MEMORY: PAGING AND TLBS

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

- Project 1b is due Friday
- Project 1a grades are out
- Project 2a going out tomorrow

- Discussion section: Process API, Project 2a

AGENDA / LEARNING OUTCOMES

Memory virtualization

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

RECAP

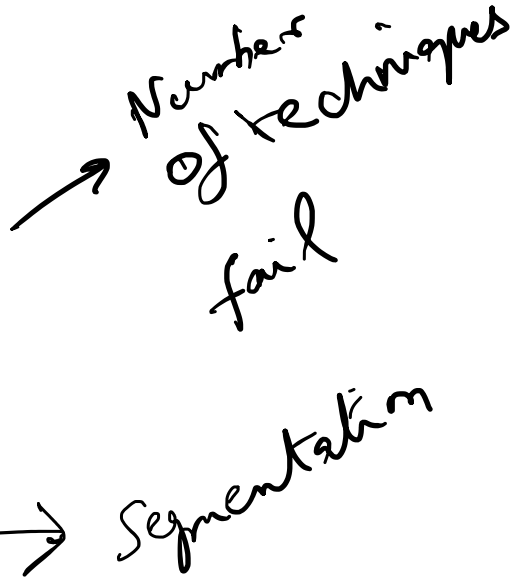
MEMORY VIRTUALIZATION

Transparency: Process is unaware of sharing

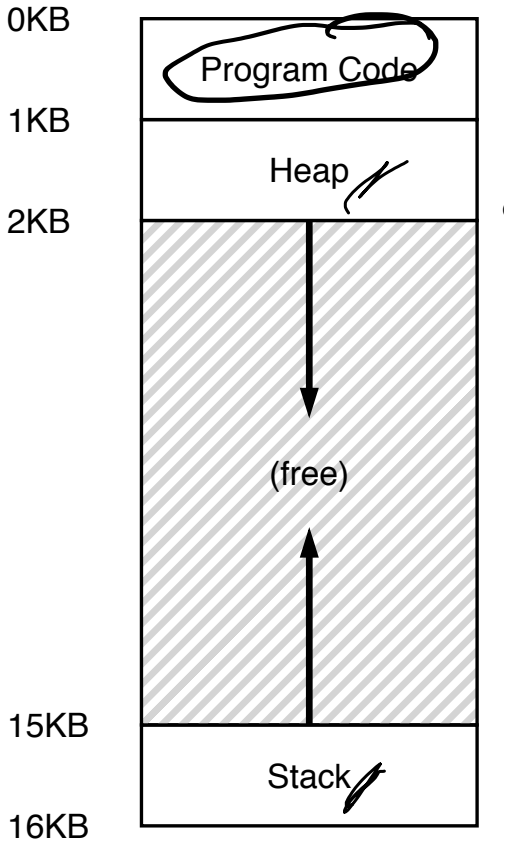
Protection: Cannot corrupt OS or other process memory

Efficiency: Do not waste memory or slow down processes

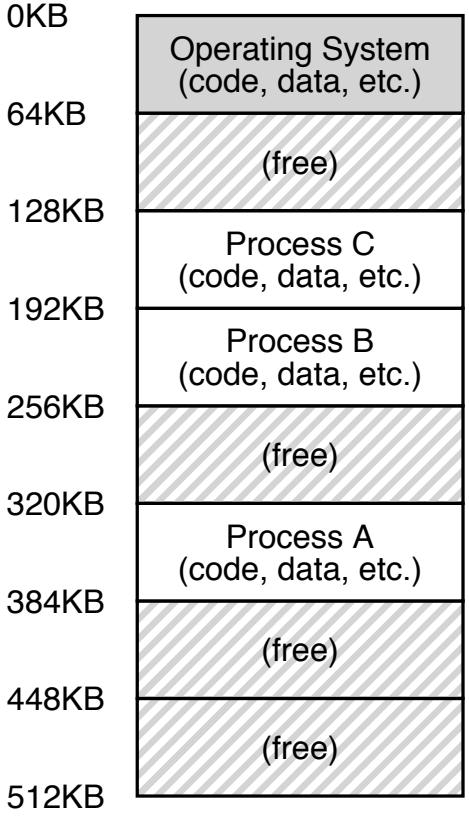
Sharing: Enable sharing between cooperating processes



ABSTRACTION: ADDRESS SPACE



View that a process sees



SEGMENTATION IMPLEMENTATION

MMU contains Segment Table (per process)

- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments;

How many bits for segment?

How many bits for offset?

Segment	Base	Bounds	R	W
0	0x2000	0x6ff	1	0
1	0x0000	0x4ff	1	1
2	0x3000	0xfff	1	1
3	0x0000	0x000	0	0

remember:
1 hex digit → 4 bits

QUIZ: ADDRESS TRANSLATIONS WITH SEGMENTATION

14 bit = 3 hex bits are (2 bit top hex bit is 2 bits

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

N

Remember:
1 hex digit→4 bits

0x0011 fff

max addr 0x3fff

Translate logical (in hex) to physical

0x0240: 0x2000 + 0x0240 = 0x2240

0x1108:

0x265c: 0x3000 + 0x065c = 0x365c

0x3002: 0x0000 + 0x002 = 0x0002

FAIL

REVIEW: MEMORY ACCESSES

No extra memory accesses

```
0x0010: movl  0x1100, %edi
0x0013: addl  $0x3, %edi
0x0019: movl  %edi, 0x1100
```

