MEMORY: PAGING AND TLBS

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

- Project 2 done!
- Project 3 is out! Start early?

$\hookrightarrow$ Discussion $\rightarrow$ notes, example code

$\hookrightarrow$ More time $\rightarrow$ More work?

AGENDA / LEARNING OUTCOMES

Memory virtualization

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

RECAP

MEMORY VIRTUALIZATION

Transparency: Process is unaware of sharing

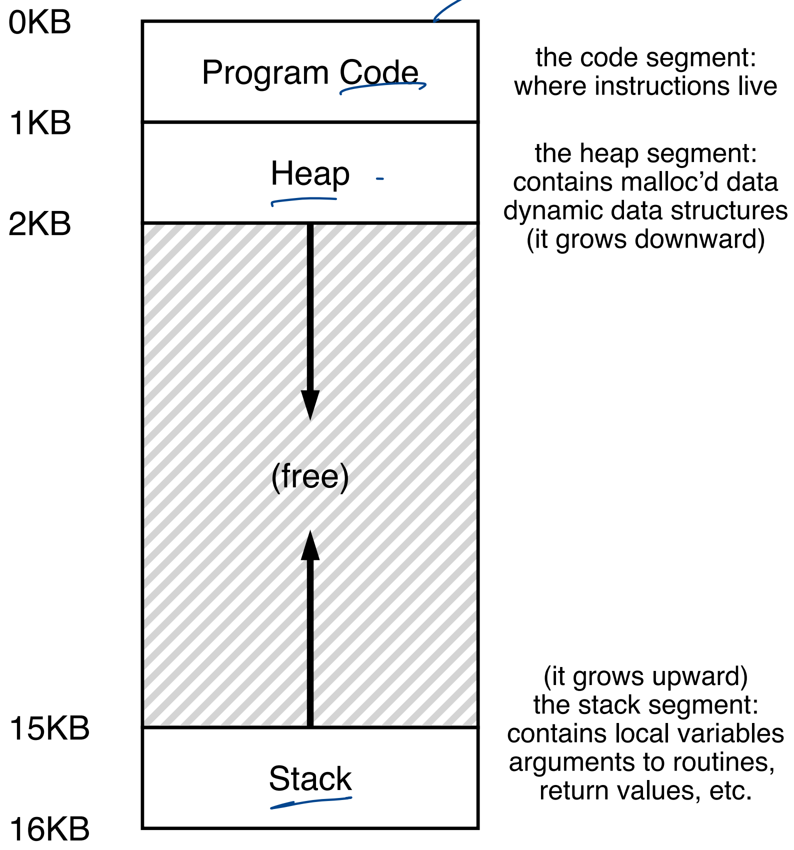
Protection: Cannot corrupt OS or other process memory

Efficiency: Do not waste memory or slow down processes

Sharing: Enable sharing between cooperating processes

RECAP: WHAT IS IN ADDRESS SPACE?

virtual memory



0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.

Static: Code and some global variables

Dynamic: Stack and Heap

REVIEW: SEGMENTATION

Divide address space into logical segments

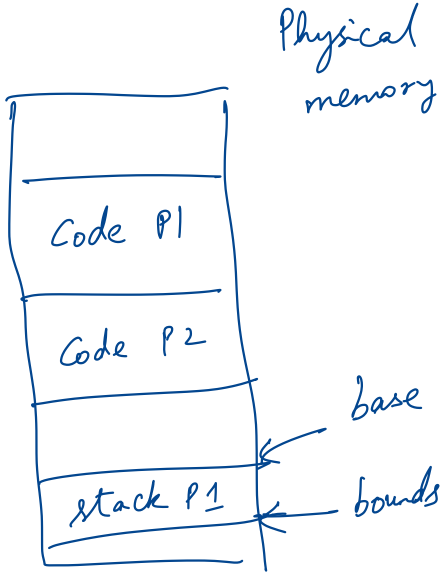
Each segment corresponds to logical entity in address space

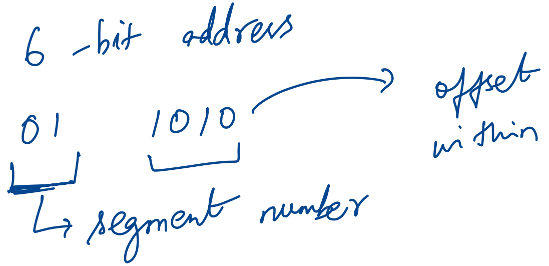
(code, stack, heap)

Each segment has separate base + bounds register

How does process designate a particular segment?

- Top bits of logical address select segment
- Low bits of logical address select offset within segment





EXAMPLE: SEGMENTATION

Code

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

12 bits of offset

2 bits segment number

%rip: 0x0010

Seg	Base	Bounds
0	0x4000	0xfff
1	0x5800	0xfff
2	0x6800	0x7ff

1. Fetch instruction at logical addr 0x0010

   Physical addr: 0x4000 + 0x 0010

   = 0x 4010

2. Exec, load from logical addr 0x1100

   Physical addr: 0x 5800 + 0x 0100

   = 0x 5900

14 bit address

2-bit | 4-bit | 4-bit | 4-bit

segment | 12 bits offset

QUIZ 8!

https://tinyurl.com/cs537-sp23-quiz8

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	$\boxed{0x0000}$	$\boxed{0x000}$	0 0

0x

↓ Don't have permission

Remember:
1 hex digit → 4 bits

Translate logical (in hex) to physical

0x0240:	0x 2240
0x1108:	0x 0108
0x265c:	0x 365c
0x3002:	Fails!

