PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Memory Management III - Segmentation

Baskin Engineering UC SANTA CRUZ

Notices

- **Lab 3** due **Sunday March 1**
- **Assignment 4** due **Monday February 24**
  - 24 hour delay due to Presidents' Day
  - Make sure you have install from yesterday evening
    - `manpage` test was passing even when it wasn't ☺
    - `make submit` created incorrectly named archive ☹

Today's Lecture

- Intel 80386 Page Tables Revisited
- Address Translation
- TLB Hits and Misses
- Segmentation vs. Paging

- Introduction to Assignment 4

Intel 80386: Page Tables

Two problems with page tables:

- **Page table is too large**
  - Page table has 1M entries
  - Each entry is 4B (20-bit PPN)
  - **Page table is 4MB** (ouch!)
    - Very expensive in the 1980s
    - Still problematic today on embedded devices
- **Page table is stored in memory**
  - Before every memory access, we always have to fetch the PTE from comparatively slow RAM => big performance penalty

Intel 80386: Page Table Too Large

- How do we allow PTEs to become "unallocated"?
  - The page table must be restructured
- Before restructuring: **flat**

```
uint32_t PAGE_TABLE[1024*1024];
```

- After restructuring: **hierarchical**

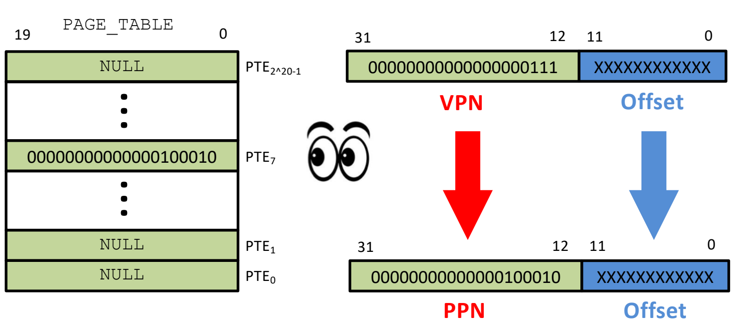
```
uint32_t *PAGE_DIRECTORY[1024];
PAGE_DIRECTORY[0] = NULL; // 1024 PTEs unallocated
PAGE_DIRECTORY[1] = NULL; // 1024 PTEs unallocated
PAGE_DIRECTORY[2] = malloc(sizeof(uint32_t)*1024);
PAGE_DIRECTORY[2][0] = 753;
PAGE_DIRECTORY[2][1023] = 21;
PAGE_DIRECTORY[3] = NULL; // 1024 PTEs unallocated
PAGE_DIRECTORY[4] = NULL; // 1024 PTEs unallocated
```


Intel 80386: Accelerating Translation

- **Problem:** Retrieving PTEs from memory is slow ☹
- **Solution:** "Cache" PTEs inside the processor ☺
  - **Translation Lookaside Buffer** (TLB)
    - "Lookaside Buffer" is an archaic term for a cache
  - Whenever a virtual address needs to be translated, the TLB is searched: "**hit**" vs. "**miss**" (we'll look at these next week)
  - 32-entry TLB on 80386
  - Each TLB entry has a **tag** and some **data**
    - Tag: 20-bit VPN + 4-bit metadata
    - Data: 20-bit PPN

Translation: "Flat" Page Table

```
pte_t PAGE_TABLE[1<<20];
PAGE_TABLE[7] = 34;
```

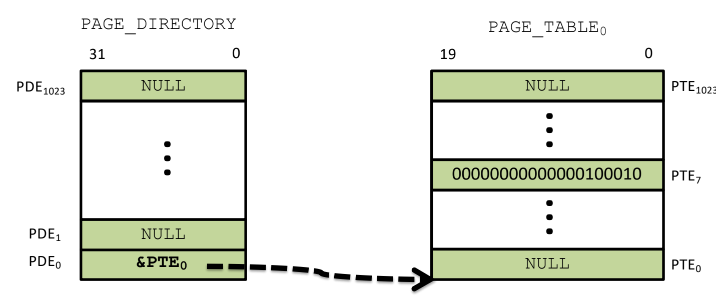


VPN is: 0000000000 0000000111

i.e. The Table Index

Translation: Two-Level Page Table

```
pte_t *PAGE_DIRECTORY[1<<10];
PAGE_DIRECTORY[0]= malloc((1<<10)*sizeof(pte_t));
PAGE_DIRECTORY[0][7]= 34;
```



