UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

VIRTUALIZING MEMORY: SMALLER PAGE TABLES

Questions answered in this lecture:

Review: What are problems with paging?

Review: How large can page tables be?

How can large page tables be avoided with different techniques?

Inverted page tables, segmentation + paging, multilevel page tables

What happens on a TLB miss?

ANNOUNCEMENTS

- P1: Will be graded by end of week (should be no surprises)
  - No discussion switches processed until then (304 is overloaded)
- Project 2: Available now
  - Due Friday, Oct 14th at 6pm; Watch discussion videos!
  - Shell and Scheduler
  - Work with partner for part b; fill out form for matches; notified tomorrow
- Exam 1: Next week: Wednesday 10/5 7:15 – 9:15pm
  - Class time on Tuesday for review, plus Wed discussion section
  - Look at previous exam / simulations for sample questions
- Reading for today: Chapter 20

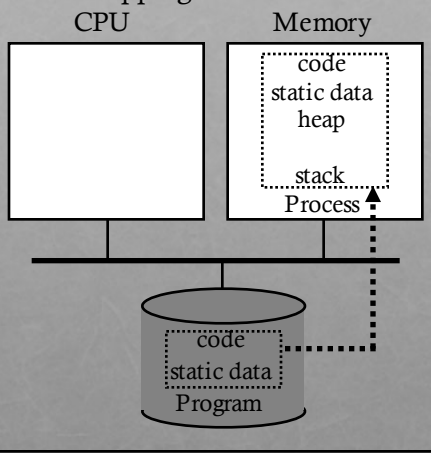
WHEN ARE PAGE TABLES CREATED?

OS creates new page table when creates process

- OS chooses where process code, heap, and stack are placed in RAM
- OS sets up page tables to contain initial mappings

OS modifies page tables when it allocates more process address space

- Calls to library malloc()
  - Library makes sbrk() system call
- Procedure calls: stack growth



DISADVANTAGES OF PAGING

1. Additional memory reference to look up in page table
   - Very inefficient
   - Page table must be stored in memory
   - MMU stores only base address of page table
   - **Avoid extra memory reference for lookup with TLBs (previous lecture)**
2. 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
   - Problematic with dynamic stack and heap within address space (today)

QUIZ: HOW BIG ARE PAGE TABLES?

How big is each page table?

1. PTE's are **2 bytes**, and **32** possible virtual page numbers
   32 * 2 bytes = 64 bytes
2. PTE's are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**
   2 bytes * 2^(24 – lg 16) = **2^21 bytes** (2 MB)
3. PTE's are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**
   4 bytes * 2^(32 – lg 4K) = **2^22 bytes** (4 MB)
4. PTE's are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**
   4 bytes * 2^(64 – lg 4K) = **2^54 bytes**

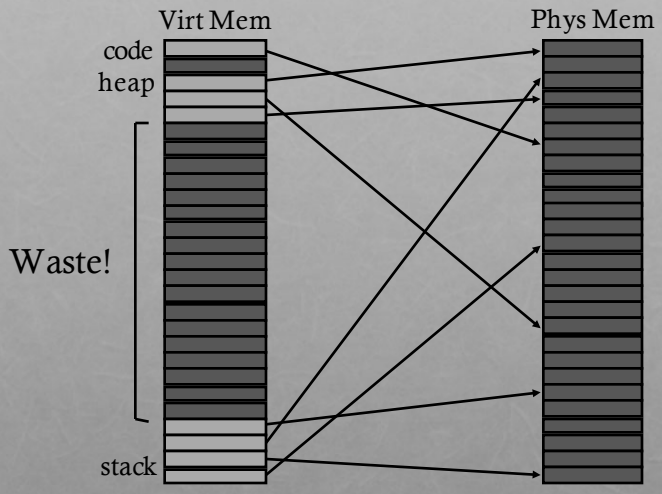
SIMULATION PRACTICE

See README-paging-linear-size and paging-linear-size.py

Example: paging-linear-size.py -c -p 4k -e 8 -v 42

- ARG bits in virtual address 42
- ARG page size 4k
- ARG pte size 8

WHY ARE PAGE TABLES SO LARGE?



MANY INVALID PT ENTRIES

Format of linear page tables:

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?

AVOID SIMPLE LINEAR PAGE TABLE

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

APPROACH 1: INVERTED PAGE TABLE

Inverted Page Tables

- Only need entries for virtual pages w/ valid physical mappings

Basic approach?
Search through data structure <ppn, vpn+asid> to find match

- Too much time to naively search entire table

Quick search: Find possible matches entries by hashing vpn+asid

- Smaller number of entries to search for exact match

TLB still manages most cases

- Managing inverted page table requires **software-controlled TLB**

OTHER APPROACHES

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

VALID PTES ARE CONTIGUOUS

how to avoid storing these page table entries?

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?