%rip: 0x0010

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

1. Fetch instruction at logical addr 0x0010

   Physical addr: 0x4010

2. Exec, load from logical addr 0x1100

   Physical addr: 0x5900

3. Fetch instruction at logical addr 0x0013

   Physical addr: 0x4013

4. Exec, no load

5. Fetch instruction at logical addr 0x0019

   Physical addr: 0x4019

6. Exec, store to logical addr 0x1100

   Physical addr: 0x5900

ADVANTAGES OF SEGMENTATION

Enables sparse allocation of address space

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

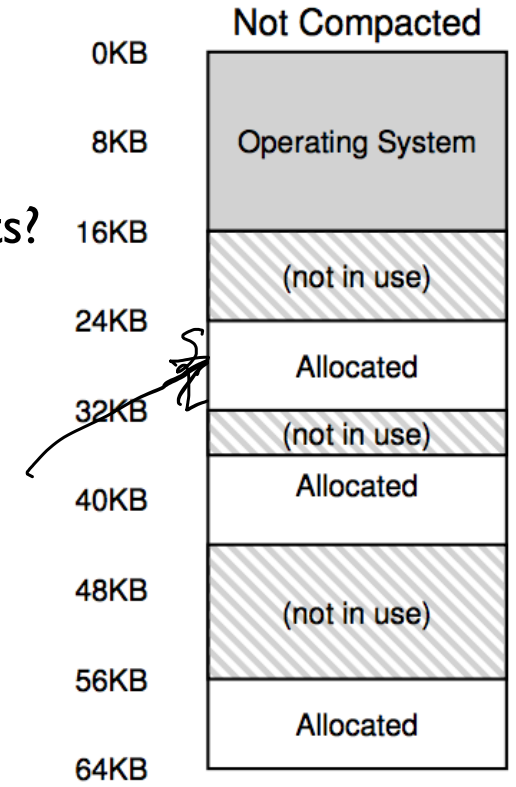
Supports dynamic relocation of each segment

DISADVANTAGES OF SEGMENTATION

Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation



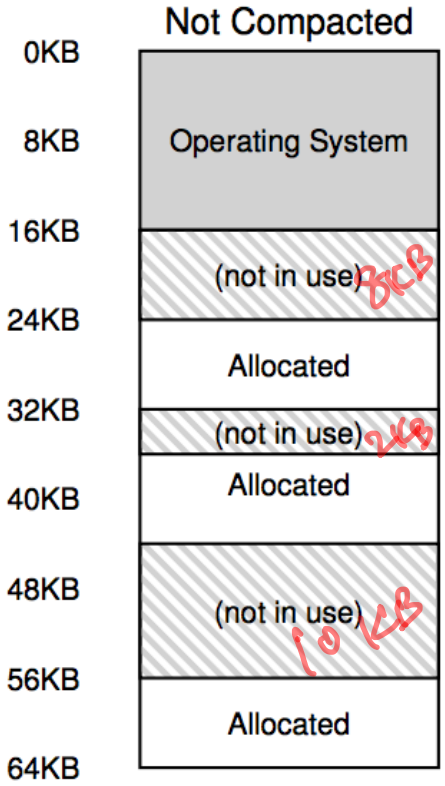
REVIEW: MATCH DESCRIPTION

Description	Name of approach
1. one process uses RAM at a time 2. rewrite code and addresses before running 3. add per-process starting location to virt addr to obtain phys addr 4. dynamic approach that verifies address is in valid range 5. several base+bound pairs per process	

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

PAGING

FRAGMENTATION

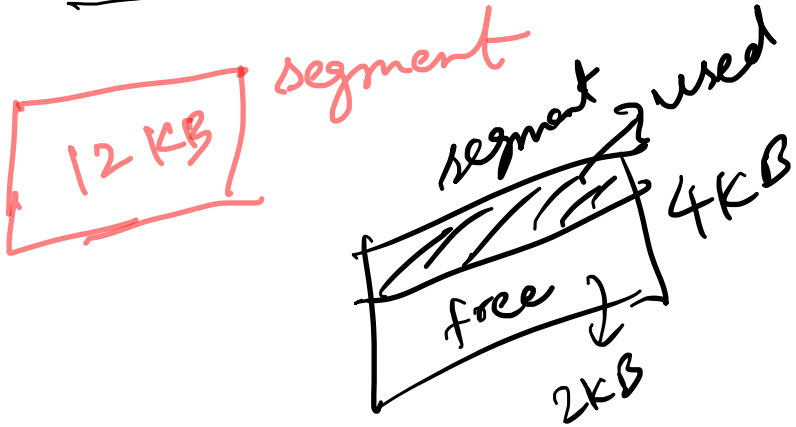


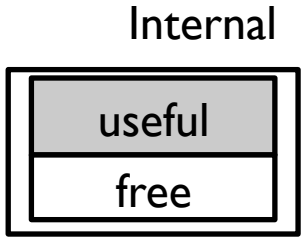
Definition: Free memory that can't be usefully allocated

