MEMORY MANGEMENT: SMALLER PAGETABLES

Kai Mast
CS 537
Fall 2022

ANNOUCEMENTS

- Midterm on 10/19 at 7:25pm
  - Will cover everything up to 10/13
  - You can bring a cheatsheet (one piece of paper)

RECAP: SEGMENTATION

From Homework 4

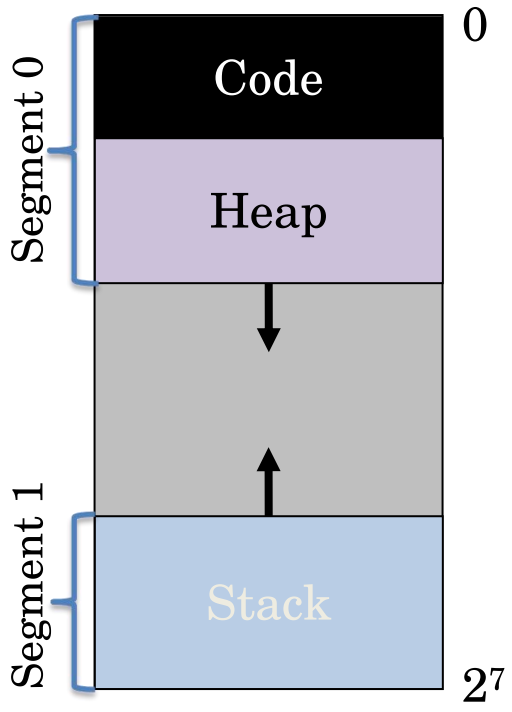
Virtual Address Space 128bytes (2^7)

Two Segments
- 0: Code+Heap (grows up)
- 1: Stack (grows down)

Virtual Address Length 7 bits

How many bits for segment number? 1 bit

Maximum Stack Size? 64 bytes (2^6)



RECAP: SEGMENTATION

From Homework 4:
Virtual Address Space 128bytes (2^7)

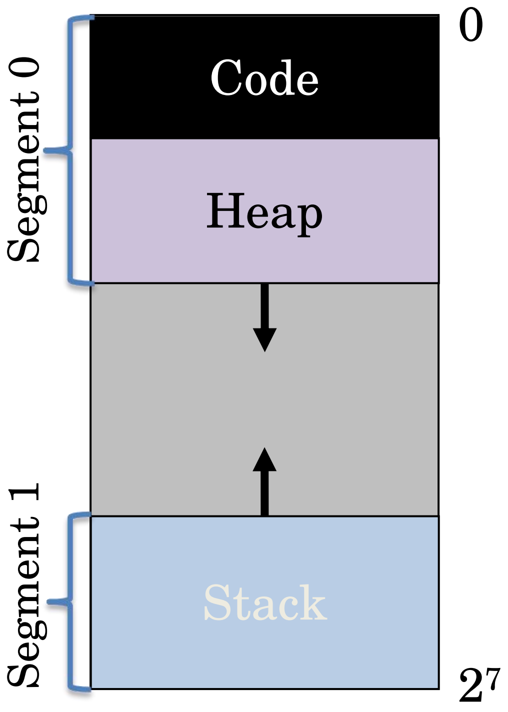
Virtual Address: 115 (0x73)

Hex	**7**				**3**			
Bits	-	1	1	1	0	0	1	1

Segment No (1) Offset (51)

Segment 1: base=300 bounds=32

Physical Address: 300 - 64 + 51 = 287
(base - max_size + offset)



PAGING TRANSLATION STEPS

For each memory reference:

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

PAGING TRANSLATION STEPS WITH TLB

For each memory reference:

1. extract **VPN** (virt. page num) from **VA** (virt. address)
2. check TLB for **VPN**

    if miss:

        3. calculate address of **PTE** (page table entry)
        4. read **PTE** from memory, replace some entry in TLB

5. extract **PFN** (physical frame number) from TLB
6. build **PA** (physical address)
7. read contents of **PA** from memory

CONTEXT SWITCHES

What happens if a process uses TLB entry from another process?