HOW DOES THIS LOOK IN X86

Stack Segment (SS): Pointer to the stack

Code Segment (CS): Pointer to the code

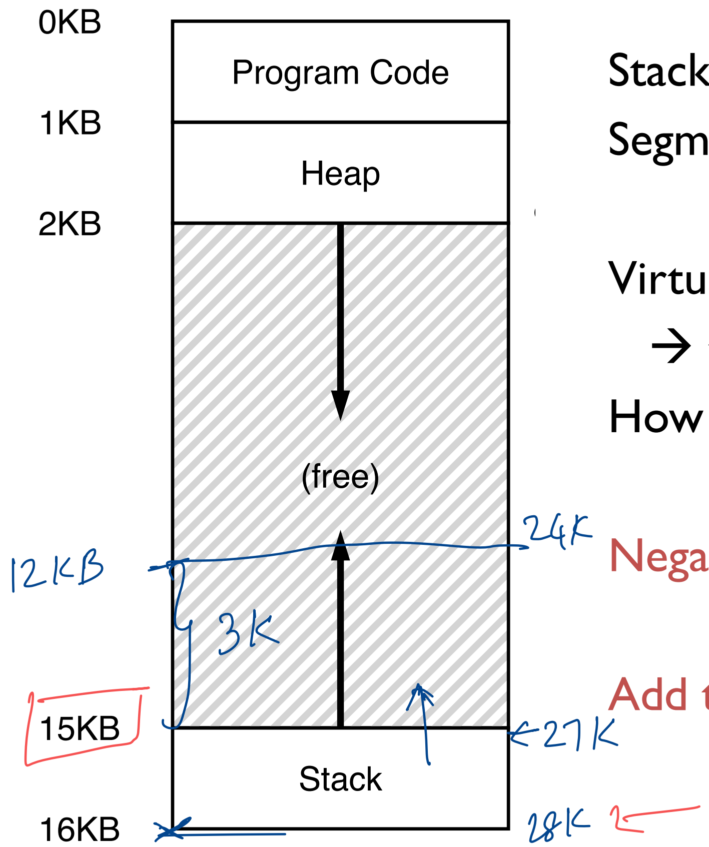
Data Segment (DS): Pointer to the data

Extra Segment (ES): Pointer to extra data

F Segment (FS): Pointer to more extra data

G Segment (GS): Pointer to still more extra data

NOTE: HOW DO STACKS GROW ?



Stack goes 16K → 12K, in physical memory is 28K → 24K

Segment base is at 28K

Virtual address 0x3C00 = 15K

→ top 2 bits (0x3) segment ref, offset is 0xC00 = 3K

How do we make CPU translate that ?

Negative offset = subtract max segment from offset

= 3K – 4K = -1K

Add to base = 28K – 1K = 27K

stack pointer

ADVANTAGES OF SEGMENTATION

Stack and heap can grow independently

- Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
- Stack: OS recognizes reference outside legal segment, extends stack implicitly

Different protection for different segments

- Enables sharing of selected segments
- Read-only status for code

Supports dynamic relocation of each segment

DISADVANTAGES OF SEGMENTATION

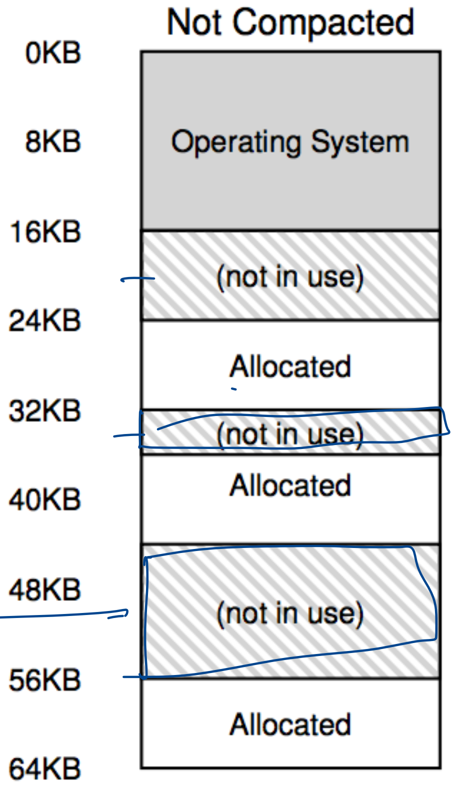
Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation

Free space Fragmented

→ allocations need to be contiguous !