Types of fragmentation

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

Internal: Visible to requester





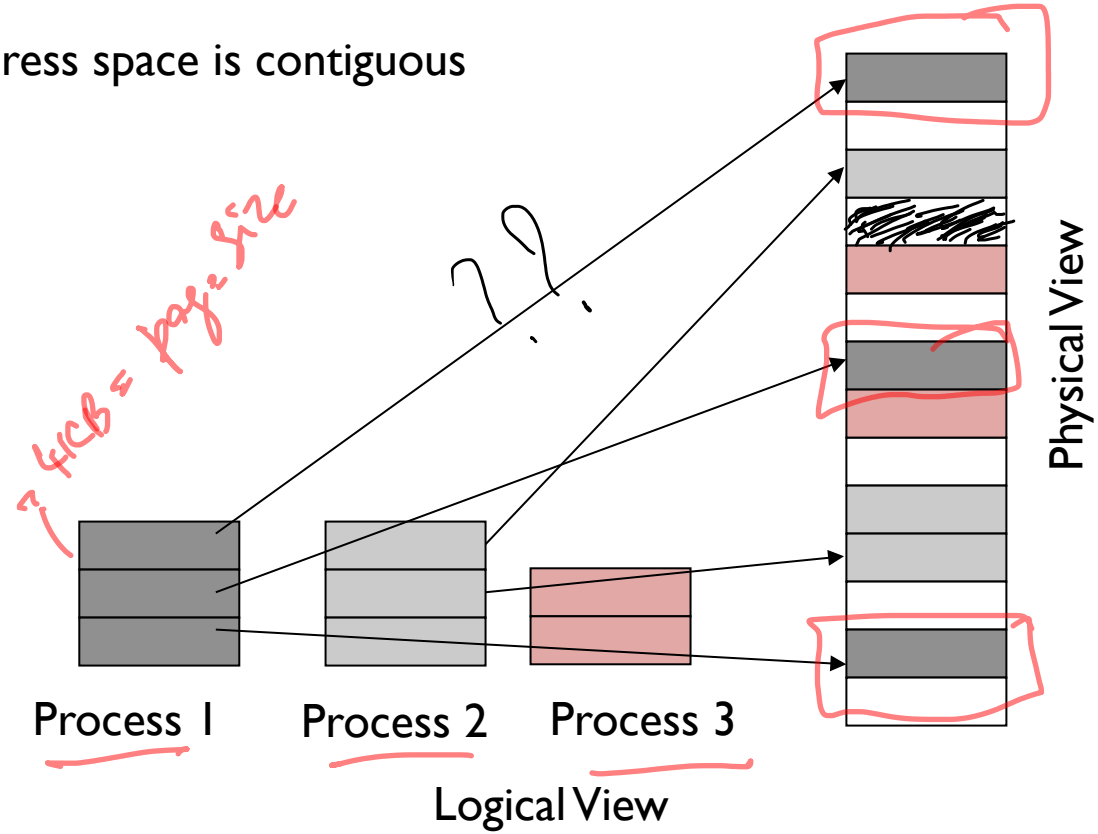
PAGING

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

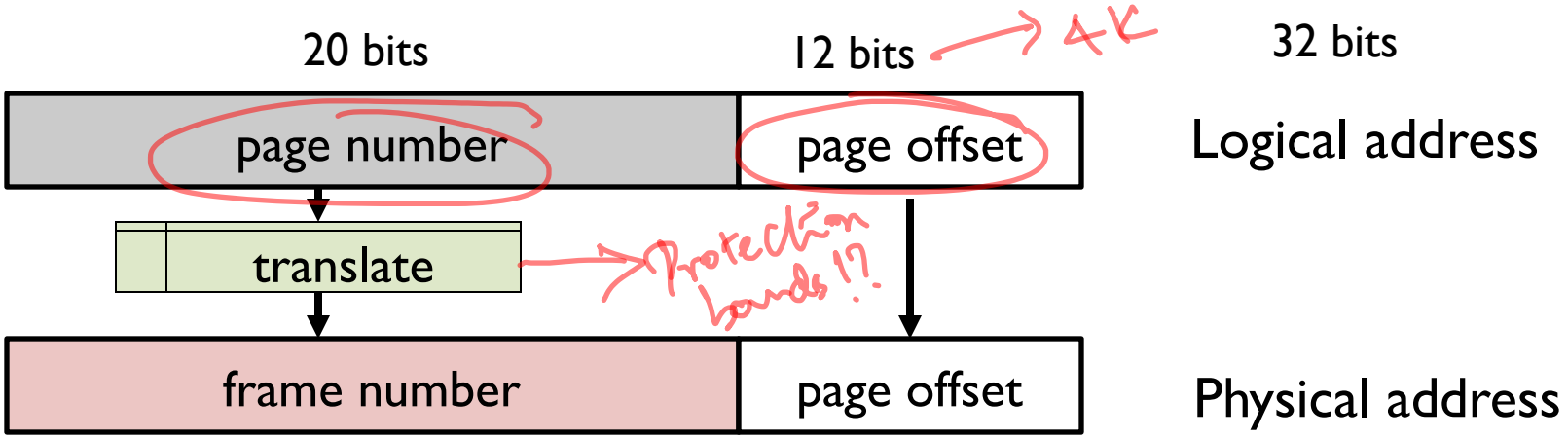
Size: 2^n, Example: 4KB



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?

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

$\log_2(512)$

$(2^{10} = 1KB)$

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 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?

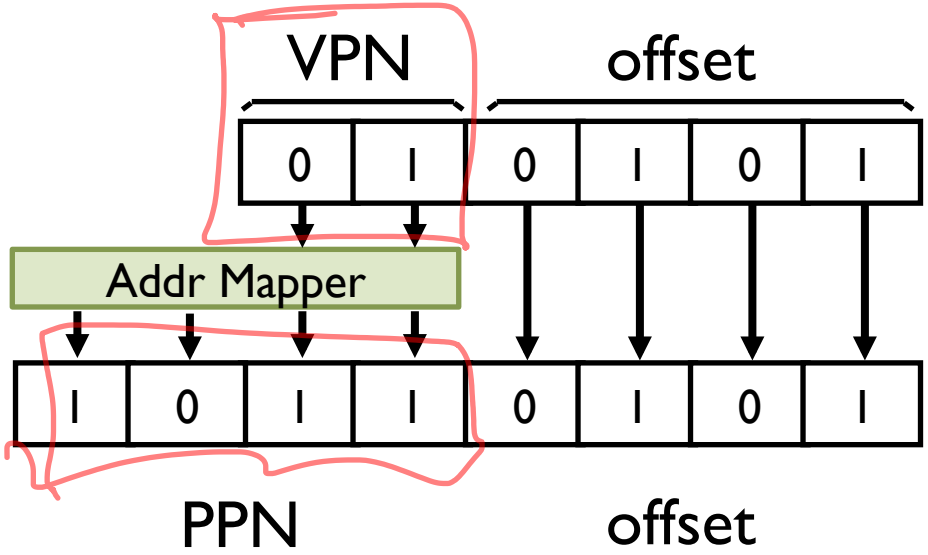
Page Size	Low Bits (offset)	Virt Addr Bits	High Bits (vpn)	Virt Pages
16 bytes	4	10	6	$2^6 = 64$
1 KB	10	20	10	$2^{10} = 1024$
1 MB	20	32	12	$2^{12} = 4K$
512 bytes	9	16	7	$2^7 = 128$
4 KB	12	32	20	$2^{20} =$ 1MB