Access some other processes address space (no protection)

Solutions:

1. Flush TLB on each context switch (privileged instruction)

Poor performance: lose all recently cached translations, increases miss rate

2. Track which TLB entries are for which process
   - Address Space Identifier (ASID) – similar to PID (remember in PCB)
   - Must match ASID for TLB entry to be used

TLB MISSES: HW & OS ROLES

Who Handles TLB Hit? **H/W**

Who Handles TLB Miss? **H/W or OS**

For H/W:
H/W must know where page tables are stored in memory
- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW “walks” known pagetable structure and fills TLB

TLB MISSES: HW & OS ROLES

Who Handles TLB Miss? **H/W or OS**

For OS:

- Common on Restricted Intrstruction Set Circuits (RISC)

CPU traps into OS upon TLB miss

- “Software-managed TLB”

OS interprets page tables as it chooses; any data structure possible

- Modifying TLB entries is privileged instruction

DISADVANTAGES OF PAGING REVISITED

Page table itself stored in memory
- Will not if in MMU or CPU registers
- Need to touch main memory twice for every memory access

Solution: Use TLB (*Last Lecture*)

Page tables are too big
- Array of size N, where N=(MemorySize)/(PageSize)
- Need one page table per process

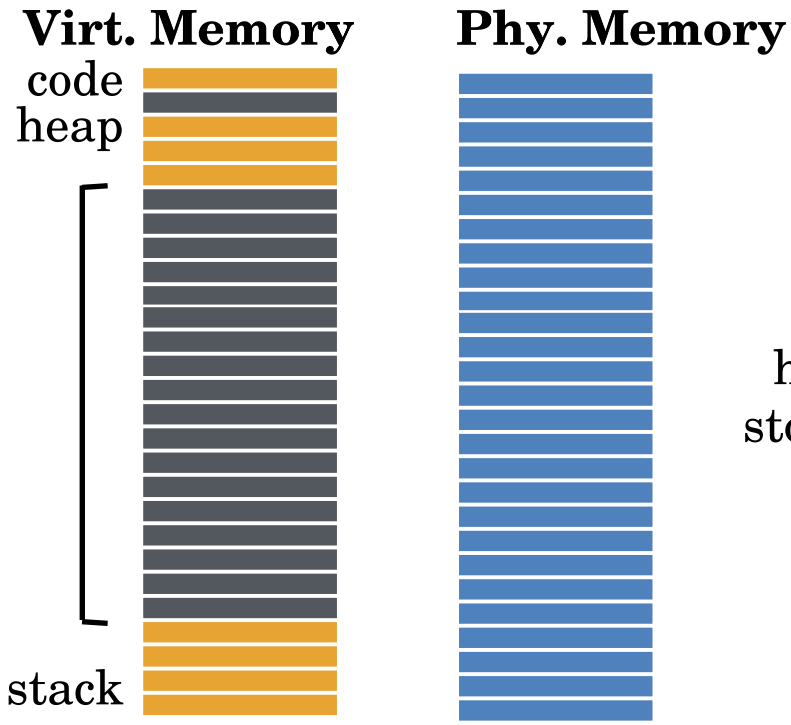
Solution: Find a more efficient data structure *(Today's Lecture)*

SIZE OF LINEAR PAGE TABLES

How big is each page table?

1. PTEs are **2 bytes**, and **32** possible virtual page numbers	32*2 bytes = 64 bytes
2. PTEs are **2 bytes**, virtual addrs **24 bits**, and **16 byte** pages	2^{21} bytes (2 MB)
3. PTEs are **4 bytes**, virtual addrs **32 bits**, and **4 KB** pages	2^{22} bytes (4 MB)
4. PTEs are **8 bytes**, virtual addrs **64 bits**, and **16 KB** pages	2^{53} bytes (8PB)

WHY ARE LINEAR PAGE TABLES SO LARGE?