PAGING

PAGING

Goal: Eliminate requirement that address space is contiguous

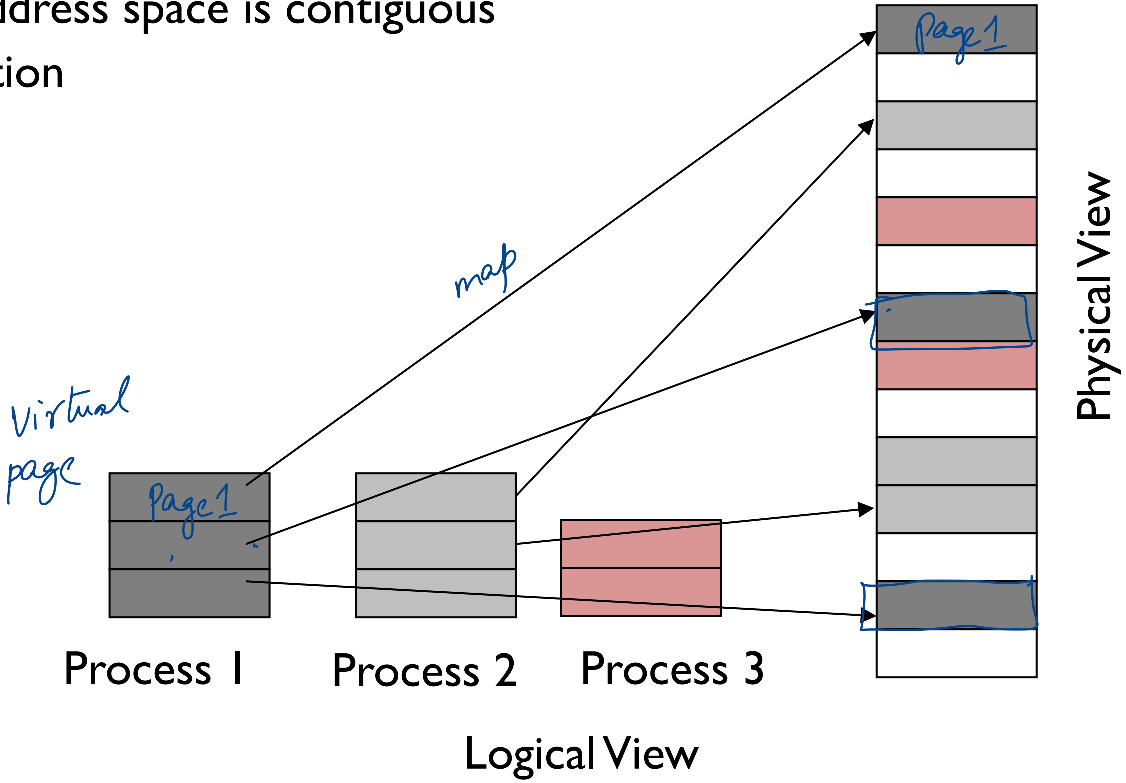
Eliminate external fragmentation

Grow segments as needed

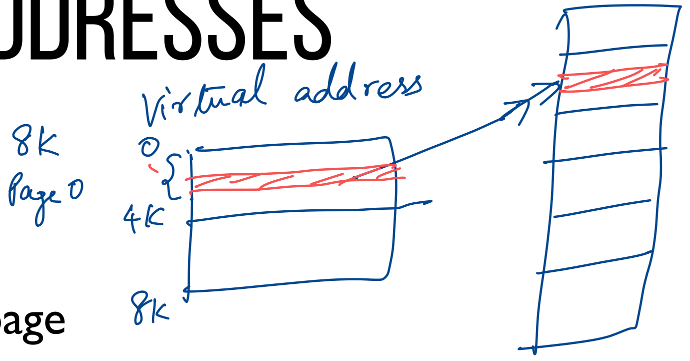
Idea:

Divide address spaces and physical memory into fixed-sized pages

Size: 2^n, Example: 4KB

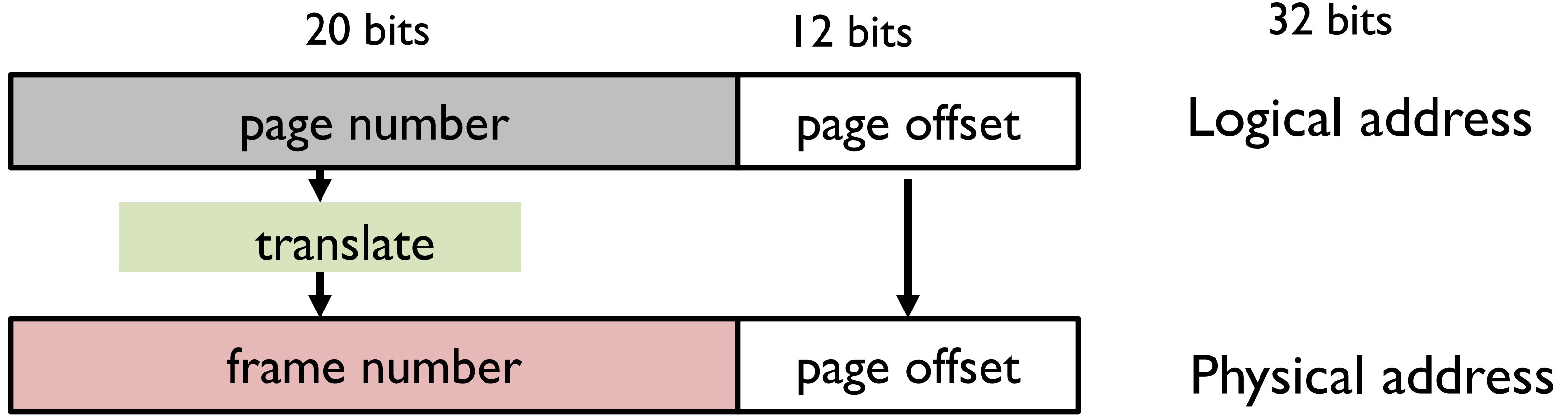


TRANSLATION OF PAGE ADDRESSES



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



No addition needed; just append bits correctly!

ADDRESS FORMAT

Given known page size, how many bits are needed in address to specify offset in page?

$\log_2(\text{Page size}) = \text{bits}$

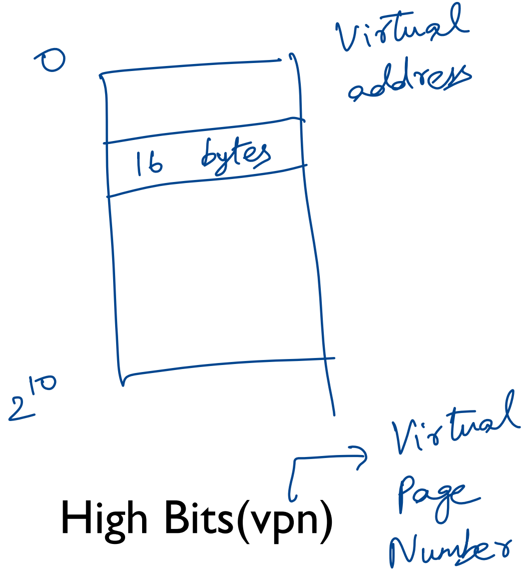
$\updownarrow$

$2^{\text{bits}} = \text{Page size}$

Page Size	Low Bits (offset)
16 bytes	4
1 KB	10
1 MB	20
512 bytes	9
4 KB	12

$\underline{0101} \hookrightarrow 0 \text{ to } 15$

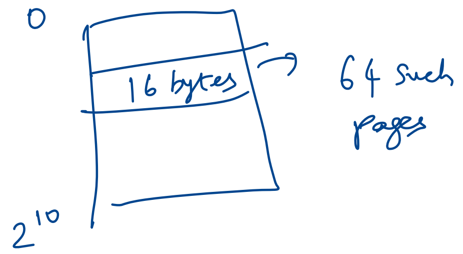
ADDRESS FORMAT



Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

Page Size	Low Bits(offset)	Virt Addr Total Bits	High Bits(vpn)
16 bytes	4	10	6
1 KB	10	20	10
1 MB	20	32	12
512 bytes	9	16	7
4 KB	12	32	20