Segmentation

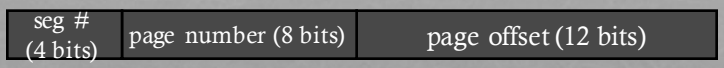
2) COMBINE PAGING AND SEGMENTATION

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
- Each segment tracks base (physical address) and bounds of **page table** for that segment

QUIZ: PAGING AND SEGMENTATION

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
0x010424 write:    error
0x210014 write:    error
0x203568 read:   0x02a568
```

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

ADVANTAGES OF PAGING AND SEGMENTATION

Advantages of Segments

- Supports sparse address spaces
  - Decreases size of page tables
  - If segment not used, no need for page table

Advantages of Pages

- No external fragmentation
- Segments can grow without any reshuffling
- 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
  - How?

SHARING: PAGING AND SEGMENTATION

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

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

P1: 0x802070 read:
P2: 0x902070 read:
P2: 0xa00100 read:

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

DISADVANTAGES OF PAGING AND SEGMENTATION

Potentially large page tables (for each segment)

- Must allocate each page table contiguously
- More problematic with more address bits
- 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

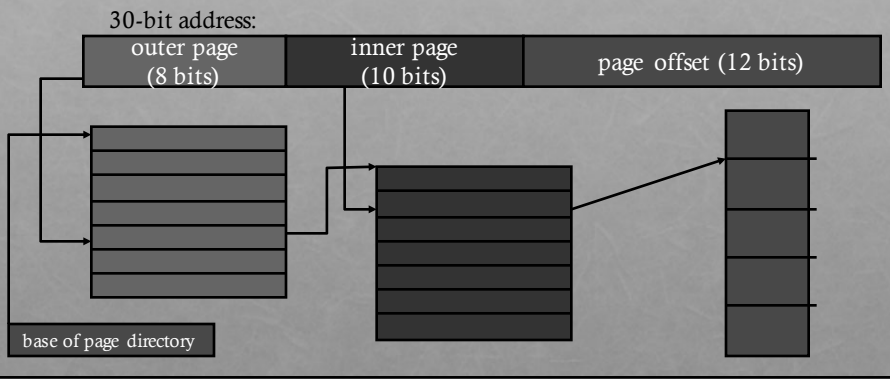
1. ~~Inverted Pagetables~~
2. ~~Segmented Pagetables~~
3. Multi-level Pagetables
   - Page the page tables
   - Page the pages of page tables…

3) MULTILEVEL PAGE TABLES

Goal: Allow each page tables 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)



QUIZ: MULTILEVEL

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
0x55	1
0x45	1

translate 0x01ABC

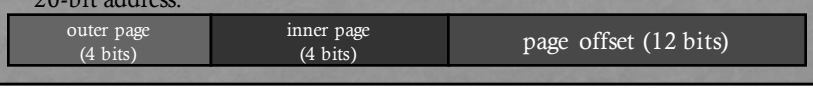
0x23ABC

translate 0x04000

0x80000

translate 0xFEED0

0x55ED0



QUIZ: ADDRESS FORMAT FOR MULTILEVEL 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
- PTE size * number PTE = page size
  - Assume PTE size = 4 bytes
  - Page size = 2^12 bytes = 4KB
  - 2^2 bytes * number PTE = 2^12 bytes
  - → number PTE = 2^10
- → # bits for selecting inner page = 10

Remaining bits for outer page:

- 30 – 10 – 12 = 8 bits

BREAK

When do you think someone with a CS degree will become President of the US?

How do you think they will be different than previous Presidents?

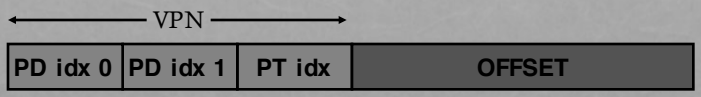
PROBLEM WITH 2 LEVELS?

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 given 1, 2, 3 levels?

4KB / 4 bytes → 1K entries per level

1 level: 1K * 4K = **2^22** = 4 MB

2 levels: 1K * 1K * 4K = **2^32** ≈ 4 GB

3 levels: 1K * 1K * 1K * 4K = **2^42** ≈ 4 TB

SIMULATIONS: MULTI-LEVEL PAGETABLES

Each page is 32 bytes

The virtual address space for the process in question (assume there is only one) is 1024 pages, or 32 KB

Physical memory consists of 128 pages

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN).

A physical address requires 12 bits (5 offset, 7 for the PFN).

The system assumes a multi-level page table. Thus, the upper five bits of a virtual address are used to index into a page directory; the page directory entry (PDE), if valid, points to a page of the page table. Each page table page holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual page in question.

The format of a PTE is thus:

VALID | PFN6 ... PFN0

and is thus 8 bits or 1 byte.

The format of a PDE is essentially identical:

VALID | PT6 ... PT0

`paging-multilevel-translate.py`

`README-paging-multilevel-translate`

SIMULATIONS: MULTI-LEVEL PAGETABLES

Virtual Address **611c:** What Physical Address (What Value)? Or Fault?

0x611c → 0110 0001 0001 1100

PDBR: 108 (decimal) [page directory is held in this page]

```
page 108: 83 fe e0 da 7f d4 7f eb be 9e d5 ad e4 ac 90 d6 92 d8 c1 f8 9f e1 ed e9 a1 e8 c7 c2 a9 d1 db ff
```

Which entry of PageDir?

PDE: 16+8=24

→ 0xa1

→ 1010 0001

→ Valid and 33

```
page  33: 7f 7f 7f 7f 7f 7f 7f 7f b5 7f 9d 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f f6 b1 7f 7f 7f 7f
```

Which entry of Page Table?

PTE: 8

→ b5

→ 1011 0101

→ valid and 32+16+4+1 = Page 53

```
page  53: 0f 0c 18 09 0e 12 1c 0f 08 17 13 07 1c 1e 19 1b 09 16 1b 15 0e 03 0d 12 1c 1d 0e 1a 08 18 11 00
```

Which offset on page?

→ offset 16+8+4 = 28

Final data value: 08!

QUIZ: FULL SYSTEM WITH TLBS

On TLB miss: lookups with more levels more expensive

How much does a miss cost?

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 previous ops changing TLB)

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

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

Total: 8

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

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

Total: 1

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

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

Total: 5

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 Topic:
What if desired address spaces do not fit in physical memory?

.