MEMORY: TLBS, SMALLER PAGETABLES

Shivaram Venkataraman

CS 537, Fall 2024

ADMINISTRIVIA

- Project 3 in progress
- Discussion?

- Midterm 1

AGENDA / LEARNING OUTCOMES

Memory virtualization

- What are the challenges with paging ?
- How we go about addressing them?

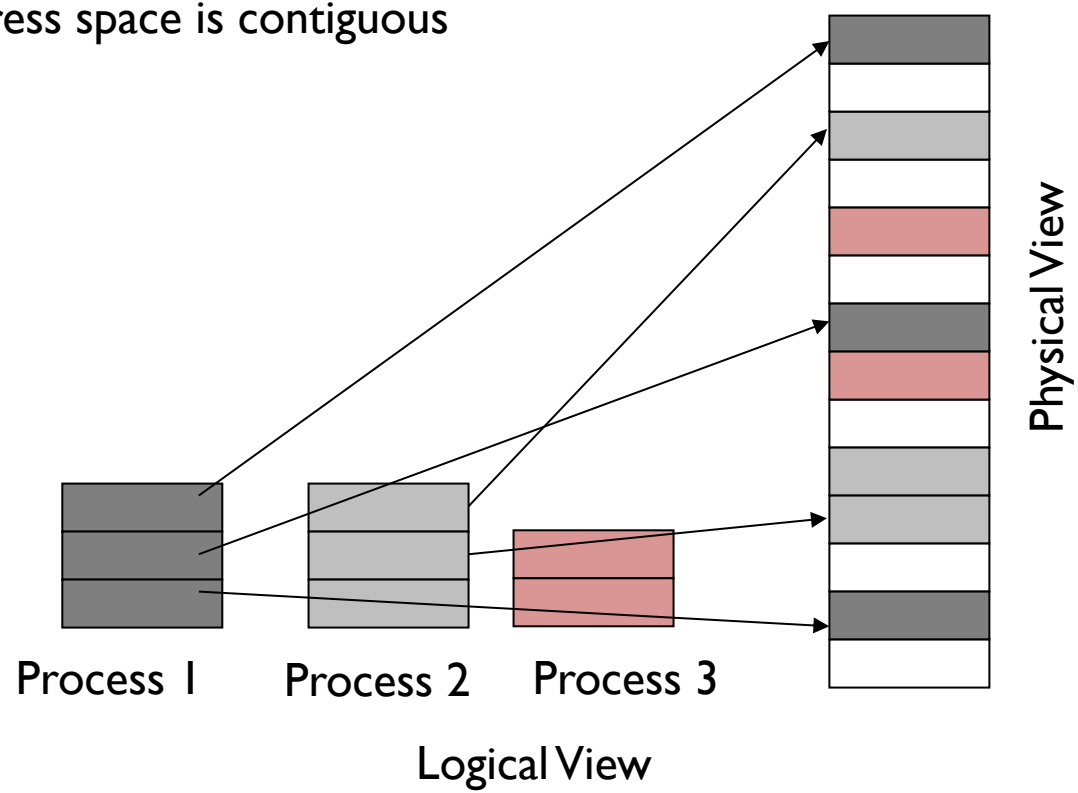
RECAP

PAGING

Goal: Eliminate requirement that address space is contiguous

Idea:
Divide address spaces and physical memory into fixed-sized pages

Example page size: 4KB



PAGING TRANSLATION STEPS

For each mem reference:

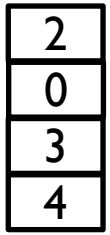
1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory

```
READ 0x1100
```

14 bit addresses

Assume PT is at phys addr 0x5000
Assume PTE's are 4 bytes
Assume 4KB pages – 12 bit offset