ADDRESS FORMAT



Given number of bits for vpn, how many virtual pages can there be in an address space?

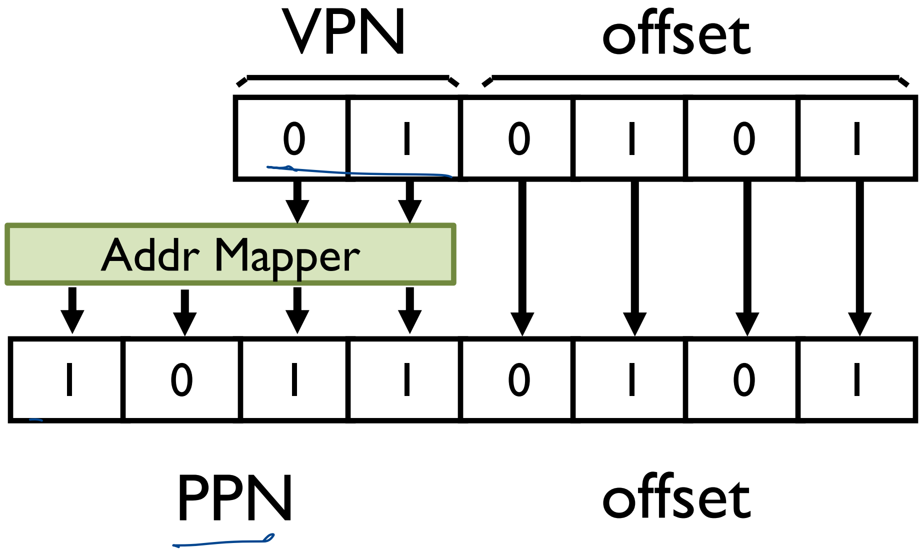
Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	Virt Pages
16 bytes	4	10	6	$2^6 = \underline{64}$
1 KB	10	20	10	2^{10}
1 MB	20	32	12	2^{12}
512 bytes	9	16	7	2^7
4 KB	12	32	20	2^{20}

VIRTUAL → PHYSICAL PAGE MAPPING

Number of bits in virtual address

need not equal

number of bits in physical address



How should OS translate VPN to PPN?

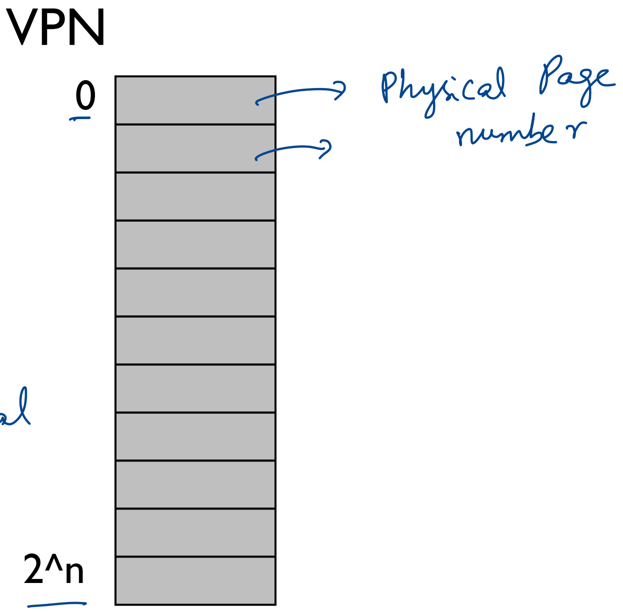
PAGETABLES

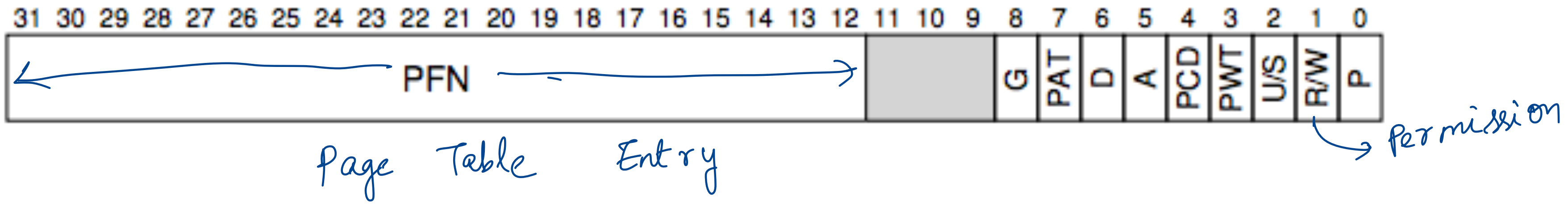
What is a good data structure ?