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-

Problem: Linear page tables must allocate PTE for each page in address space (even for unallocated pages)

AVOID SIMPLE LINEAR PAGE TABLES?

Use more complex page tables, instead of just large array

With software-managed TLB any data structure is possible

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->PFN translation
  - OS notifies TLB of VPN->PFN for future accesses

Other structures are possible for hardware to walk as well…

OTHER APPROACHES

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

VALID PAGE TABLE ENTRIES ARE CONTIGUOUS

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-

how to avoid storing these PTEs?

Note the "hole" in the address space: valid and invalid pages are clustered

What is another mechanism that avoids holes in the address space?

Segmentation

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

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

Implementation

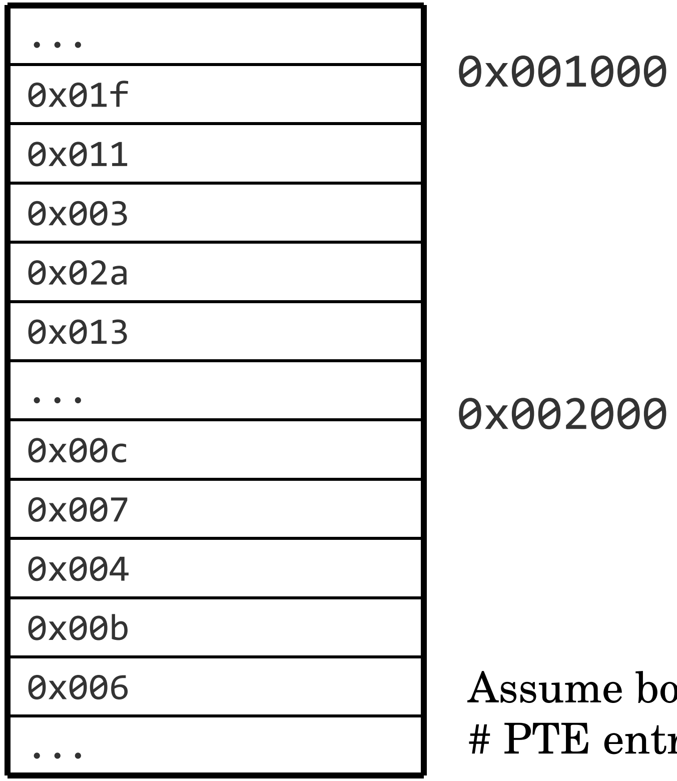
- Each segment has a page table
- Each segment tracks base (physical addr.) and bounds of the **page table**

SEGMENTED PAGE TABLES

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

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

0x002070 read:	0x004070
0x202016 read:	0x003016
0x104c84 read:	error (out-of-bounds)
0x010424 write:	error (read-only)
0x210014 write:	error (out-of-bounds)
0x203568 read:	0x02a568



Assume bounds:
# PTE entries

ADVANTAGES OF SEGMENTED PAGE TABLES

Advantages of Segments

- Supports sparse address spaces.
  - Decreases size of page tables (only need PTEs for allocated portions)
  - If segment not allocated, no need for page table

Advantages of Pages

- No external fragmentation
- Segments can grow without any compaction or page movement
- Can run process when some pages are swapped to disk (next lecture)

Advantages of Both

- Increases flexibility of sharing: Share either single page or entire segment

SHARING W/ SEGMENTED PAGE TABLES

seg # (4 bits)	page number (8 bits)	page offset (12 bits)

P1: 0x802070 read

P2: 0x902070 read

P2: 0xa00100 read

P1

seg	base	bounds	R W
8	0x002000	0xff	1 0
9	0x000000	0x00	0 0
a	0x001000	0x0f	1 1

P2

seg	base	bounds	R W
8	0x000000	0x00	0 0
9	0x002000	0xff	1 1
a	0x003000	0x0f	1 1

...	
0x01f	0x001000
0x011	
0x003	
0x02a	
0x013	
...	
0x00c	0x002000
0x007	
0x004	
0x00b	
0x006	
...	
0x01f	0x003000

