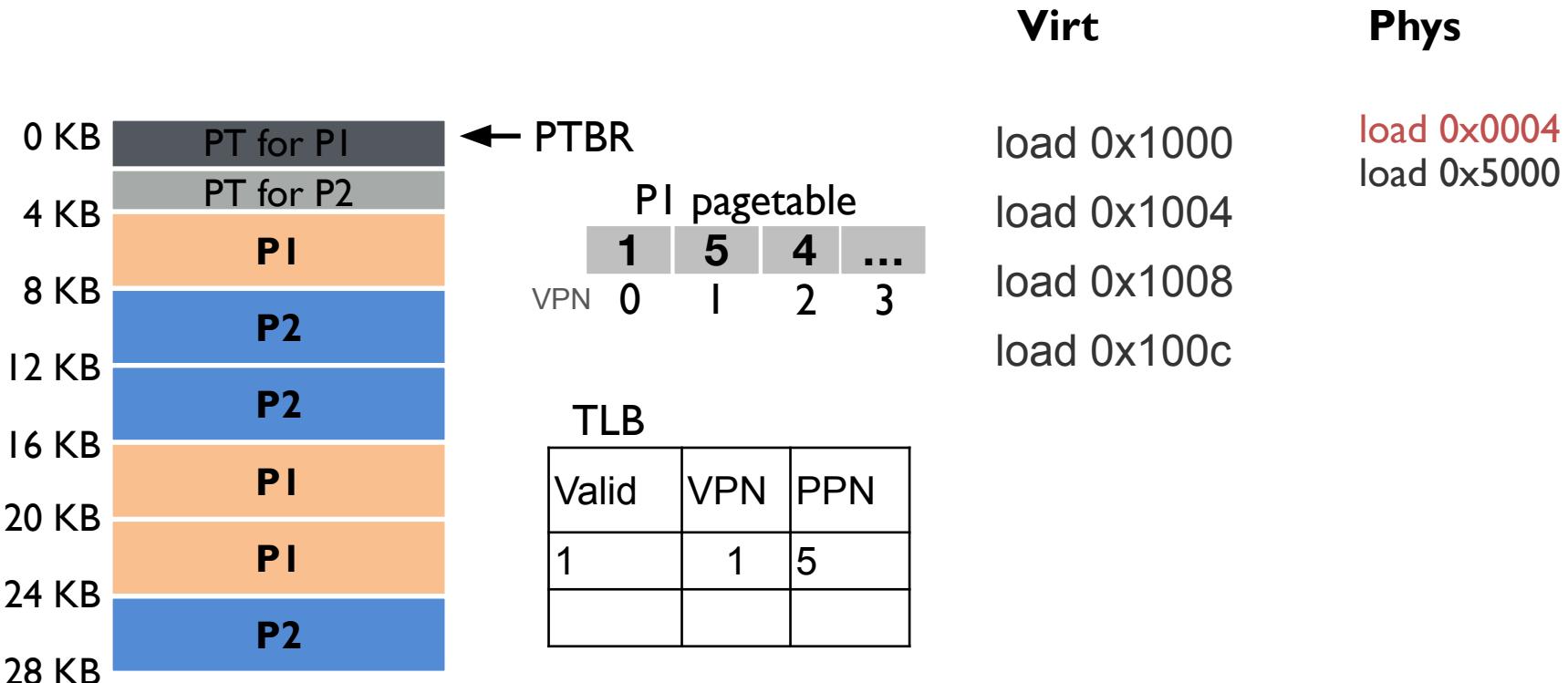
load 0x1008

load 0x100C

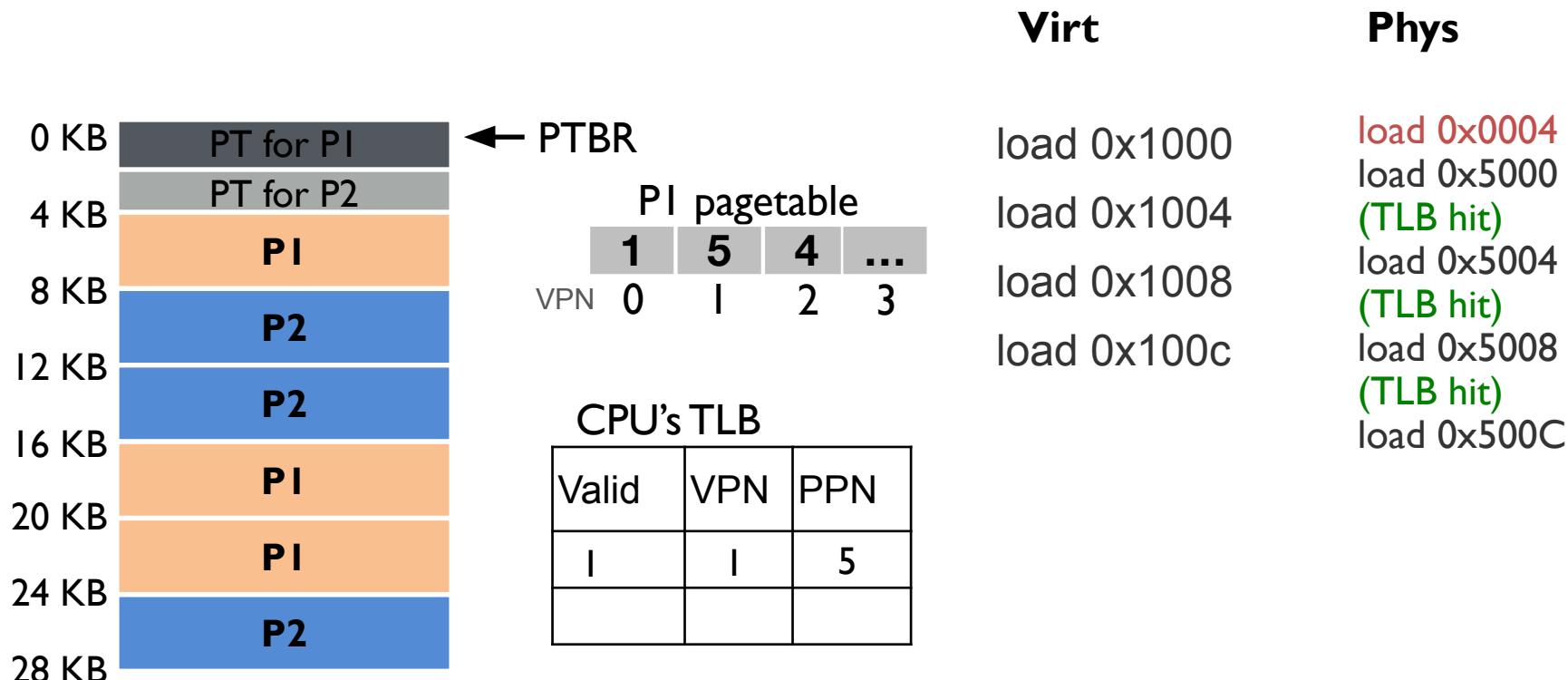
...

What will TLB behavior look like?

TLB Accesses: SEQUENTIAL Example



TLB Accesses: SEQUENTIAL Example



PERFORMANCE OF TLB?

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

Would hit rate get better or worse with smaller pages?

Would hit rate get better or worse with more iterations?

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

Let N = 2048

TLB lookups? number of accesses to a = 2048

TLB misses?

$$\begin{aligned}&= \text{number of unique pages accessed} \\&= 2048 / (\text{elements of 'a' per 4K page}) \\&= 2K / (4K / \text{sizeof(int)}) = 2K / 1K \\&= 2\end{aligned}$$

Miss rate? = 2/2048 < 0.1%

Hit rate? (1 - miss rate) > 99.9%

TLB PERFORMANCE

How can system improve 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

WORKLOAD ACCESS PATTERNS

Workload A

```
int sum = 0;  
for (i=0; i<1024; i++) {  
    sum += a[i];  
}
```

Sequential array access is almost always better for TLB hit rate.