Simple solution: Linear page table aka *array*

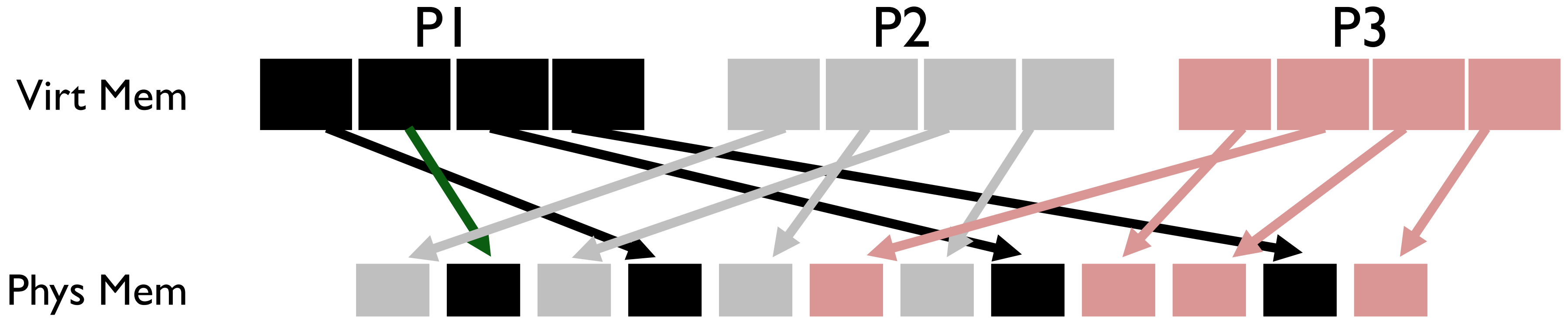
PT = 64 virtual pages ⇒ 64 entries array

PT[0] → physical address for virtual page zero

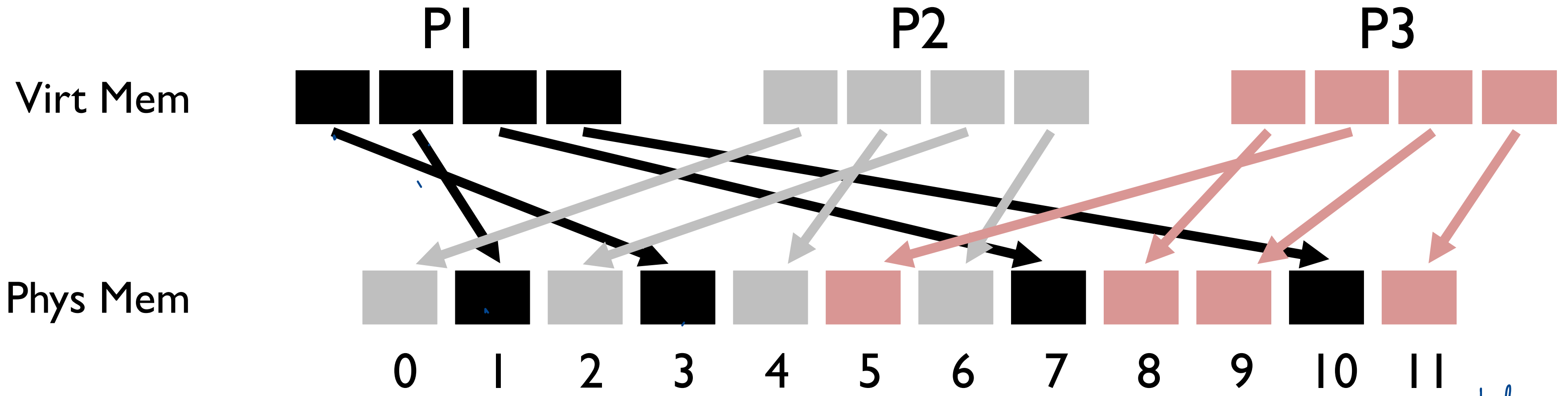




PER-PROCESS PAGETABLE



FILL IN PAGETABLE



Page Tables:

P1	P2	P3
3		
1		
7		
10		

→ OS switch page table when context switching

QUIZ 9

https://tinyurl.com/cs537-sp23-quiz9

Description	Name of approach
1. one process uses RAM at a time	Time Sharing
2. rewrite code and addresses before running	Static relocation
3. add per-process starting location to virt addr to obtain phys addr	Base $\leftarrow$ Dynamically using MMU
4. dynamic approach that verifies address is in valid range	Base + bounds
5. several base+bound pairs per process	Segmentation

Candidates: Segmentation, Static Relocation, Base, Base+Bounds, Time Sharing

QUIZ9: HOW BIG IS A PAGETABLE?

Page Table Entry

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

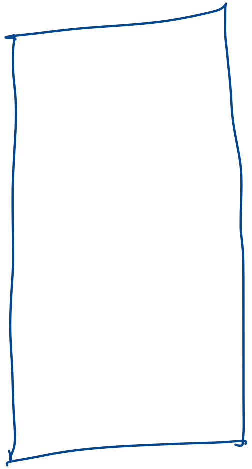
How many bits do we need to represent the offset within a page?

12 bits

How many virtual pages will we have in this case?

2^{20} = num of virtual page

array = num entries × size of each



What will be the overall size of the page table?

$2^{20} \times 4$ bytes = 2^{22} bytes = 4 MB

WHERE ARE PAGETABLES STORED?

