CS 423
Operating System Design: TLBs and Smaller Page Tables
02/19

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Logistics

MP0 grades released (on Canvas)

MP1 due 2/28 - Utilize TA office hours if you need help with MP1

AGENDA / LEARNING OUTCOMES

Continue discussion on TLBs - solves the performance problem with address translation

Smaller page tables: tackle memory overheads of page tables

RECAP

Disadvantages of Paging

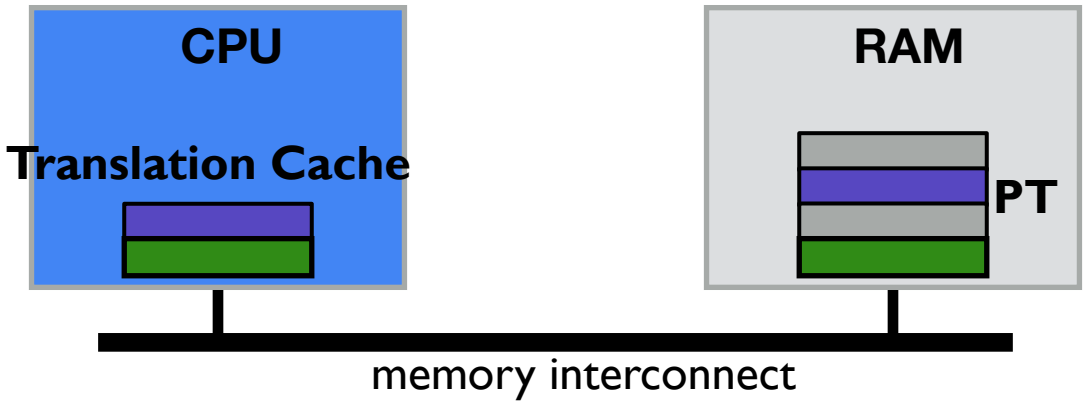
Additional memory reference to page table Very inefficient

- Every memory reference results in two references
- One to PT and one to actual address
- How to solve this problem? TLB

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
- Better ways to save space…

TLB: CACHE PAGE TRANSLATIONS



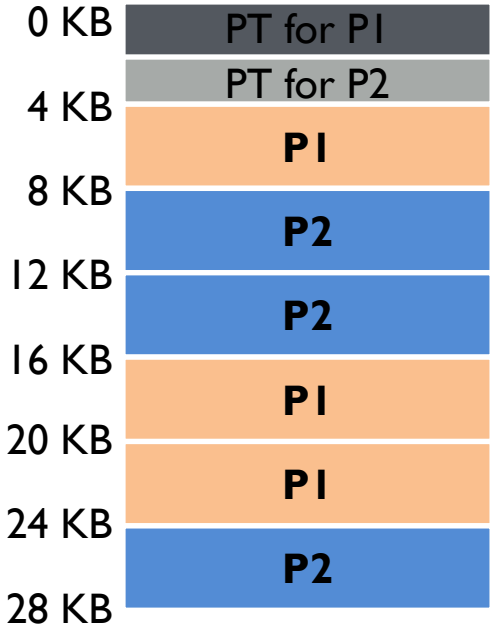
TLB: TRANSLATION LOOKASIDE BUFFER

PAGE TRANSLATION WITH TLB

For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. check TLB for **VPN**
   if miss:
      3. calculate addr of **PTE** (page table entry)
      4. read **PTE** from memory, add to TLB
5. extract **PFN** from TLB (page frame num)
6. build **PA** (phys addr)
7. read contents of **PA** from memory

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

HW AND OS ROLES

Who handles TLB hit?

Who handles TLB miss?