DISADVANTAGES OF SEGMENTED PAGE TABLES

Potentially large page tables (for each segment)

- Must allocate each page table contiguously
- Page tables can still be large for sparsely allocated address spaces

Growing a segment might not be easy

- We might need to relocate the entire page table
- Variable size page tables will cause some fragmentation

OTHER APPROACHES

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

MULTILEVEL PAGE TABLES

Goal: Allow each page table to be allocated non-contiguously

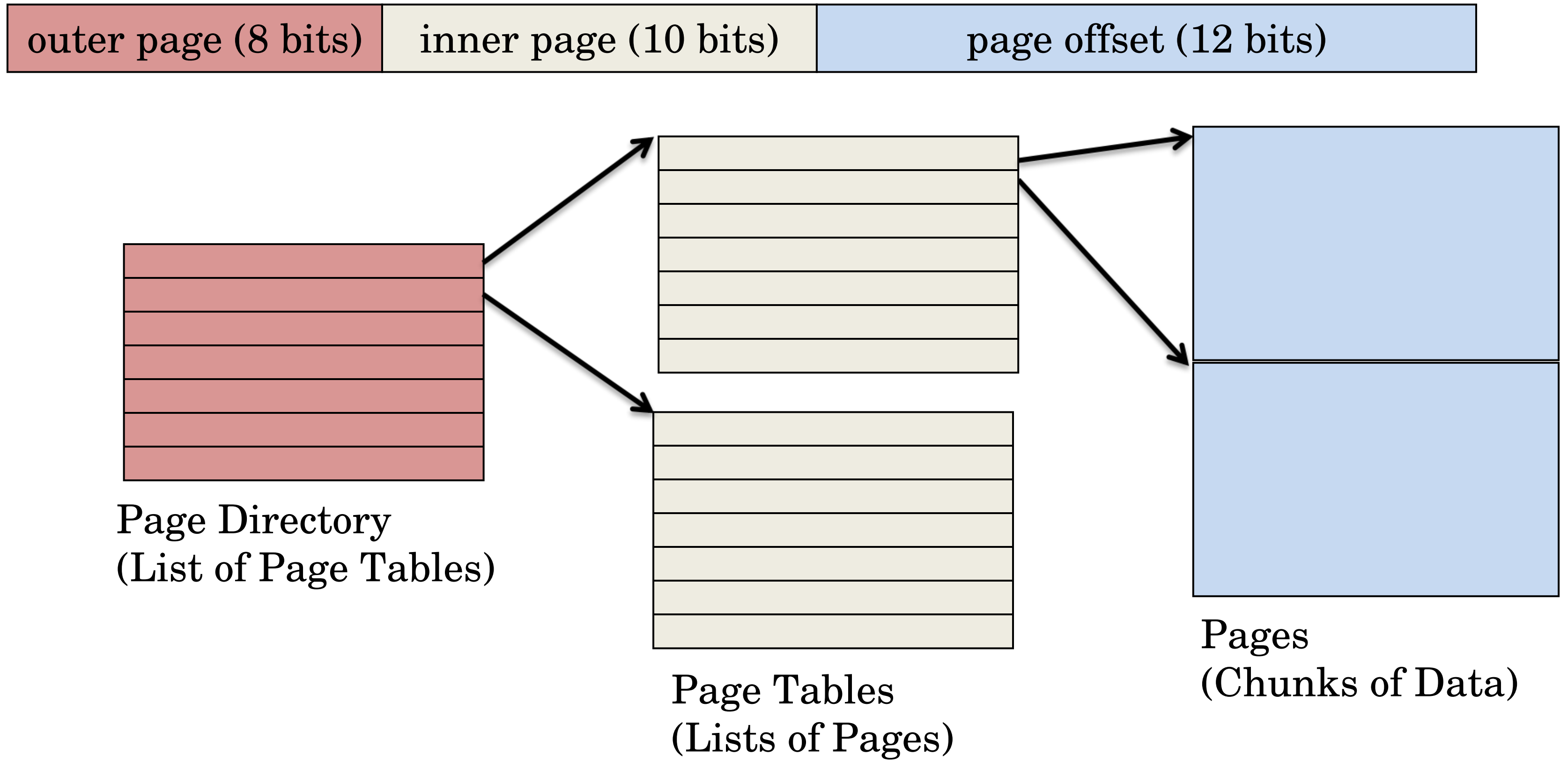
Idea: Page the page tables

- Creates multiple levels of page tables
  - Outer level is called page directory (per process)
- Only allocate portion of page table if at least one of its pages is in use

Used in x86 architectures (hardware can walk known structure)

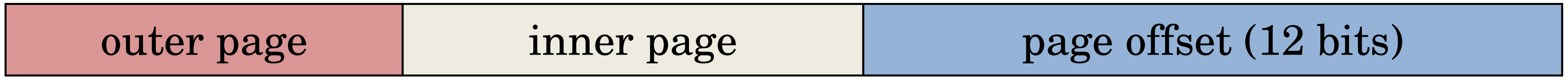
MULTI-LEVEL PAGE TABLES

30-bit virtual address:



ADDRESS FORMAT FOR MULTI-LEVEL PAGING

30-bit address



