CS 423
Operating System Design: Paging and TLBs
02/14

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

MP0 done. Will release grades by Monday.

MP1 due 2/28

TAs will be available today after the lecture here for MP1 help

AGENDA / LEARNING OUTCOMES

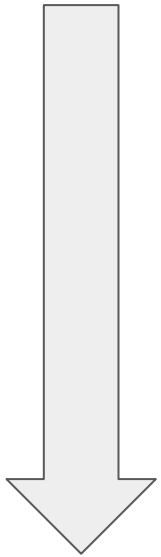
Memory virtualization

- What is paging and how does it work?
- What are some of the challenges in implementing paging?
- Caching address translations using 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

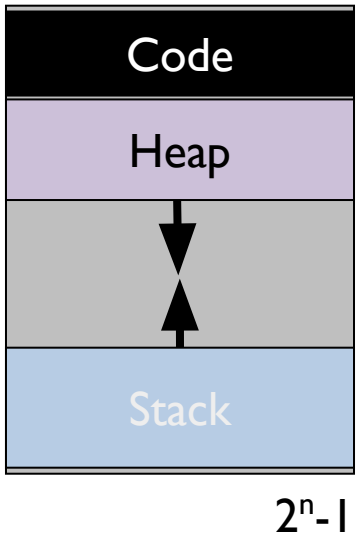
Divide address space into logical segments

- 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 (read/write/exec)



Advantages of Segmentation

Enables sparse allocation of address space

Supports dynamic relocation of each segment

Enables sharing of selected segments (two process with same code)

Different protection for different segments

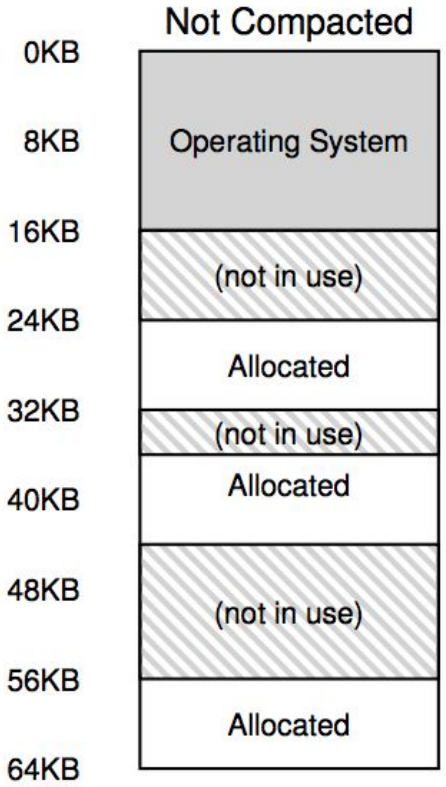
- Example: no write permissions for code segment

Disadvantages of Segmentation

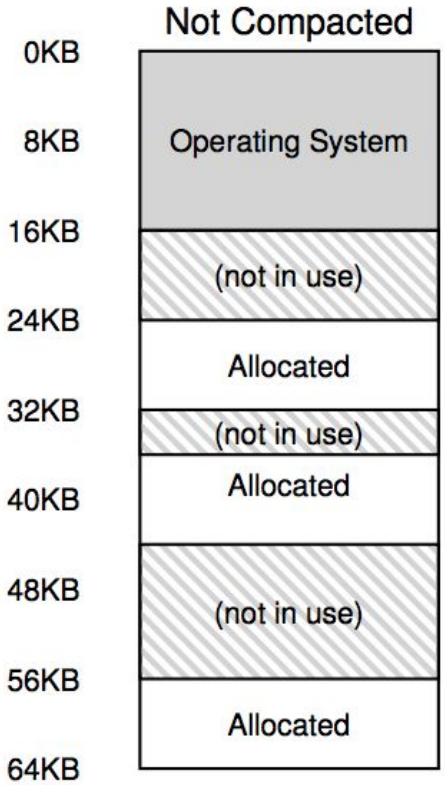
Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation



FRAGMENTATION

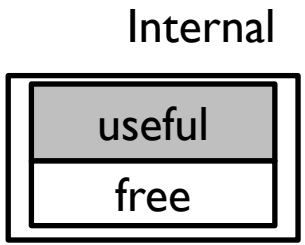


Definition: Free memory that can't be usefully allocated

Types of fragmentation

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

Internal: Visible to requester



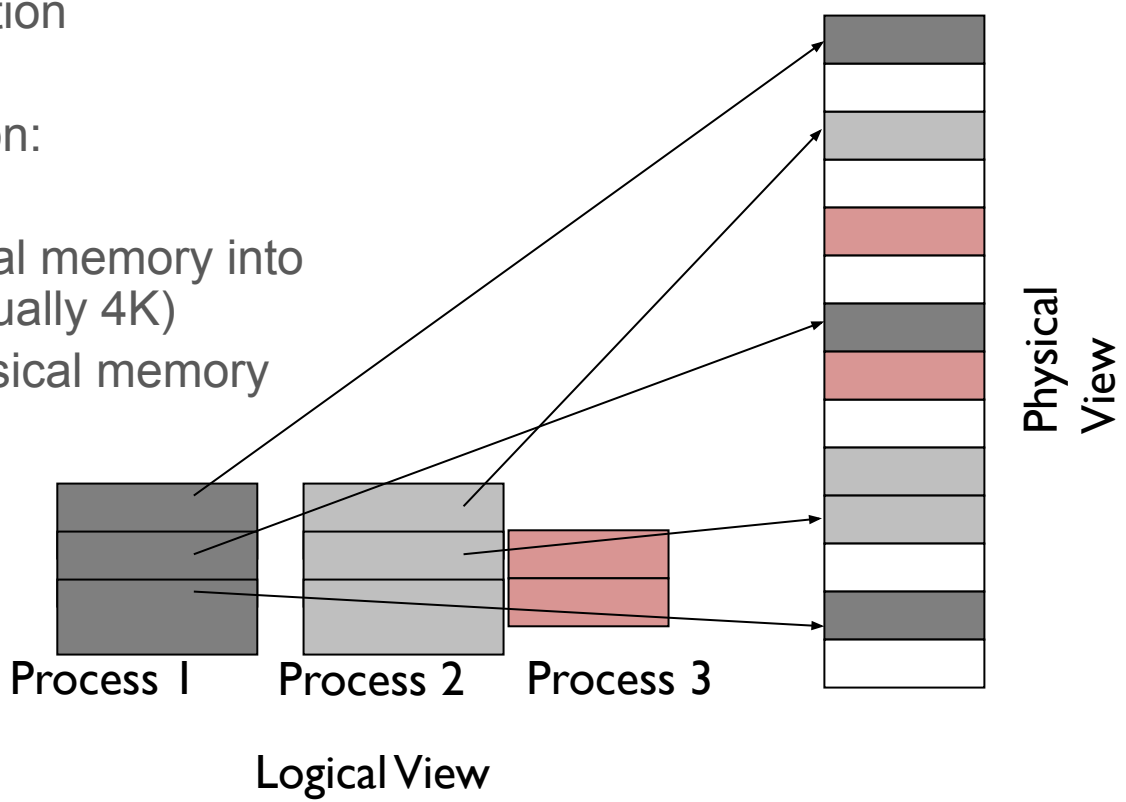
PAGING
(more modern systems including Linux use this)

Paging

Goal: Eliminate requirement that segment is contiguous
Eliminate external fragmentation

Idea to avoid external fragmentation:

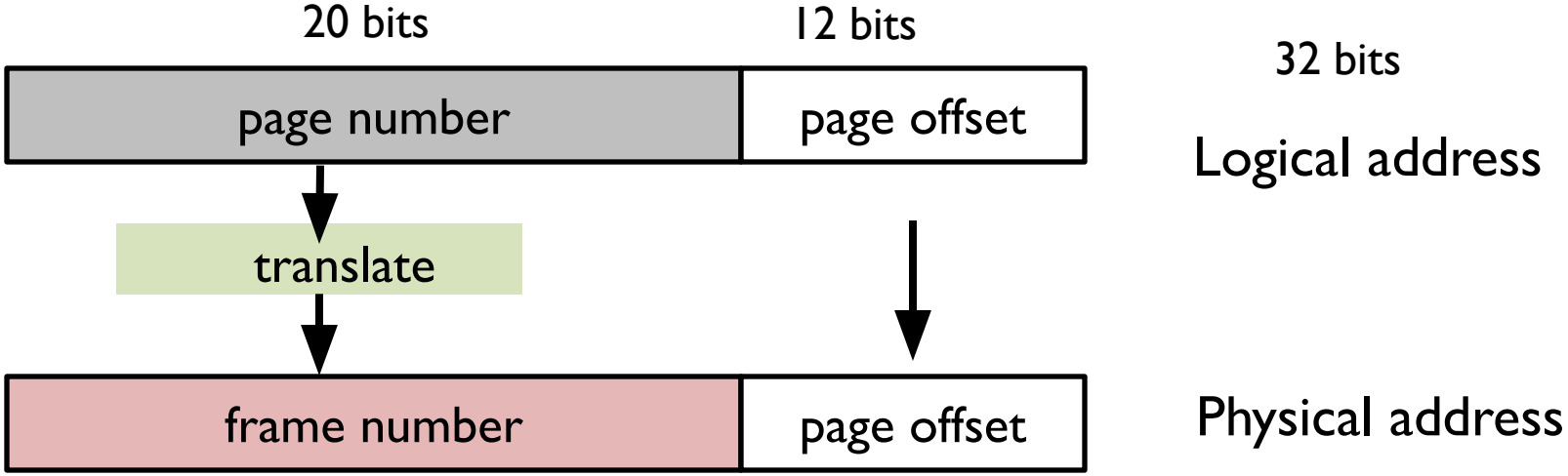
Divide address spaces and physical memory into fixed-sized pages/frames (usually 4K)
Can place pages anywhere in physical memory



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

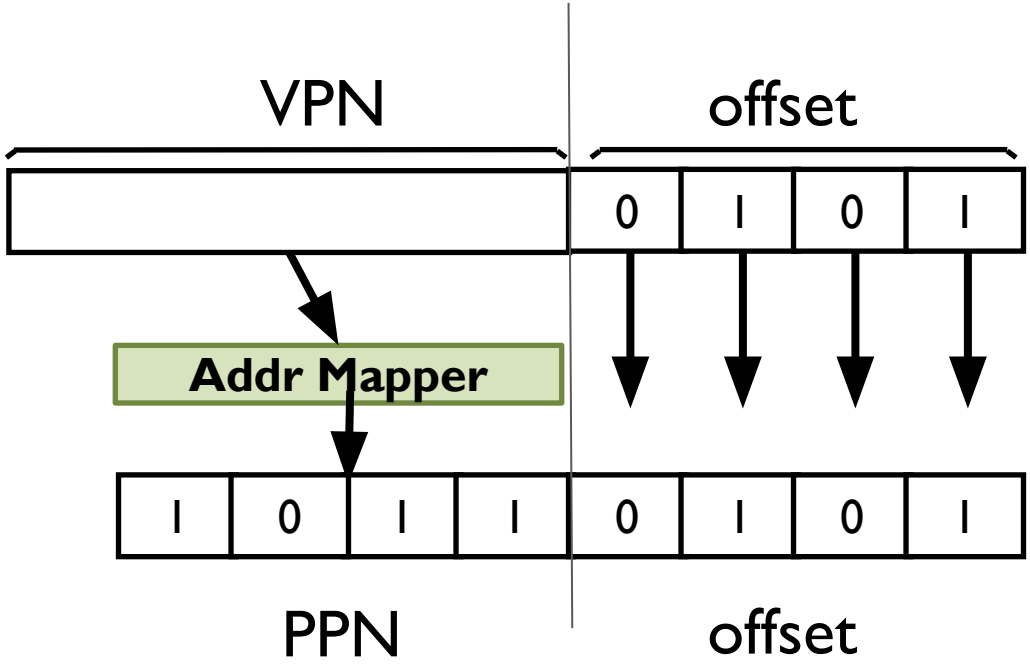


ADDRESS FORMAT

- Chat with neighbors for 2 mins…

Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	# Virt Pages
16 bytes		10		
1 KB		20		
1 MB		32		
512 bytes		16		
4 KB		32		

VIRTUAL □ PHYSICAL PAGE MAPPING



Number of bits in virtual address

need not equal

number of bits in physical address

bits in VPN

> # bits in PPN?