TLB MISS - HW AND OS ROLES

If HW, then HW must know where the PT is in memory

- CR3 in x86
- Page table structure agreed upon between OS and HW
- Hardware "walks" page table and fills in TLB

If OS ("software managed TLB")

- HW traps into OS upon TLB miss
- OS walks page table (any data structure chosen by OS)
- OS fills in TLB (TLB operations are privileged instructions)

END RECAP

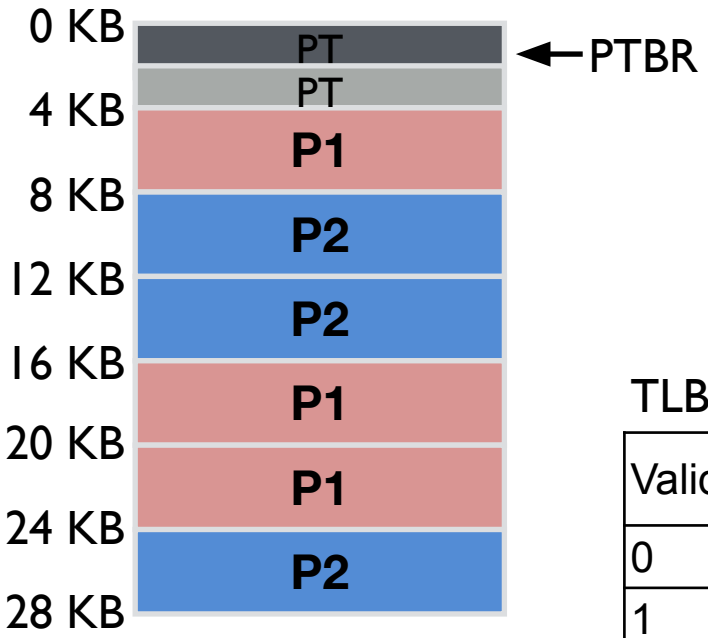
CONTEXT SWITCHES

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

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"

UNMAPPING MEMORY AND TLB

A process may unmap certain regions of virutal memory (e.g., using 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 process – `void flush_tlb_mm(struct mm_struct *mm)`

Works but inefficient –But useful when doing fork and exec

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

Inline assembly for GCC (from Linux kernel source):

```
static inline void __native_flush_tlb_single(unsigned long addr) {
   asm volatile("invlpg (%0)" ::"r" (addr) : "memory");
}
```

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 (if enough TLB entries)