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

Goal? Each page table fits within a page (assume page size = 4KB, 2^12 bytes)

- number PTE * PTE size (assume 4bytes) = page size

How many page table entries can we fit on page?

- 4KB / 4bytes = 1K (1024 = 2^10) entries
- bits for selecting inner page = 10

Remaining bits for outer page: 30 bits - 12 bits - 10 bits = 8 bits

MULTI-LEVEL 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`

`0x23ABC`

translate `0x04000`

`0x80000`

translate `0xFEED0`

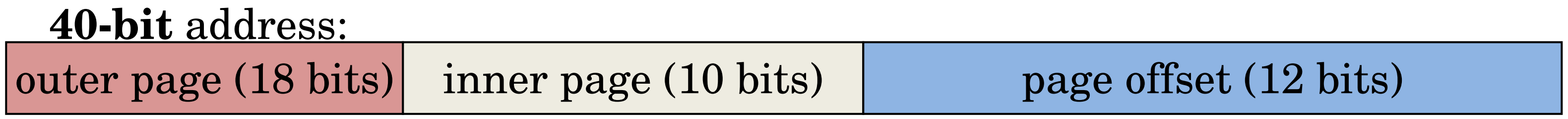
`0x55ED0`

20-bit address

outer page(4 bits)	inner page(4 bits)	page offset (12 bits)

PROBLEMS WITH TWO LEVELS?

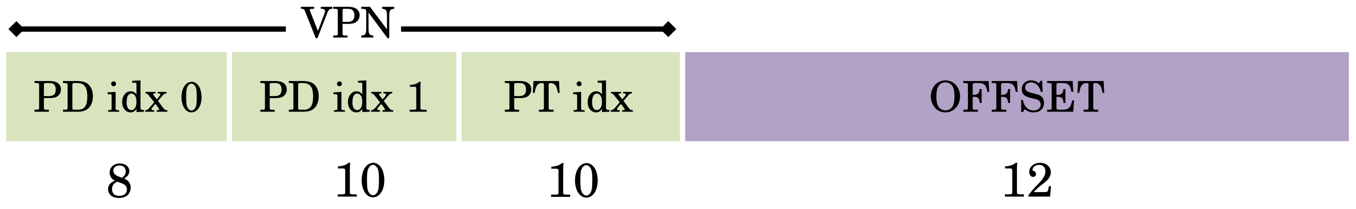
Problem: page directories (outer level) may not fit in a page



PageSize / PTE Size = 4KB / 4 bytes => max 1K entries per level

Solution:

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


VPN
PD idx 0
PD idx 1
PT idx
OFFSET
8
10
10
12