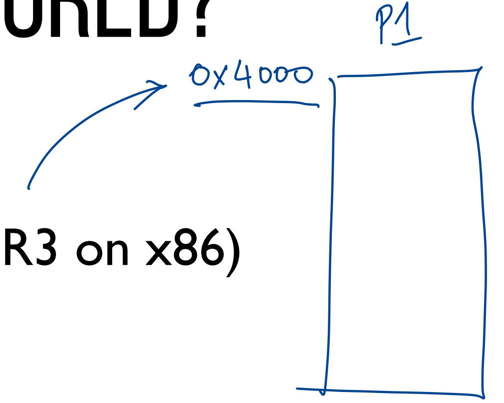
Implication: Store each page table in memory

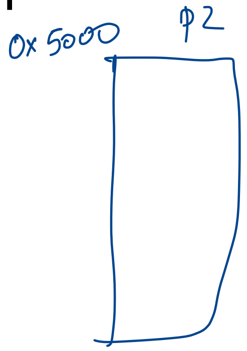
Hardware finds page table base with register (e.g., CR3 on x86)

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





$CR3 \leftarrow 0x5000$ when P2 is selected

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 $\rightarrow$ 2 bits for VPN

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

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

0x5000

Simplified view of page table

2
0
80
99

Fetch instruction at logical addr 0x$\boxed{0}$010

Access page table to get ppn for vpn 0

Mem ref 1: 0x5000 $\leftarrow$ get PTE for VPN 0

Learn vpn 0 is at ppn 2

Fetch instruction at 0x 2010 (Mem ref 2)

Two steps

- Fetch PTE for VPN
- Append PPN and offset

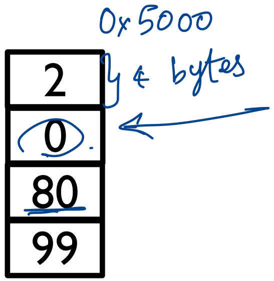
MEMORY ACCESSES WITH PAGING

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



Exec, load from logical addr 0x$\boxed{1}$100

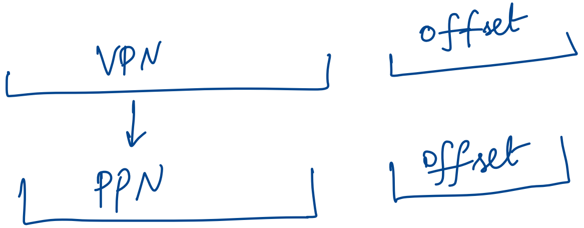
Access page table to get ppn for vpn 1

Mem ref 3: 0x 5004

Learn vpn 1 is at ppn 0

Movl from ______ into reg (Mem ref 4)

0x 0 100



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

2
0
80
99

Fetch instruction at logical addr 0x0010

Access page table to get ppn for vpn 0

Mem ref 1: ____0x5000____

Learn vpn 0 is at ppn 2

Fetch instruction at ___0x2010___ (Mem ref 2)

Exec, load from logical addr 0x1100

Access page table to get ppn for vpn 1

Mem ref 3: ____0x5004____

Learn vpn 1 is at ppn 0

Movl from ___0x0100___ into reg (Mem ref 4)

PROS/CONS OF PAGING

Within Process

Page size 4KB

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

Internal fragmentation

- Page size may not match process needs
- Wasted memory grows with larger pages

Additional memory reference to page table →

- Page table must be stored in memory
- MMU stores only base address of page table

Storage for page tables may be substantial

- Requires PTE for all pages in address space
- Entry needed even if page not allocated ?

4MB for one process ←

SUMMARY: PAGE TRANSLATION STEPS

For each mem reference:

Given VA

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory → Go to DRAM
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory into register

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'

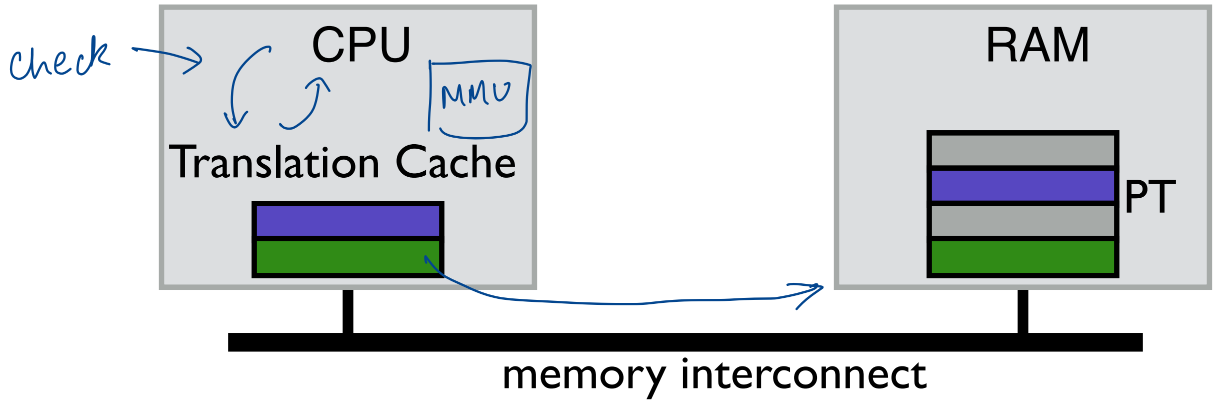
What virtual addresses?

load 0x3000	a[0]
load 0x3004	a[1]
load 0x3008	a[2]
	⋮
load 0x300C	

What physical addresses?

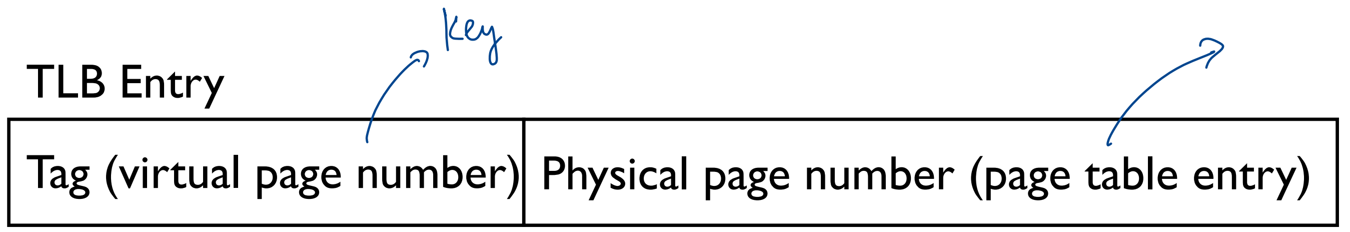
load 0x100C	→ Page Table Entry
load 0x7000	→ data
load 0x100C	
load 0x7004	
load 0x100C	
load 0x7008	
load 0x100C	
load 0x700C	