What does this mean?

Is inverse possible?

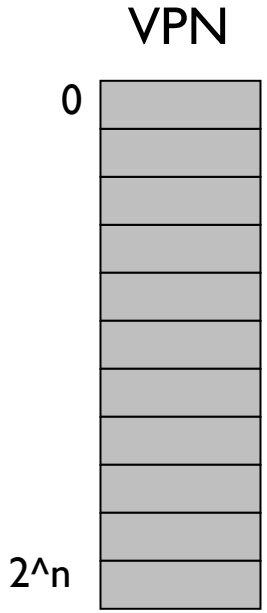
How should OS translate VPN to PPN/PFN?

Linear PageTable

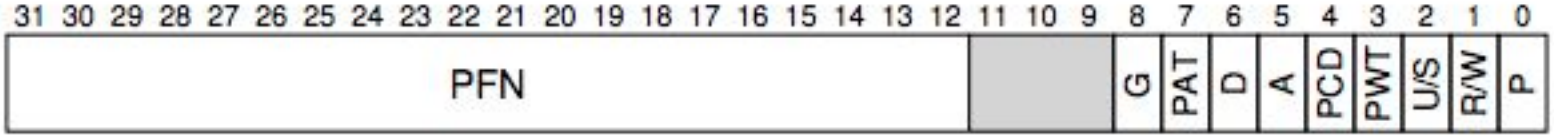
What is a good data structure ?

Simple solution: Linear page table aka *array*

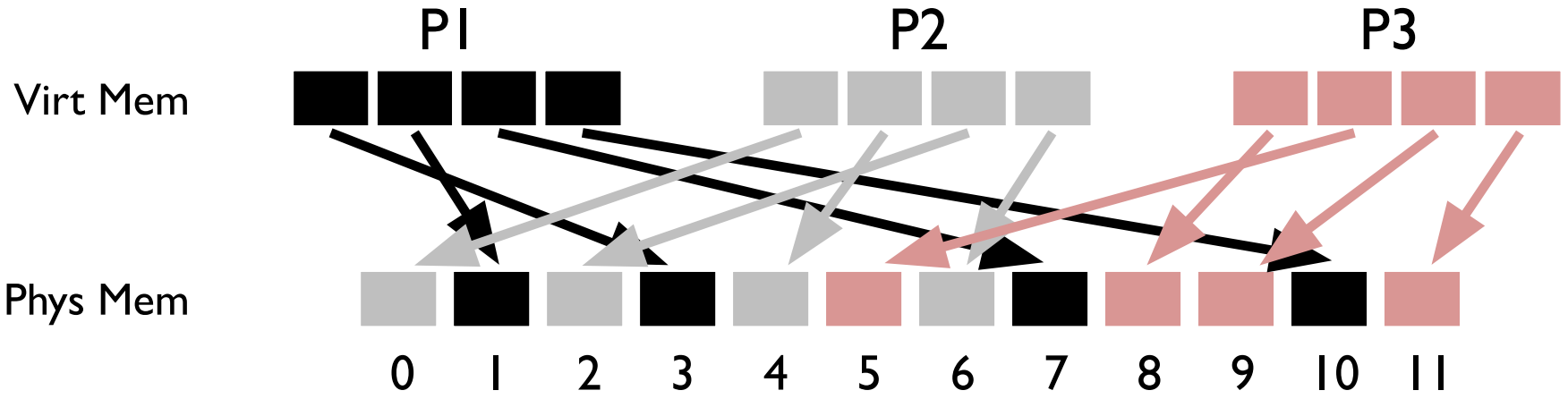
A: accessed, P: present, D: dirty



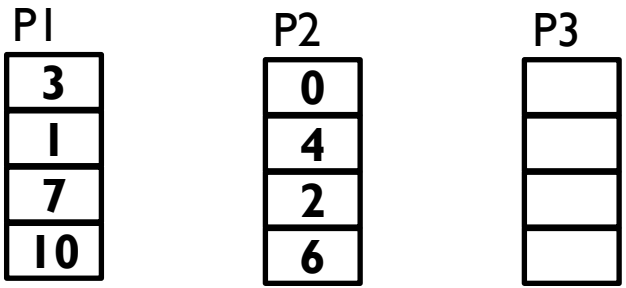
A Single PTE:



FILL IN PAGETABLE



Page Tables:



HOW BIG IS A PAGETABLE?