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

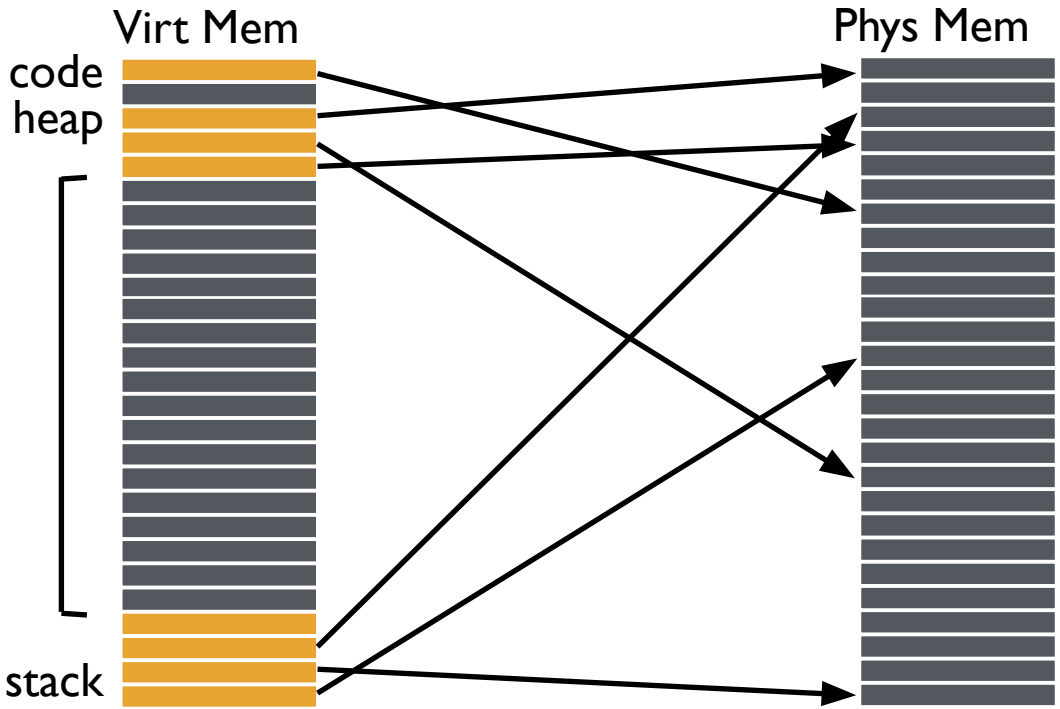
Solve the Space Problem Now

Storage for page tables may be substantial

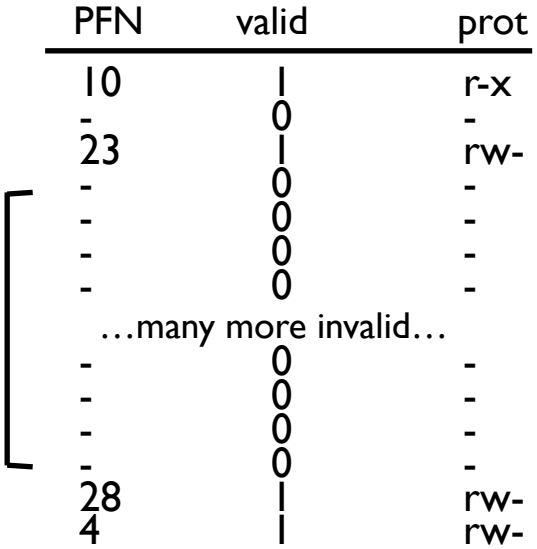
- Simple page table: Requires PTE for all pages in address space
- Better ways to save space…

Why are Page Tables so Large?

Unallocated
Unused
But PT entries exist



MANY INVALID PTES



how to avoid storing these?

Problem: linear PT must still allocate PTE for each page (even unallocated ones)

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

Remember - page tables are just data structures! One can use any data strucuture

What about hardware managed TLBs?

OTHER APPROACHES

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

VALID PTES ARE CONTIGUOUS

how to avoid storing these?

PFN	valid	prot
10	1	r-x
-	0	-
23	1	rw-
-	0	-
-	0	-
-	0	-
-	0	-
…many more invalid…		
-	0	-
-	0	-
-	0	-
-	0	-
28	1	rw-
4	1	rw-

Note “hole” in addr space:
valids vs. invalids are clustered

How did OS avoid allocating holes in phys memory?

Combine Paging and Segmentation

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages.

Logical address divided into three portions

Implementation

- Each segment has a page table
- Track base physical address and bounds of the **page table** per segment
  - This is different from original segmentation

Paging and Segmentation - Chat for 2 mins

seg	base	bounds	R W
0	0x002000	0xff	1 0
1	0x000000	0x00	0 0
2	0x001000	0x0f	1 1

0x002070 read:
0x202016 read:
0x104c84 read:
0x010424 write:
0x210014 write:
0x203568 read:

...
0x01f
0x011
0x003
0x02a
0x013
...
0x00c
0x007
0x004
0x00b
0x006
...

0x001000

0x002000

Bounds: # PTE entries

Advantages of Paging and Segmentation

Advantages from using Segments

- Decreases size of page tables. If segment not used, no need for page table

Advantages from using Pages

- No external fragmentation
- Segments can grow without any reshuffling
- Can run process with some pages swapped to disk

Disadvantages of Paging and Segmentation

Potentially large page tables (for each segment)

- Must allocate page table for each segment *contiguously*
- Page table size?
  - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:
= Number of entries * size of each entry
= Number of pages * 4 bytes
= 2^18 * 4 bytes = 2^20 bytes = 1 MB!