Simplified view of page table



PROS/CONS OF PAGING

Pros

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

Cons

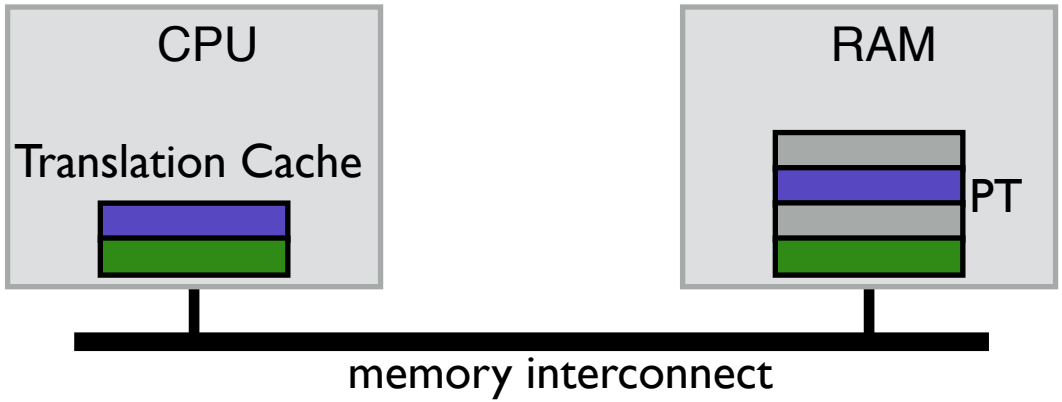
Additional memory reference

- 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 ?

STRATEGY: CACHE PAGE TRANSLATIONS



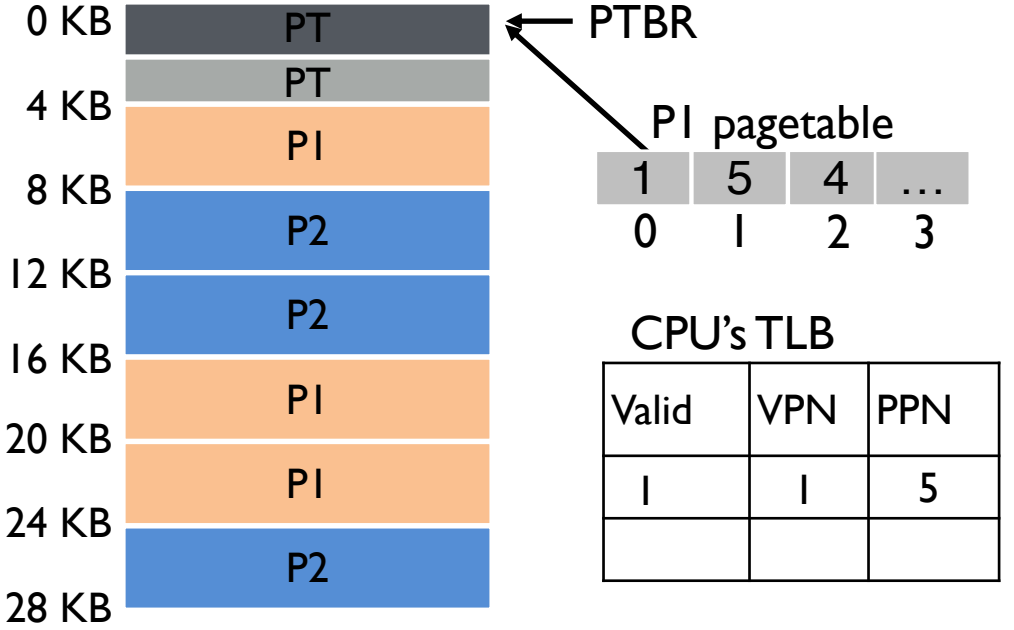
TLB Entry

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

TLB ACCESSES: SEQUENTIAL EXAMPLE



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

TLB: POLICIES

How to we replace entries in the TLB?

How do we handle context switches?