VIRTUAL → PHYSICAL PAGE MAPPING

Number of bits in virtual address

need not equal

number of bits in physical address



64 bit

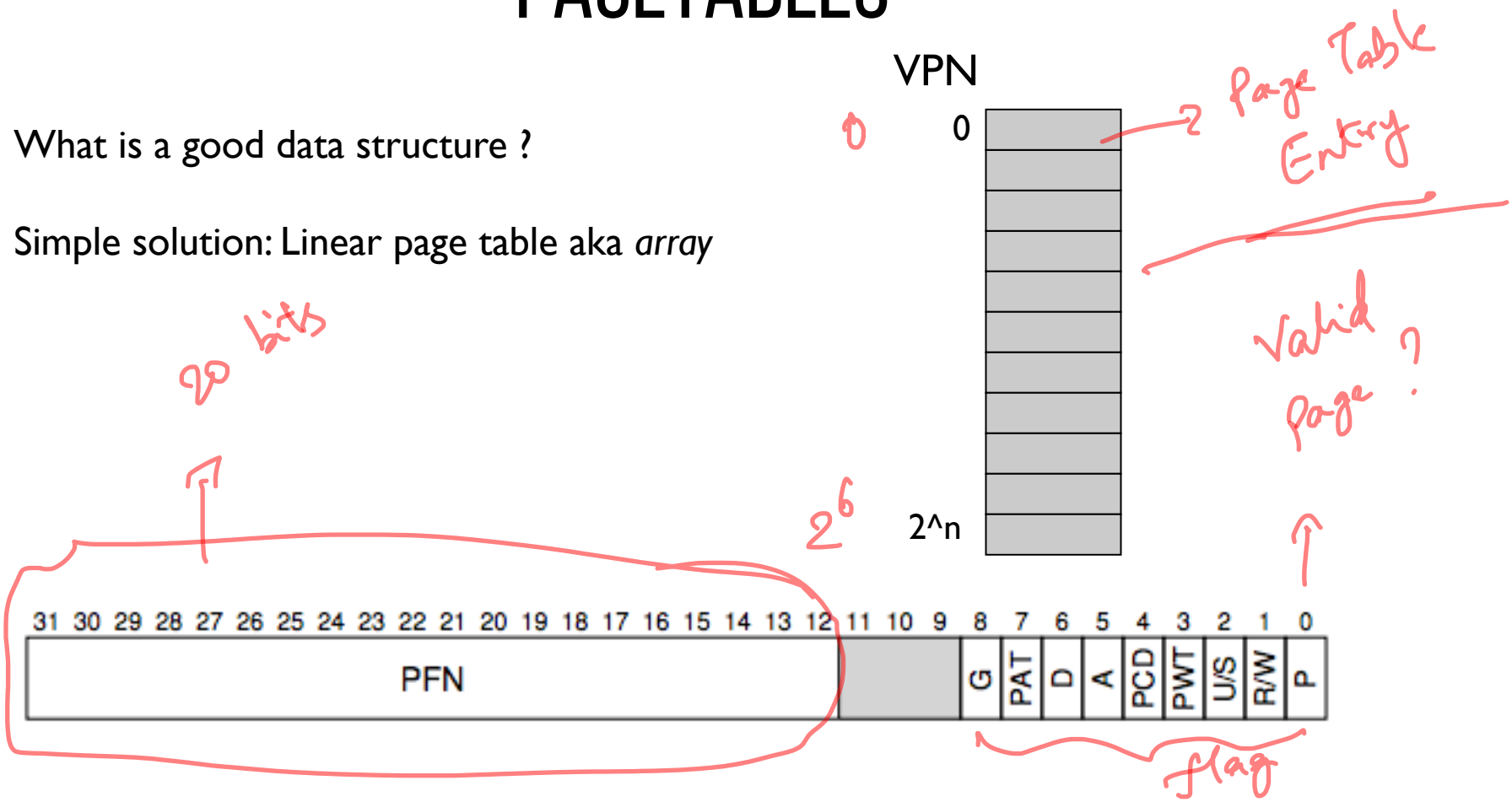
2^{64} Virtual addresses

How should OS translate VPN to PPN?