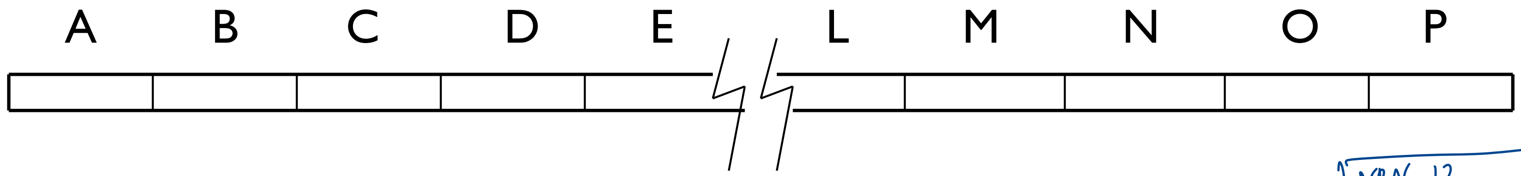
STRATEGY: CACHE PAGE TRANSLATIONS

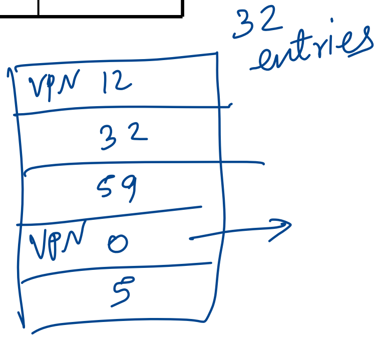


TLB: TRANSLATION LOOKASIDE BUFFER

TLB ORGANIZATION


TLB Entry
Tag (virtual page number)
Physical page number (page table entry)
Key





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'

Assume following virtual address stream:

load 0x1000 $\rightarrow$ load 0x100C
load 0x7000

load 0x1004 $\rightarrow$ TLB hit
load 0x7004

load 0x1008

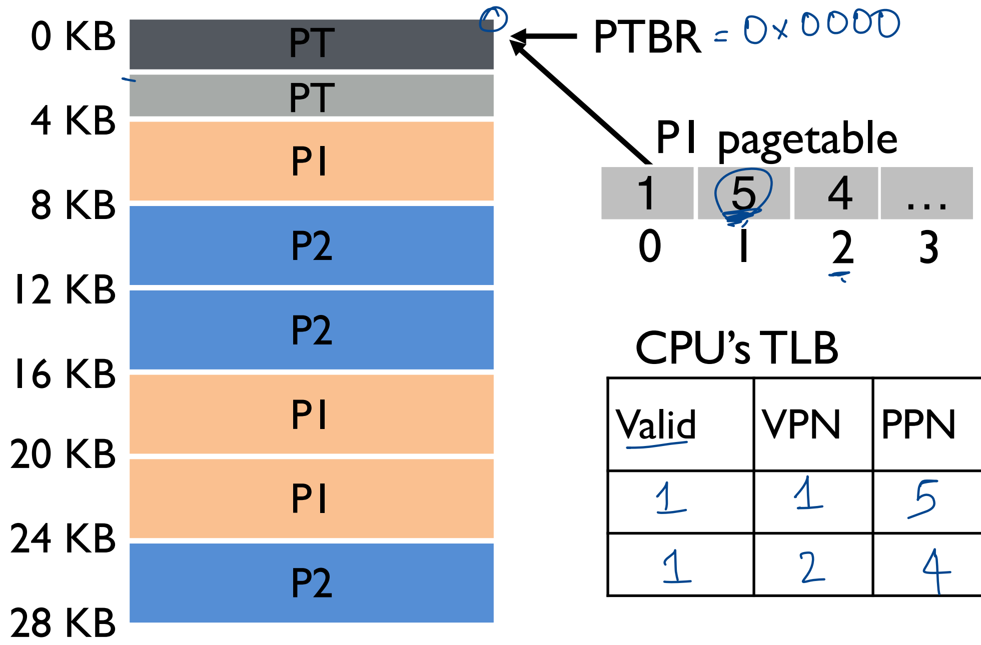
load 0x100C

…

1025 $\rightarrow$ TLB miss
load (0x100C + 4)

What will TLB behavior look like?

TLB ACCESSES: SEQUENTIAL EXAMPLE



CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

Virt

load 0x1000 → PTE → load 0x0004; data → load 0x5000

load 0x1004 → TLB hit load 0x5004

⋮

load 0x1008

load 0x100c

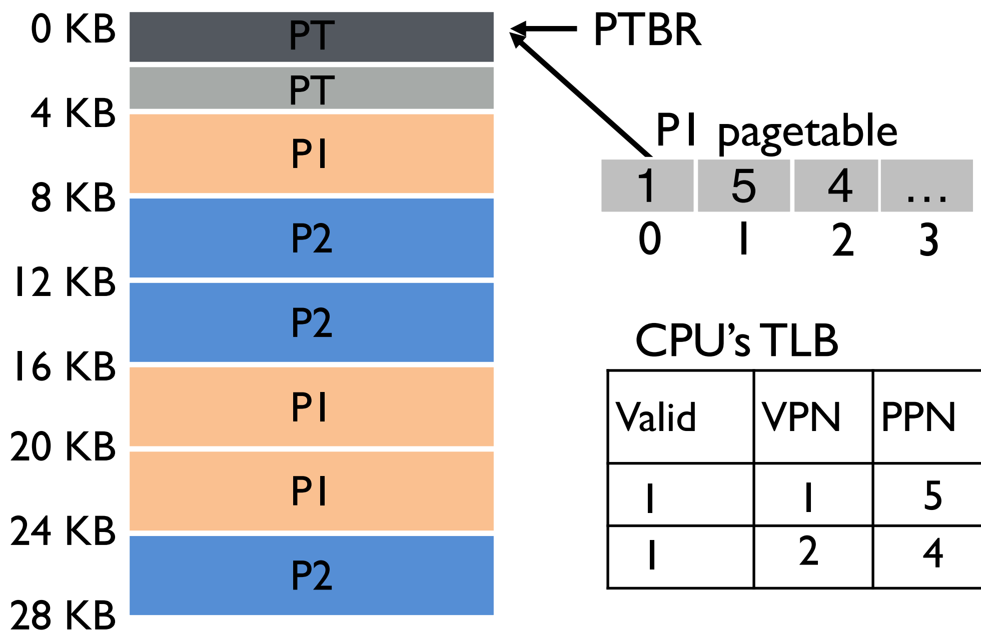
...