OTHER APPROACHES

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

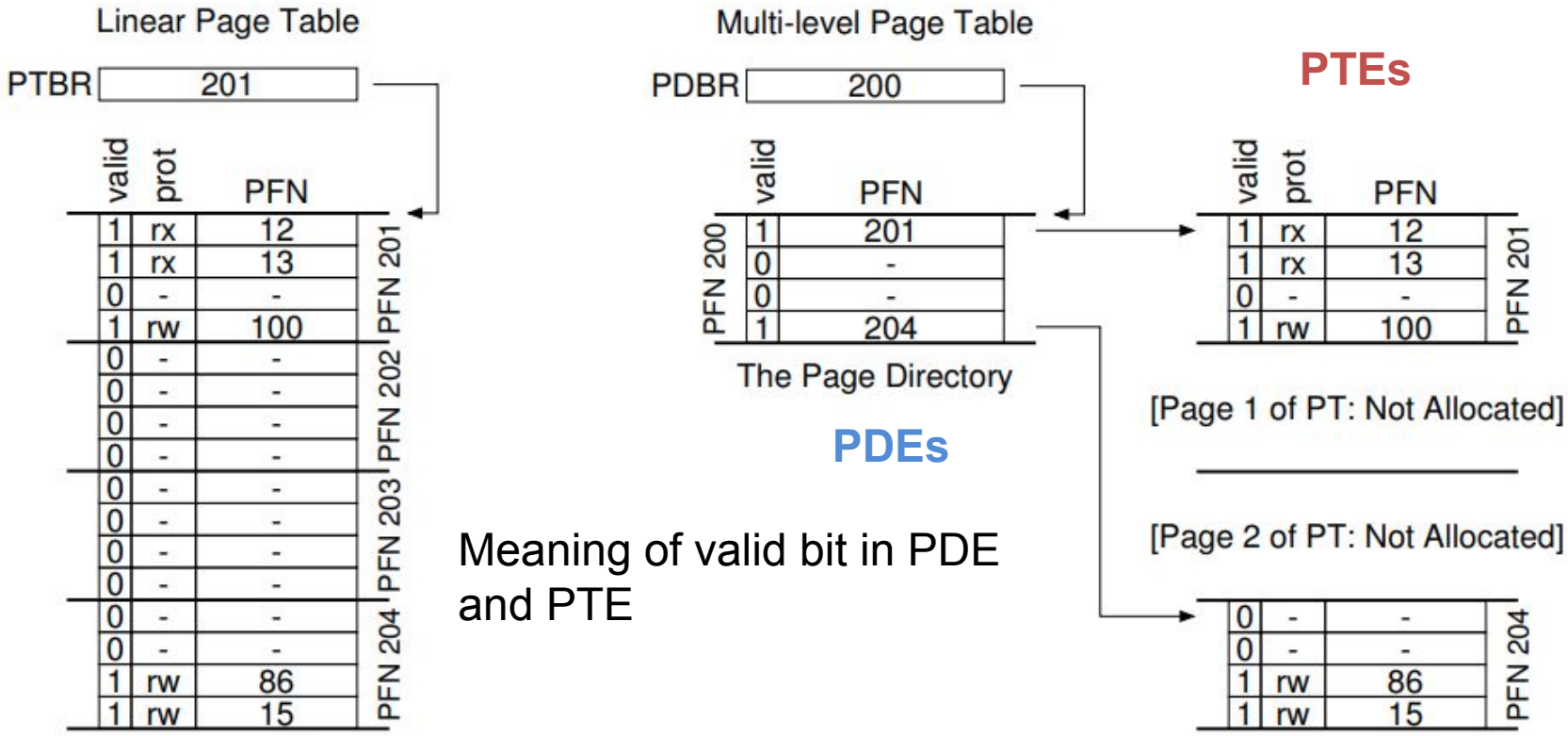
Multilevel Page Tables

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 Table – Key Idea



Meaning of valid bit in PDE and PTE

Multilevel Page Tables

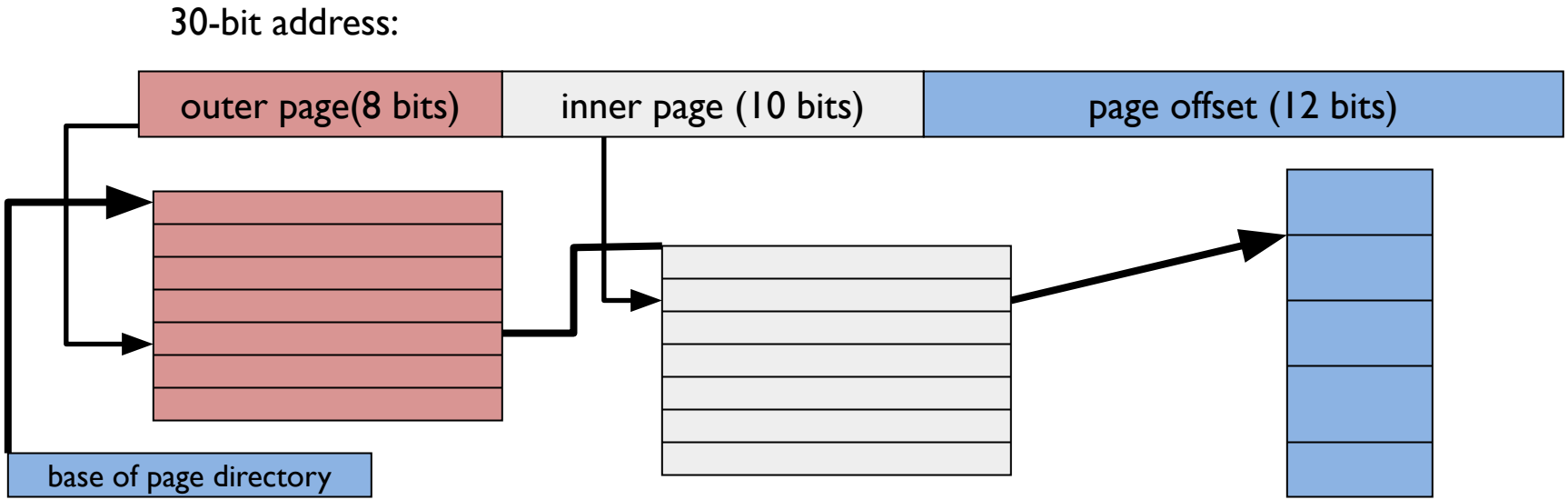
Chat for 1 minute with your neighbors…

Can pages of the PT be dynamically relocated to a different physical location?

Can the PD be dynamically relocated to a different physical location?

How many memory accesses on a TLB miss with this 2-level PT?

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

  Each inner page can have 1024 PTE