WORKLOAD ACCESS PATTERNS

Workload A

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

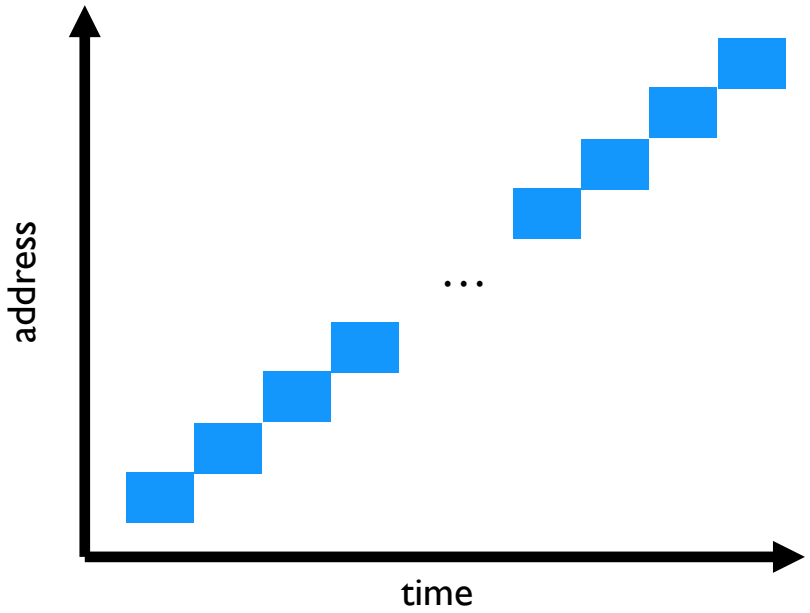
Workload B

```
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

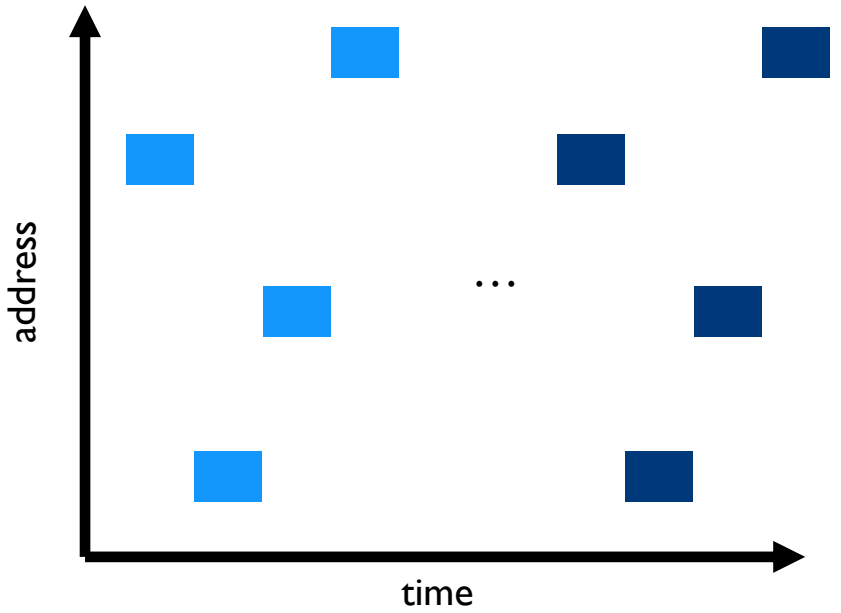
Spatial Locality

Sequential Accesses



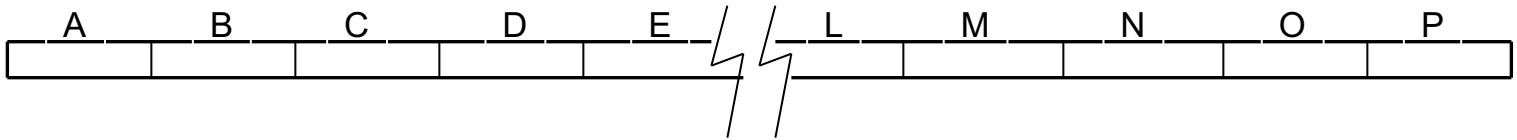
Temporal Locality

Repeated Random Accesses

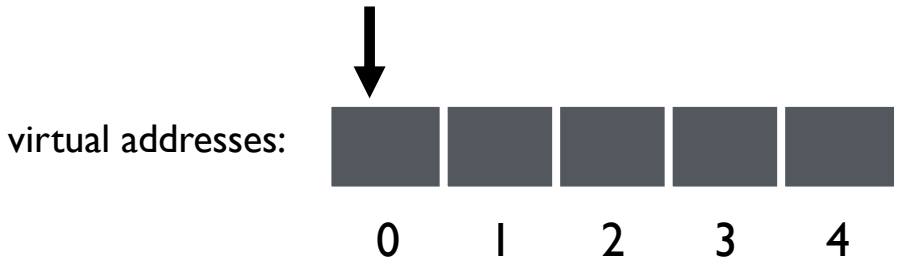


TLB REPLACEMENT POLICIES

LRU: evict Least-Recently Used TLB slot when needed



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 will TLB contents be over time?
How will TLB perform?

TLB REPLACEMENT POLICIES

LRU: evict Least-Recently Used TLB slot when needed

Random: Evict randomly chosen entry

Sometimes random is better than a “smart” policy!

CONTEXT SWITCHES

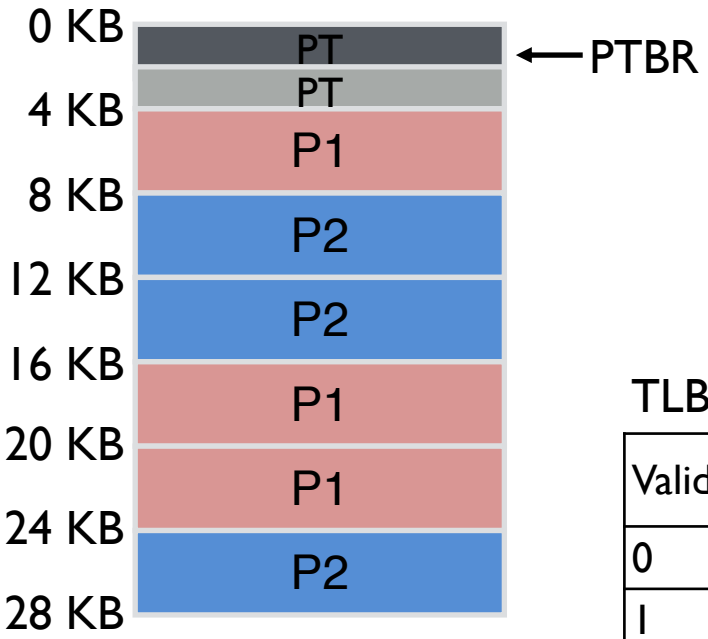
What happens if a process uses cached TLB entries from another process?

1. Flush TLB on each switch

   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