load 0x2000 → TLB miss load 0x0008

load 0x4000

load 0x2004

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

QUIZ 10: TLBS

https://tinyurl.com/cs537-sp23-quiz10

Consider a processor with 16-bit address space and 4kB page size.
Assume Page Table is at 0x2000 and each PTE is of 4 bytes.

Simplified view of the PT

VPN	PPN
4	7
5	8
3	9
2	1

Virtual Addresses

```
0x3000: load 0x5320, %eax
0x3004: load 0x4004, %ebx
0x3008: mul %ecx, %eax, %ebx
0x300C: store %ebx, 0x5324
0x3010: load 0x5328, %ebx
```

Memory accesses

Total number of memory accesses

QUIZ 10: TLBS

Simplified view of the PT

VPN	PPN
4	7
5	8
3	9
2	1

Virtual Addresses
```
0x3000: load 0x5320, %eax
0x3004: load 0x4004, %ebx
0x3008: mul %ecx, %eax, %ebx
0x300C: store %ebx, 0x5324
0x3010: load 0x5328, %ebx
```

Valid	VPN	PPN
0	2	6
0	7	23
0	2	5
0	3	2
0	1	89

Memory accesses

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?

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!

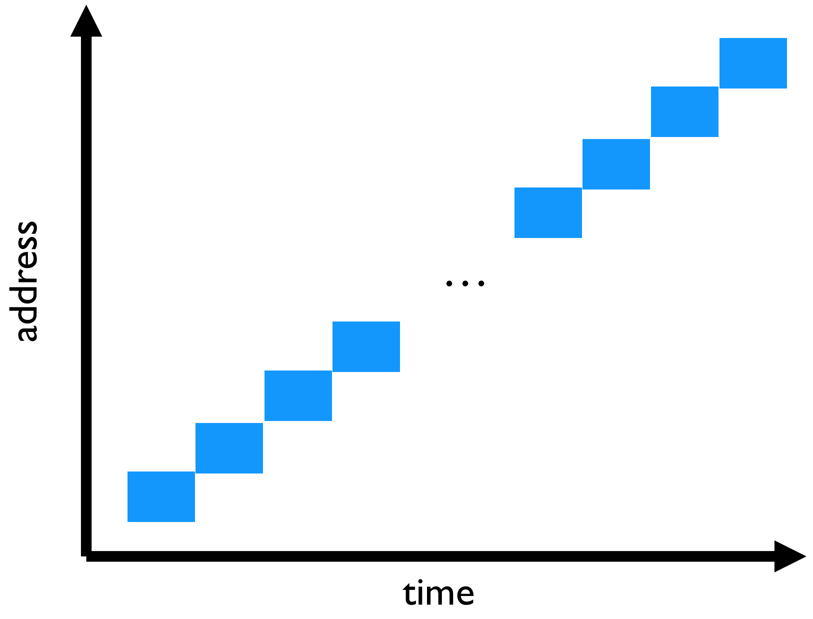
Workload B

```
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

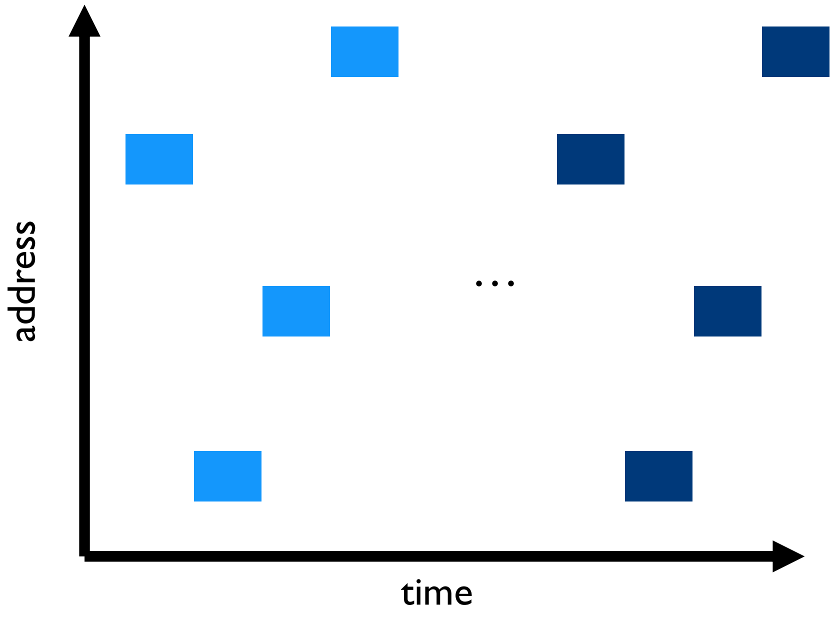
Spatial Locality

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?

OTHER TLB CHALLENGES

How to replace TLB entries ? LRU ? Random ?

TLB on context switches ? HW or OS ?

NEXT STEPS

Project 3 is out!

Next class: More TLBs and better pagetables!