  # bits for inner page = 10

Remaining bits for outer page:

- 30 – 12 – 10 = 8 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

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

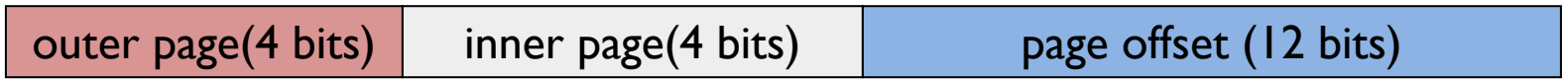
translate 0x01ABC

Physical address?

Look PD[0]

Look PT[1] → arrive at page 0x23

Concat ABC to arrive at 0x23ABC



20-bit address:

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

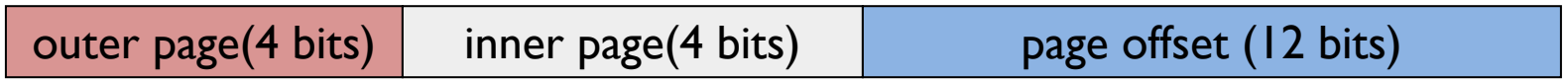
page of PT (@PPN:0x92)

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

Translate

VA: 0x04000

VA: 0xFEED0



20-bit address:

PROBLEM WITH 2 LEVELS?