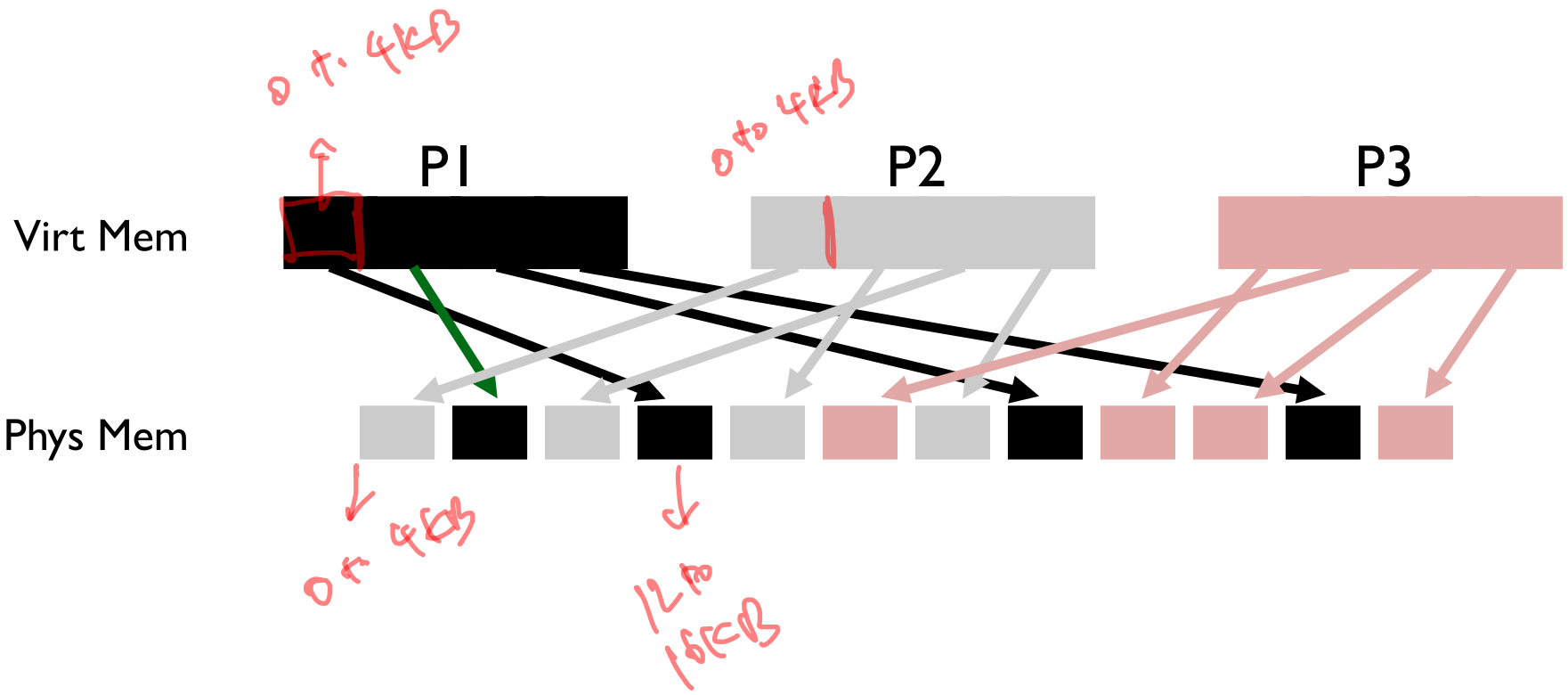
PAGETABLES

What is a good data structure ?

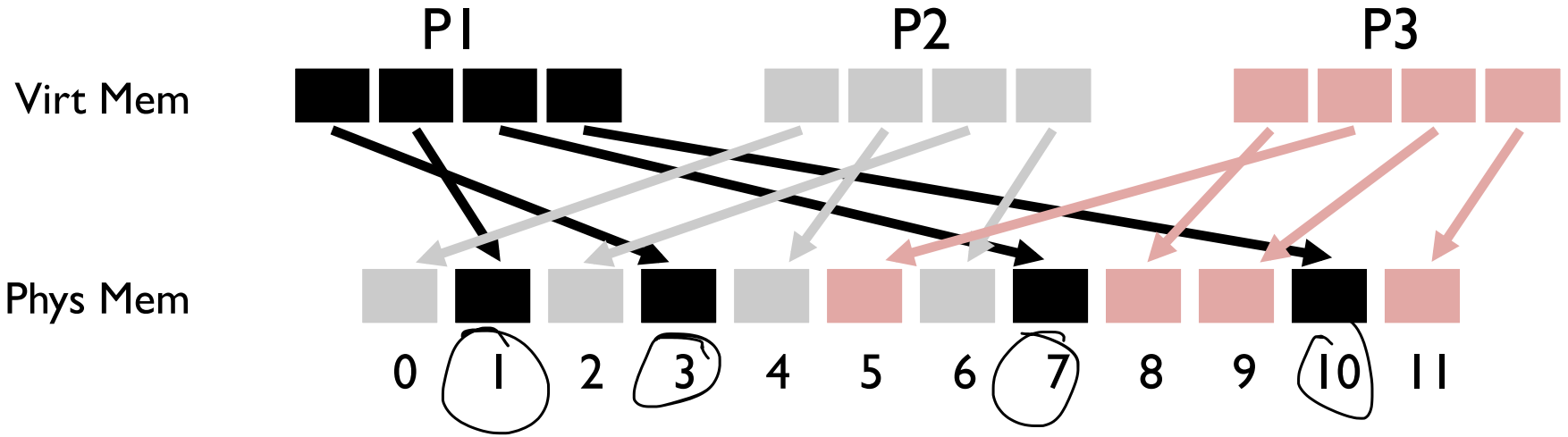
Simple solution: Linear page table aka *array*



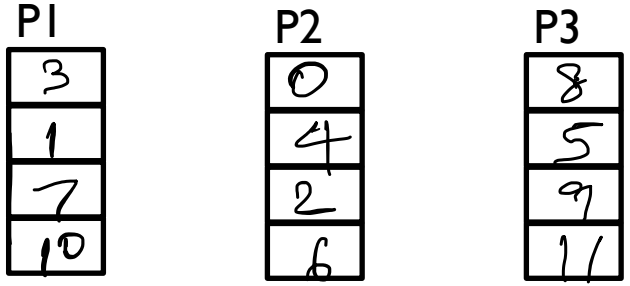
PER-PROCESS PAGETABLE



FILL IN PAGETABLE



Page Tables:



QUIZ: HOW BIG IS A PAGETABLE?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages = 12 bit offset
- assume 4 byte entries

Page table size = Number of entries x size entry

Number of entries = 20 bits of VPN

= 2^20 virtual pages

= 1 MB of virtual pages

x 4 bytes

= 4 MB

WHERE ARE PAGETABLES STORED?