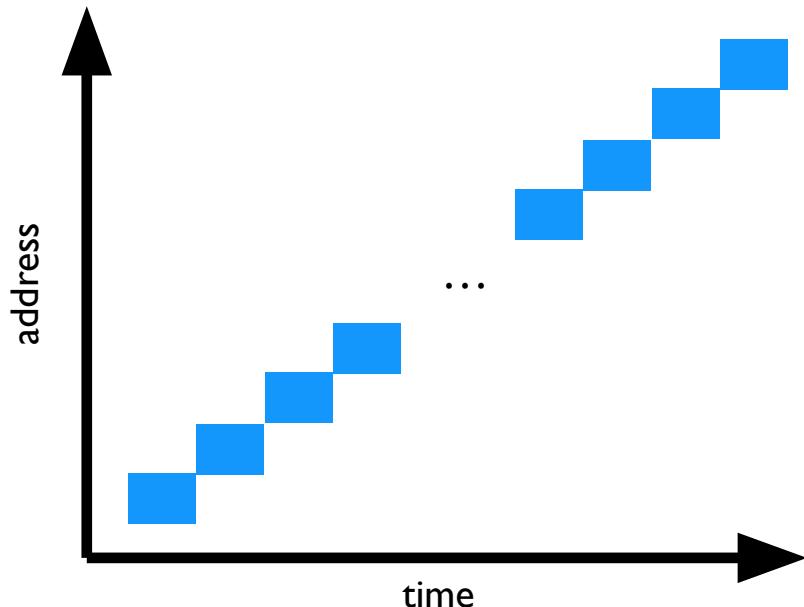
Workload B

```
N = 1024;  
srand(1234);  
for (i=0; i<10; i++) {  
    something(a[rand() % N]);  
}  
srand(1234);  
for (i=0; i<10; i++) {  
    something(a[rand() % N]);  
}
```

Workload ACCESS PATTERNS

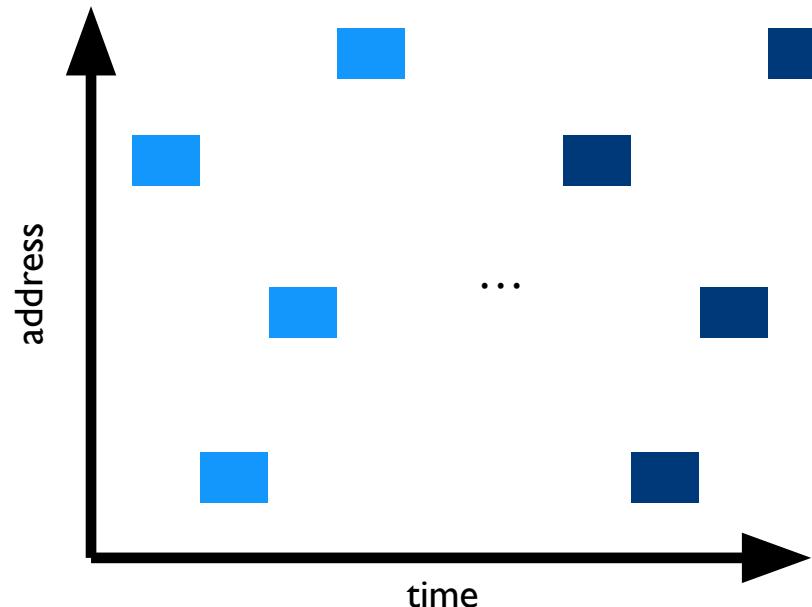
Spatial Locality

Sequential Accesses



Temporal Locality

Repeated Random Accesses



WORKLOAD LOCALITY

Spatial Locality: future access to neighboring data

Temporal Locality: future access to the same data

What TLB characteristics are best for each type?

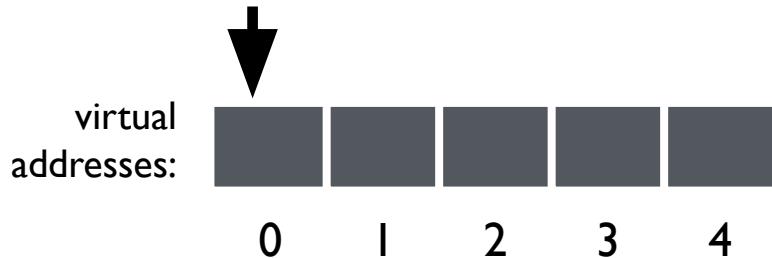
Spatial:

- Access same page repeatedly; need same vpn -> ppn translation
- Same TLB entry re-used immediately

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?