Consider a **32-bit** virtual address space with 4 KB pages. Assume each PTE is 4 bytes

What will be the overall size of the page table?

WHERE ARE PAGETABLES STORED?

Implication: Store each page table in memory

- Hardware finds page table base with register
- What is this register in 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 PAGETABLE 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)

Pagetable entries are just bits stored in memory

- Agreement between HW and OS about interpretation

MEMORY ACCESSES WITH PAGING

14 bit addresses

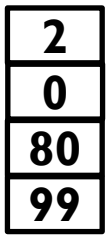
```
0x0010: movl   0x1100, %edi
```

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

How many memory accesses?
To which physical addresses?

Chat with neighbors for 2 mins…

Simplified view of page table



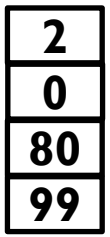
MEMORY ACCESSES WITH PAGING

14 bit addresses

```
0x0010: movl   0x1100, %edi
```

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



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 coalesce with adjacent free space

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
- **Tension? Why not make pages really small?**

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
- Page tables must be contiguously allocated - fix with 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. read contents of **PA** from memory into register

Which steps are expensive?
Chat for a minute…

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

Assume 4-byte PTE, what can you infer?
What is the pt base address?
What is the PPN for VPN 3?

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

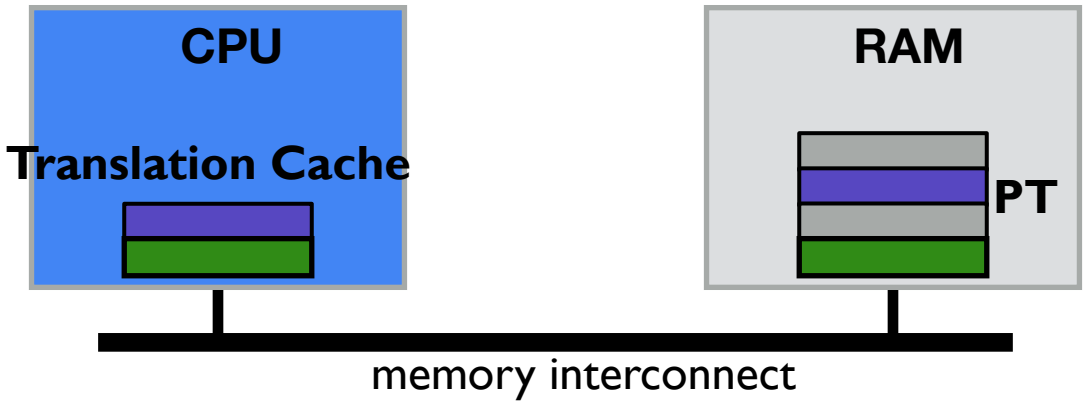
load 0x300C

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

TLB ORGANIZATION

TLB Entry

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

Fully associative

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

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’ and ‘sum’

Assume following virtual address stream:
load 0x1000

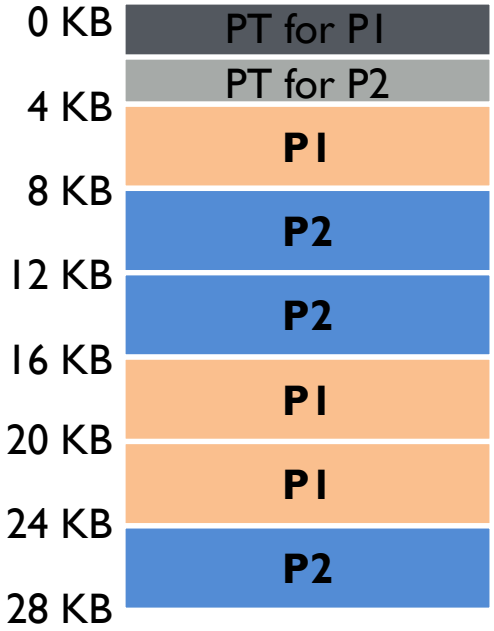
load 0x1004

load 0x1008

load 0x100C
…

What will TLB behavior look like?