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

Architectures can have multiple TLBs

- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 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

In different systems, hardware or OS handles TLB misses

TLBs increase cost of context switches

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

DISADVANTAGES OF PAGING

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 ?

QUIZ 6

https://tinyurl.com/cs537-fa24-q6

Consider a 32-bit address space with 4 KB pages.

Bits to represent the offset within a page?

Number of virtual pages?

address space size 16k
phys mem size 128k
page size 4k

The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.

Page Table (from entry 0 down to the max size)
  0x0010
  0x8000
  0x0004
  0x800a

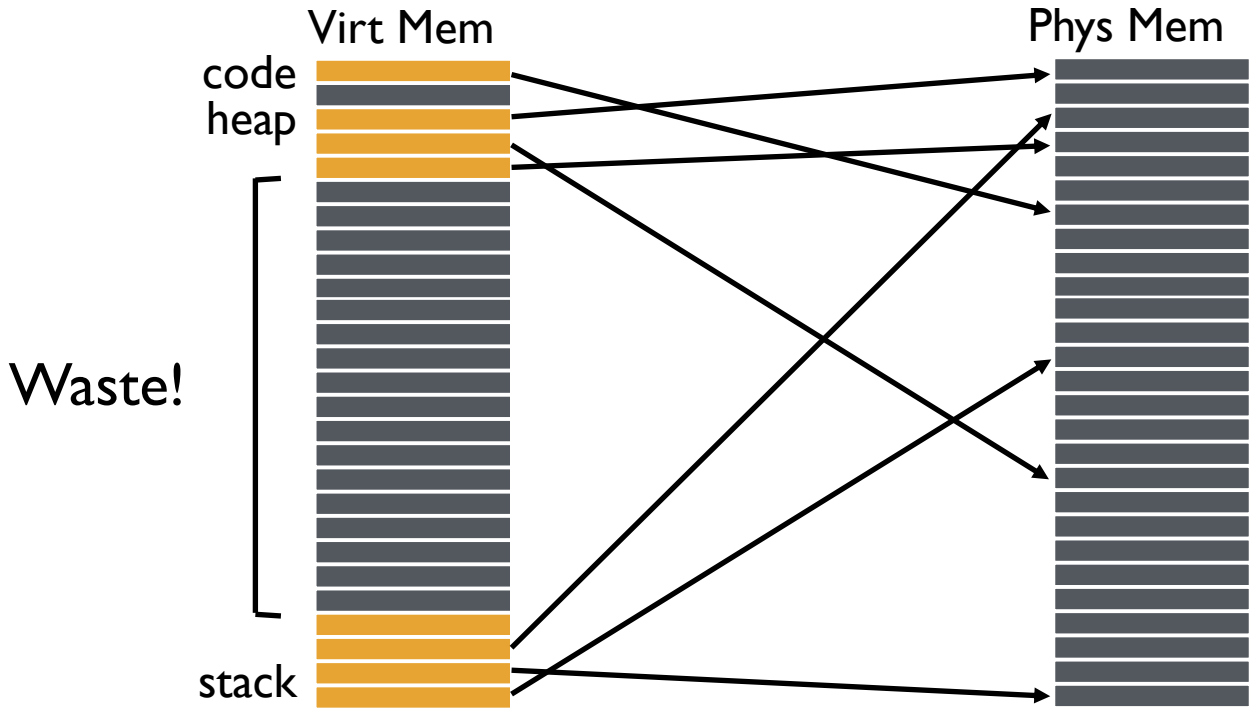
Virtual Address: 0x0837

Virtual Address: 0x164c

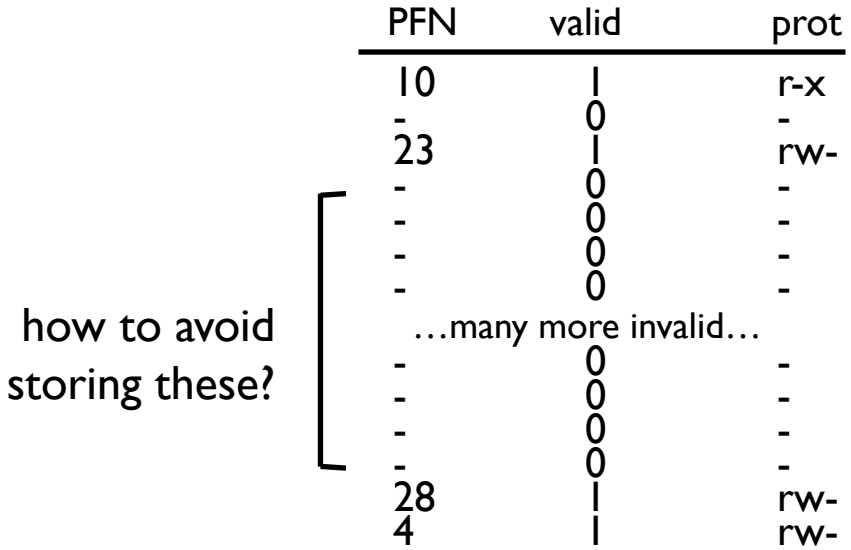
Virtual Address: 0x384d

SMALLER PAGE TABLES

WHY ARE PAGE TABLES SO LARGE?