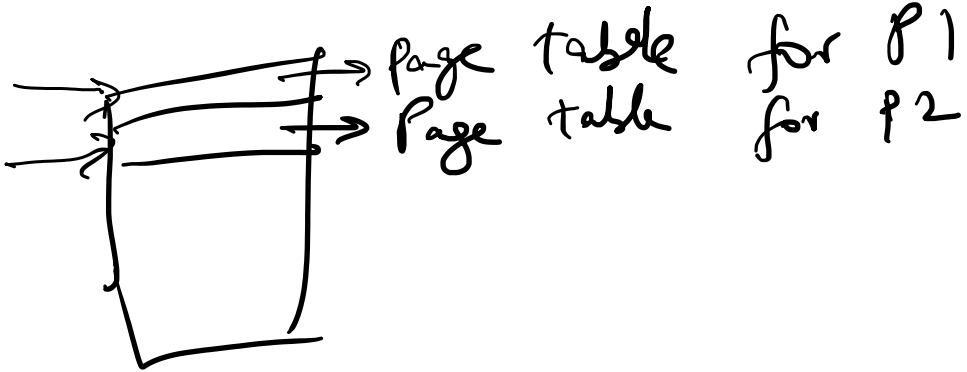
Implication: Store each page table in memory → kernel or OS