TLB Accesses: SEQUENTIAL Example



PTBR

P1 pagetable

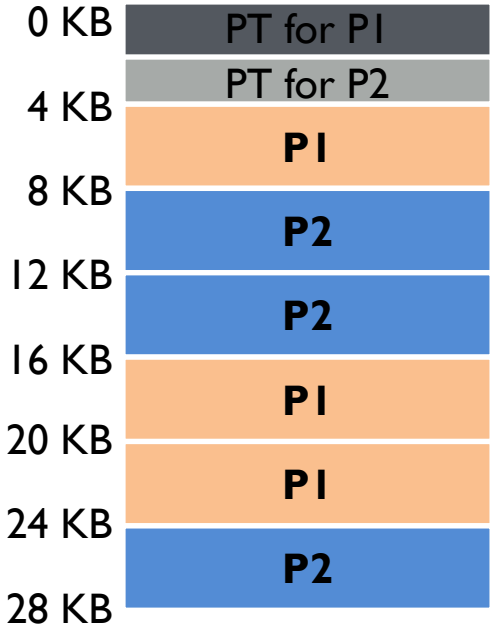
1	5	4	...
0	1	2	3

CPU's TLB

Valid	VPN	PPN

Virt	Phys
load 0x1000	load 0x0004
load 0x1004	load 0x5000
load 0x1008	
load 0x100c	

TLB Accesses: SEQUENTIAL Example



← PTBR

P1 pagetable

1	5	4	…
0	1	2	3

CPU's TLB

Valid	VPN	PPN
1	1	5

Virt	Phys
load 0x1000	load 0x0004
load 0x1004	load 0x5000 (TLB hit)
load 0x1008	load 0x5004 (TLB hit)
load 0x100c	load 0x5008 (TLB hit)
	load 0x500C

PERFORMANCE OF TLB?

```
int sum = 0;
for (i=0; i<2048; 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

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%

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<2048; i++) {
    sum += a[i];
}
```

Sequential array accesses almost always hit in TLB!