MANY INVALID PT ENTRIES



AVOID SIMPLE LINEAR PAGE TABLES?

Use more complex page tables, instead of just big array

Any data structure is possible with software-managed TLB

- Hardware looks for vpn in TLB on every memory access
- If TLB does not contain vpn, TLB miss
  - Trap into OS and let OS find vpn->ppn translation
  - OS notifies TLB of vpn->ppn for future accesses

OTHER APPROACHES

1. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
2. Inverted Pagetables

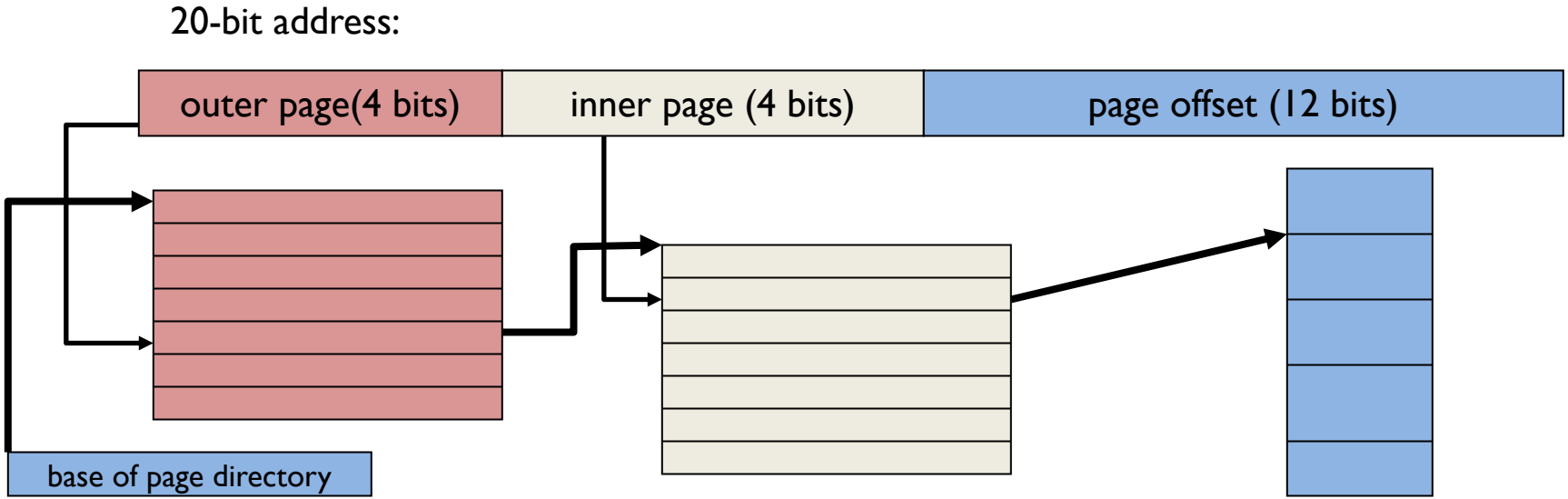
MULTILEVEL PAGE TABLES

Goal: Allow page table to be allocated non-contiguously

Idea: Page the page tables

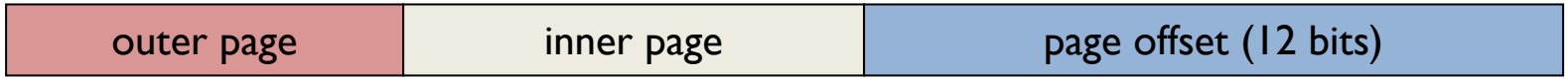
- Creates multiple levels of page tables; outer level "page directory"
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)

MULTILEVEL PAGE TABLES



ADDRESS FORMAT FOR MULTILEVEL PAGING

30-bit address:



How should logical address be structured? How many bits for each paging level?

Goal?

- Each inner page table fits within a page