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

Consider 30 bit address with 512-byte pages

We need 9 bits for offset, leaving 21 bits for VPN

Remember our goal: each inner page should fit within a page

So how many PTE per page? With 512-byte pages and 4-byte PTE, we can have 128 entries → this means 7 bits for inner page, leaving 14 bits for outer page (or directory) → 2^{14} PDEs

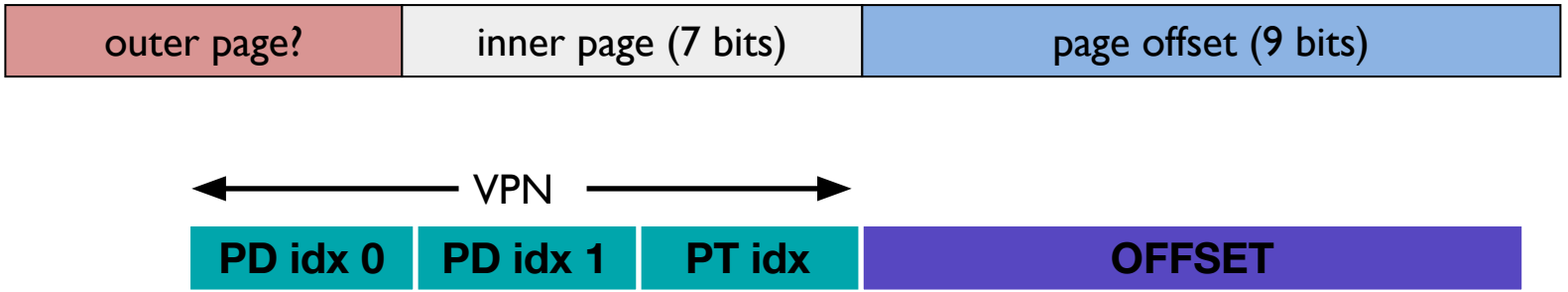
Assume 4-byte PDE, then PD itself will span _____ pages?

PD cannot be contained in one page now!

PROBLEM WITH 2 LEVELS?

Solution: page the page directory!

Add another level of page directory that points to PD pages



Can keep going recursively…

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

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)

1. 0xaa: (TLB miss -> 3 for addr trans) + 1 instr fetch
   0x11: (TLB miss -> 3 for addr trans) + 1 movl

2. 0xbb: (TLB hit -> 0 for addr trans) + 1 instr fetch from 0x9113

3. 0x05: (TLB miss -> 3 for addr trans) + 1 instr fetch
   0xff: (TLB hit -> 0 for addr trans) + 1 movl into 0x2310

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

Used in IBM PowerPC

Managing inverted page table requires software-controlled TLB (although doing some these in HW is possible)

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

SWAPPING

Motivation

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

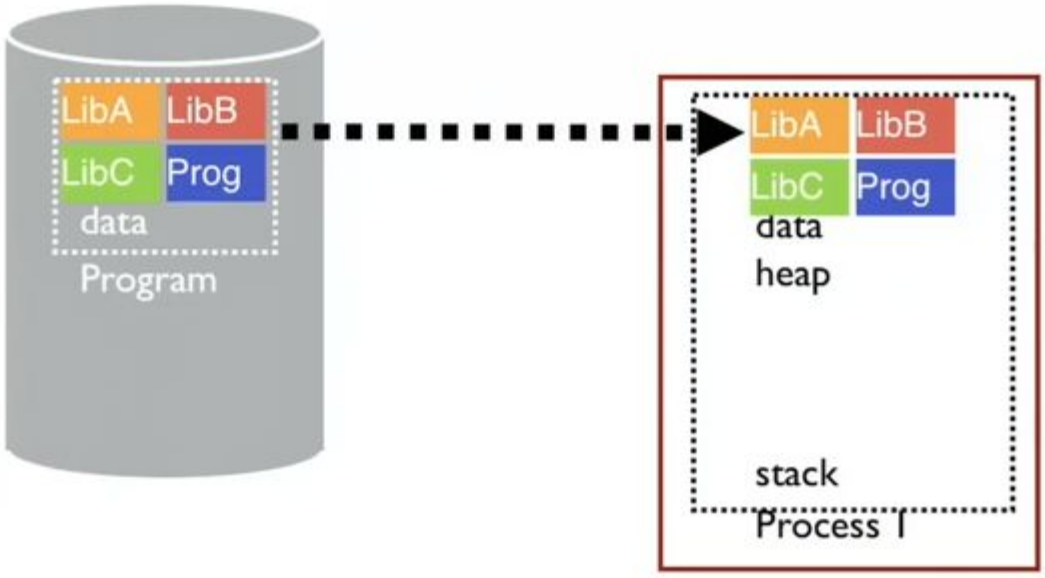
User code should be independent of amount of physical memory