LRU TROUBLES



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

Workload repeatedly accesses same offset (0x01) across 5 pages (strided access), but only 4 TLB entries

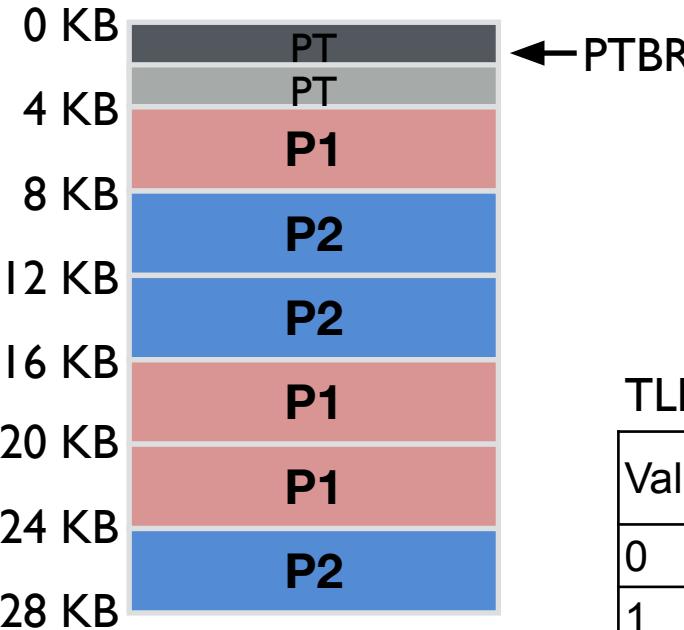
What is the TLB miss rate?

CONTEXT SWITCHES

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

- I. Flush entire TLB on each context switch (`void flush_tlb_all(void) in Linux`)
Lose all recently cached translations
2. Track which entries are for which process
 - Address Space Identifier
 - Tag each TLB entry with an 8-bit (or 16-bit) ASID
 - Must match ASID on lookups

TLB Example with ASID



Virtual		Physical	
		load 0x1444	ASID: 12
		load 0x1444	ASID: 11
TLB:			
Valid	Virt	Phys	ASID
0	1	9	11
1	1	5	11
1	1	2	12
1	0	1	11

UNMAPPING MEMORY

A process may unmap certain regions of virtual memory (e.g., munmap)

OS must ensure that translations for that unmapped region is removed from TLB

Several ways to do this:

- I. Flush all tlb entries for a process – `void flush_tlb_mm(struct mm_struct *mm)`

Inefficient - but useful for fork + exec

2. Flush specific range: `void flush_tlb_range(vm_area_struct *vma, unsigned start, unsigned end)`

TLB Performance

Context switches are expensive

Based on access pattern: Speculative prefetching of TLB entries

Can detect sequential and strided access

A CPU architecture can have multiple TLBs

- A TLB for data, and a TLB for instructions
- A TLB for regular pages, and a TLB for “super pages”

WHAT HAPPENS ON A TLB MISS?

MMU loads the PTE from memory

Inserts into TLB

Retries the instruction (and the memory access)

If H/W handles TLB Miss

CPU must know location page table

- CR3 register on x86
- Page table format fixed and agreed upon between HW and OS
- H/W “walks” the page table and fills TLB

If OS handles TLB Miss: “software-managed TLB”

- CPU forces OS to (raise privilege level and) jump to a trap handler
- Handler interprets page tables and locates PTE, inserts into TLB
- Return from trap to retry the instruction

How to replace TLB entries?

Standard policy problem in any caching solution

LRU: Evict least-recently-used TLB entry

Needs per-entry bits to track that

Another simple option: random

TLB Summary

Paging is great, but accessing page tables for every memory access is slow
Cache recent page translations in TLB

- MMU performs TLB lookup on every memory access

TLB performance depends strongly on workload

- Sequential workloads perform well
- Workloads with temporal locality can perform well

TLB increases cost of context switches

- Flush TLB on every context switch
- Add ASID to every TLB entry

TLB misses handling: hardware or OS

Depends on the system

NEXT CLASS

But, but, but...what about the fact that page tables are massive?