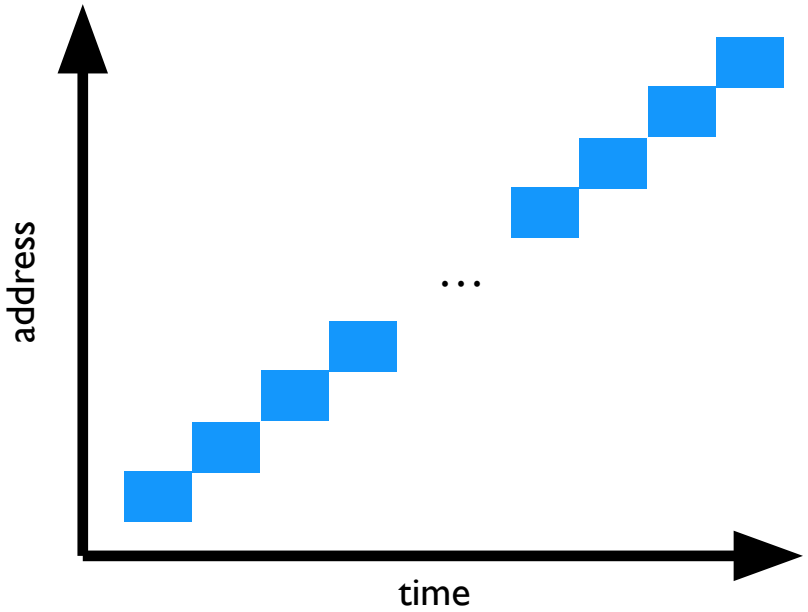
Workload B

```
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

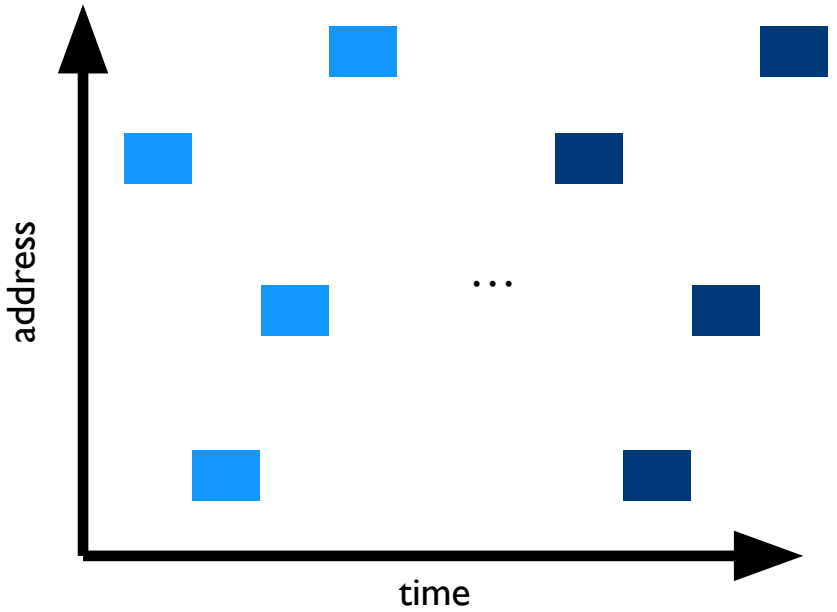
Spatial Locality

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
- How near in future? How many TLB entries are there?

How to replace TLB entries?

LRU: Evict least-recently-used TLB slot

Another option: random sometimes can be better than LRU

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?

1. Flush entire TLB on each context switch (`void flush_tlb_all(void) in Linux`)
   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
   - Must match ASID on lookups

UNMAPPING MEMORY

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

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

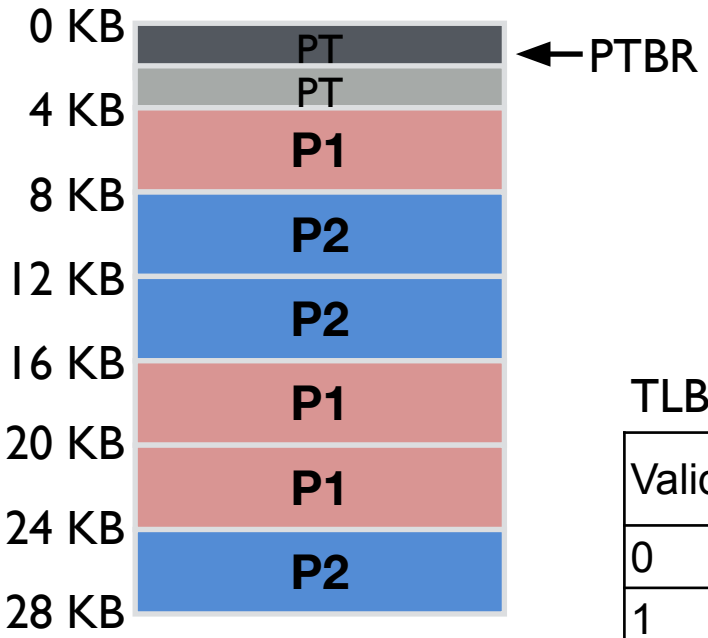
Several ways to do this:

1. Flush all tlb entries for the proces – `void flush_tlb_mm(struct mm_struct *mm)`

Inefficient - but useful for fork and exec

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

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 0x1444 ASID: 11	

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

- I TLB for data, I TLB for instructions
- I TLB for regular pages, I TLB for "super pages"

HW AND OS ROLES

If H/W handles TLB Miss

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

If OS handles TLB Miss:

“Software-managed TLB”

- CPU traps into OS upon TLB miss.
- OS interprets pagetables as it chooses
- Modify TLB entries with privileged instruction

TLB Summary

Pages are great, but accessing page tables for every memory access is slow

Cache recent page translations 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

TLBs increase cost of context switches

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

In different systems, hardware or OS handles TLB misses

NEXT STEPS

Next class: More TLBs and better pagetables!