- Correctness, performance

Virtual memory: OS provides illusion of more physical memory

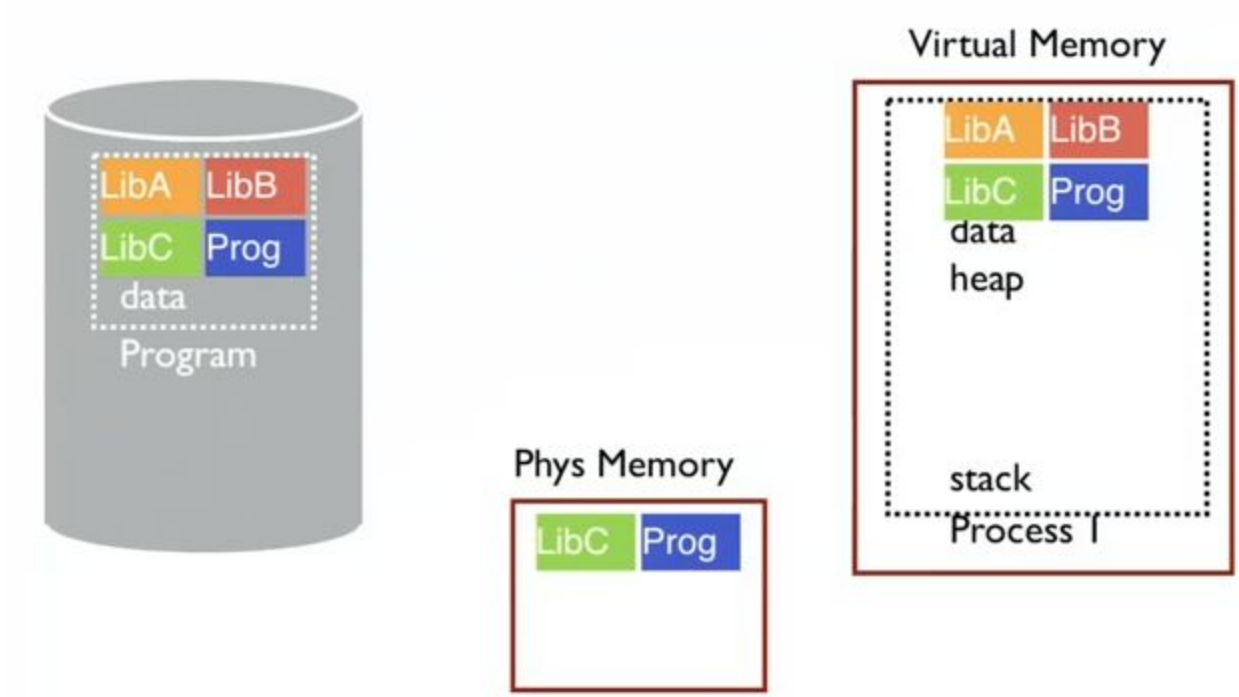
Why does this work?

- Relies on key properties of user processes (workload) and machine architecture (hardware)



Code: many large libraries, some of which are rarely/never used

How to avoid wasting physical pages to store rarely used virtual pages?


LibA
LibB
LibC
Prog
data
Program
Phys Memory
LibC
Prog
Virtual Memory
LibA
LibB
LibC
Prog
data
heap
stack
Process 1

NEXT STEPS

Next class: Swapping!