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

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

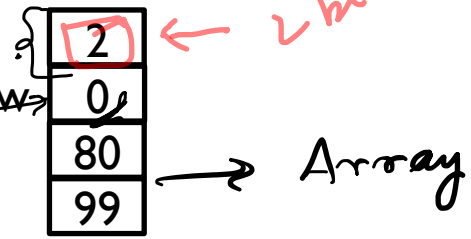
14 bit addresses

```
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

2 bit VPN

Simplified view of page table



Fetch instruction at logical addr 0x0010

VPN = 0

- Access page table to get ppn for vpn 0
- Mem ref 1: To the pagetable 0x5000
- Learn vpn 0 is at ppn 2 → 0x2010
- Fetch instruction at _____ (Mem ref 2)

[PPN][offset]

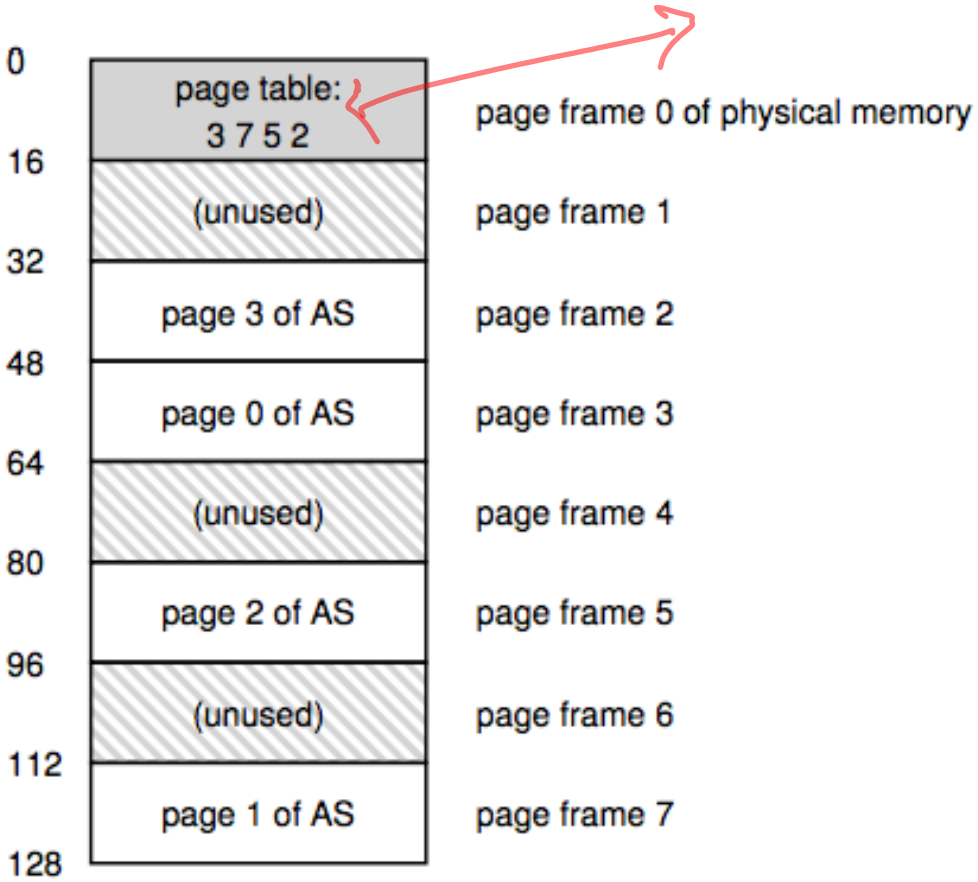
Exec, load from logical addr 0x1100

- Access page table to get ppn for vpn 1
- Mem ref 3: 0x5004

0x5000 + 4(1) =

- Learn vpn 1 is at ppn 0
- Movl from 0x0100 into reg (Mem ref 4)

SUMMARY:PAGING



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?**

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 ?

4 MB

PAGING 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 expensive step will we avoid next?

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

load 0x3004

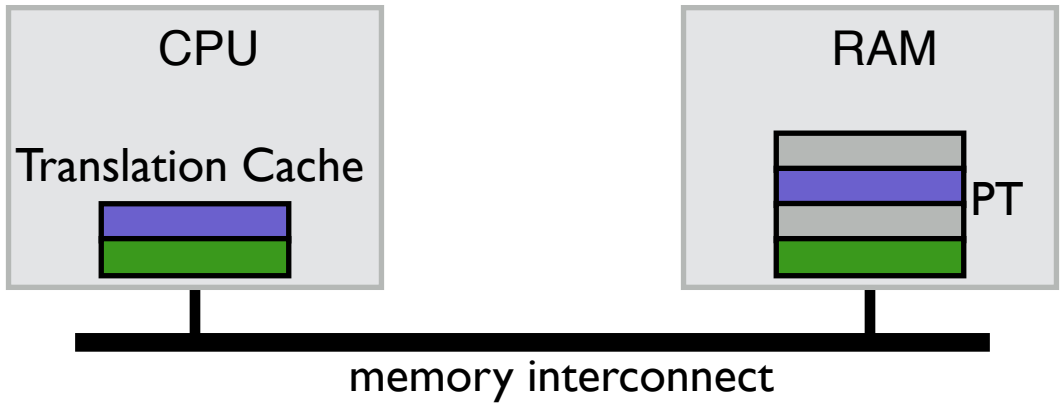
load 0x3008

load 0x300C

What physical addresses?

load 0x100C
load 0x7000
load 0x100C
load 0x7004
load 0x100C
load 0x7008
load 0x100C
load 0x700C

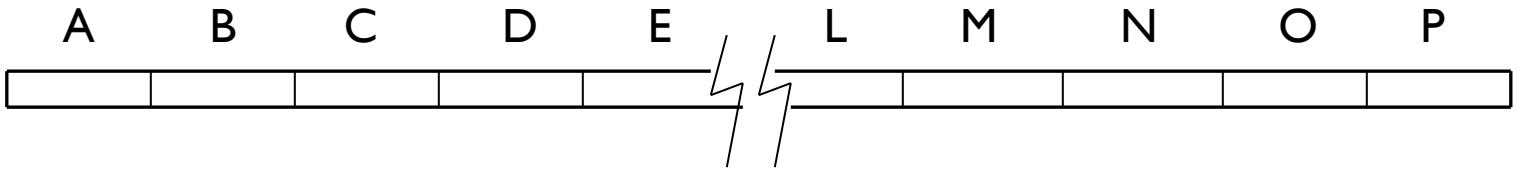
STRATEGY: CACHE PAGE TRANSLATIONS



TLB: TRANSLATION LOOKASIDE BUFFER

TLB ORGANIZATION

TLB 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'

Assume following virtual address stream:
load 0x1000

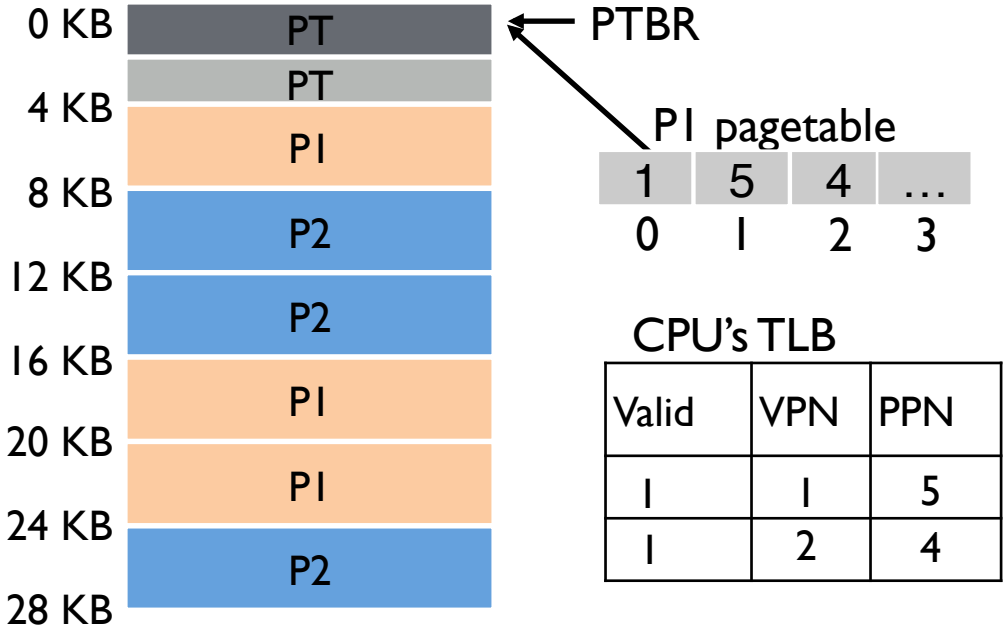
load 0x1004

load 0x1008

load 0x100C
…

What will TLB behavior look like?

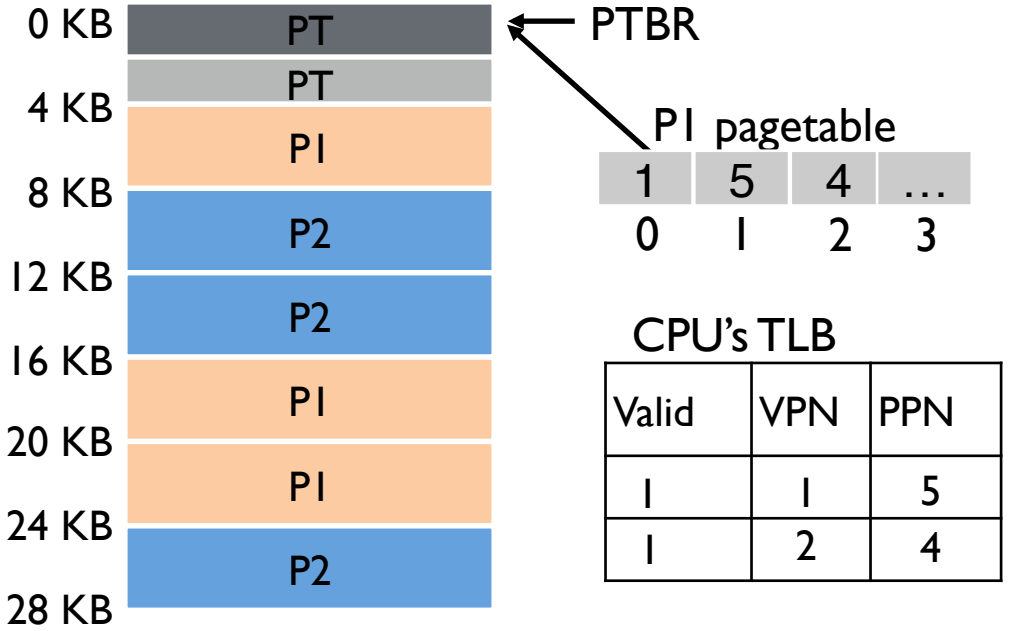
TLB ACCESSES: SEQUENTIAL EXAMPLE



CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

TLB ACCESSES: SEQUENTIAL EXAMPLE



CPU's TLB

Valid	VPN	PPN
1	1	5
1	2	4

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
load 0x2000	…
load 0x2004	load 0x0008
	load 0x4000 (TLB hit)
	load 0x4004

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

TLB PERFORMANCE WITH WORKLOADS

Sequential array accesses almost always hit in TLB

- Very fast!

What access pattern will be slow?

- Highly random, with no repeat accesses

WORKLOAD ACCESS PATTERNS

Workload A

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

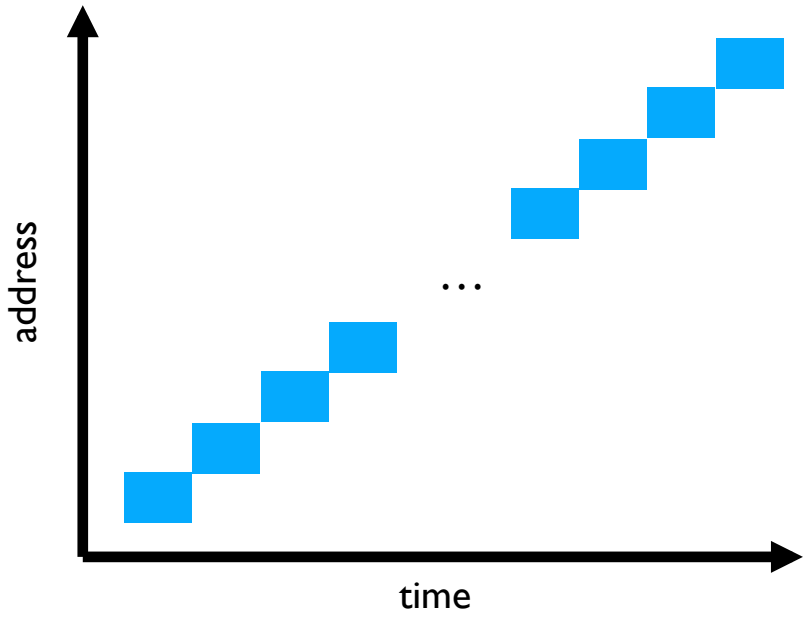
Workload B