- PTE size * number PTE = page size

  Assume PTE size = 4 bytes

  Page size = 2^12 bytes = 4KB

→ # bits for selecting inner page =

Remaining bits for outer page:

- 30 – ___ – ___ = ___ bits

MULTILEVEL TRANSLATION EXAMPLE

page directory

PPN	valid
0x3	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x92	1

page of PT (@PPN:0x3)

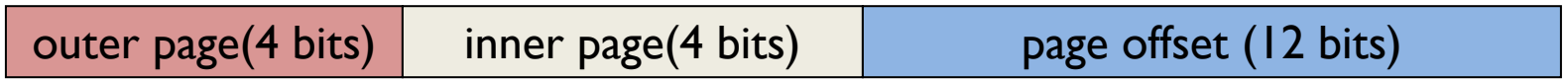
PPN	valid
0x10	1
0x23	1
-	0
-	0
0x80	1
0x59	1
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

page of PT (@PPN:0x92)

PPN	valid
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
0x55	1
0x45	1

translate 0x01ABC

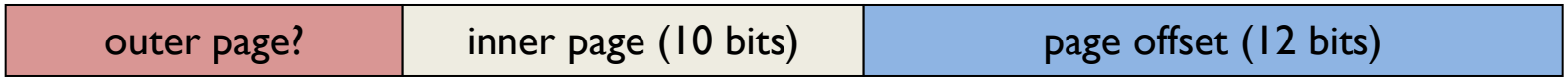
20-bit address:



PROBLEM WITH 2 LEVELS?

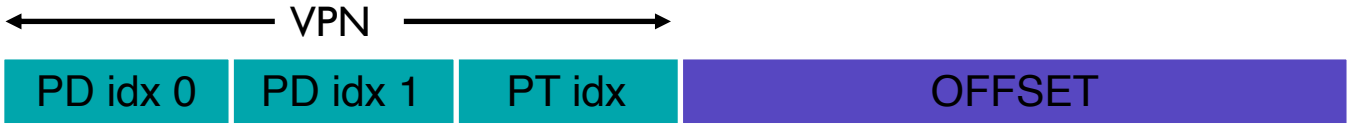
Problem: page directories (outer level) may not fit in a page

64-bit address:



Solution:

- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.



How large is virtual address space with 4 KB pages, 4 byte PTEs,
(each page table fits in page)

4KB / 4 bytes → 1K entries per level

1 level:

2 levels:

3 levels:

FULL SYSTEM WITH TLBS

On TLB miss: lookups with more levels more expensive

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

ASID	VPN	PFN	Valid
211	0xbb	0x91	1
211	0xff	0x23	1
122	0x05	0x91	1
211	0x05	0x12	0

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl $0x3, %edi

(c) 0x0519: movl %edi, 0xFF10

INVERTED PAGE TABLE

Only store entries for virtual pages w/ valid physical mappings

Naïve approach:

- Search through data structure <ppn, vpn+asid> to find match
- Too much time to search entire table

Better:

- Find possible matches entries by hashing vpn+asid
- Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB

SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure

- Inverted page tables (hashing)

If Hardware handles TLB miss, page tables must follow specific format

- Multi-level page tables used in x86 architecture
- Each page table fits within a page

NEXT STEPS

Project 3: In progress

Next class: Swapping!