```
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

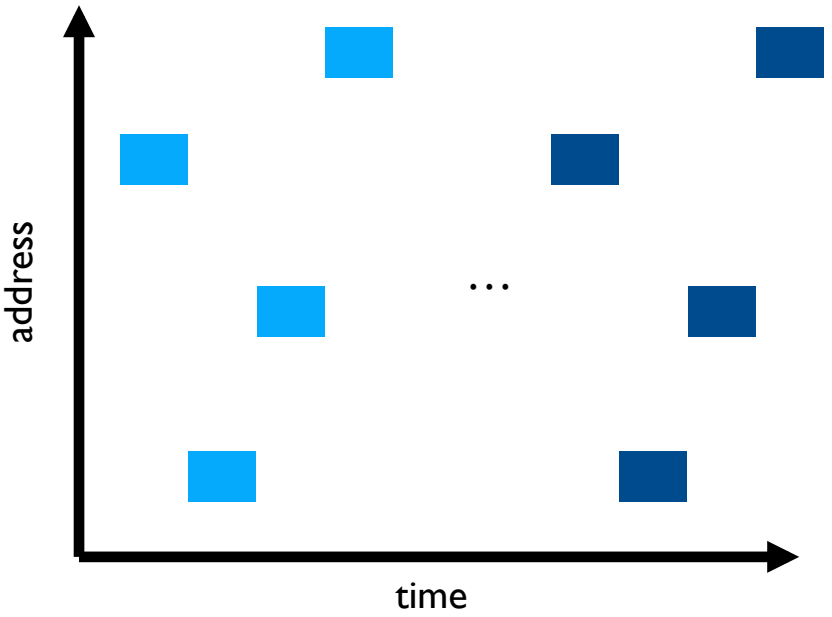
Spatial Locality

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
- 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 1b: Due tomorrow!

Project 2a: Out tomorrow

Discussion today: Process API, Project 2a

Next class: More TLBs and better pagetables!