VPN is: 0000000000 0000000111

Directory Index Table Index

Two-Level Page Table (x86)

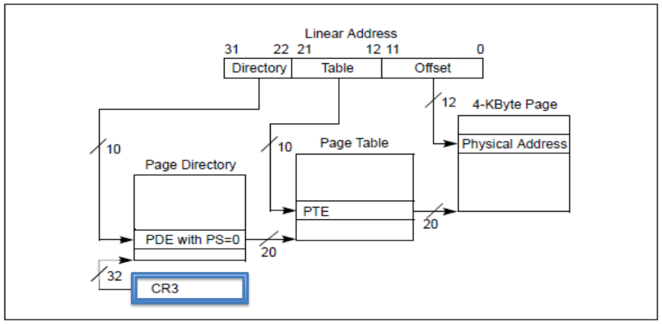
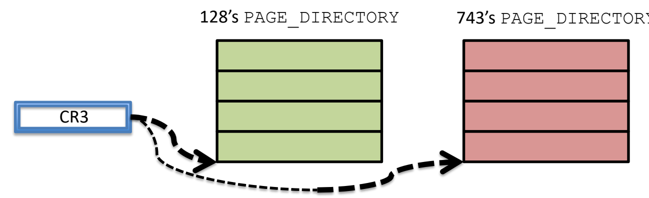


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

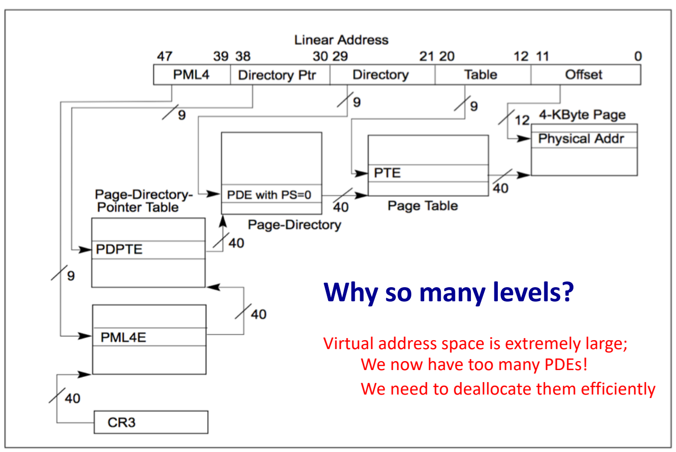
CR3: Control Register 3 (or **Page Directory Base Register**)
Stores the physical address of the page directory

Per-Process Virtual Address Space

- Each process has its own virtual address space
  - Process 128: text editor
  - Process 743: web browser
  - 128 writing to its virtual address 0 **does not** affect data stored in 743's virtual address 0 (or any other address in 128's virtual address space)
  - **This is the entire point of virtual memory!**
  - **Each process has it's own page directory and page tables**
    - On a context switch, CR3's value must be updated



Multi-Level Page Table (x86-64)



Why so many levels?

Virtual address space is extremely large;
We now have too many PDEs!
We need to deallocate them efficiently

Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

Context Switches and the TLB

- Process 128 is running
  - Process 128's VPN 5 is mapped to PPN 100
  - The TLB caches this mapping
    - VPN 5 -> PPN 100
- Now the operating system context switches to process 743
  - Process 743's VPN 5 is mapped to PPN 200
  - When process 743 tries to access VPN 5, it searches the TLB
    - Process 743 finds an entry with a tag of 5
    - Woo-hoo! It's a TLB hit!
    - **The PPN must be 200, right?**

WRONG! **(it's still 100 thanks to process 128 ☹)**

Context Switching Strategies

- **Flush the TLB**
  - On a context switch, invalidate all TLB entries
  - E.g. Intel 80386
    - Updating the value if CR3 signals a context switch
    - Automatically triggers a TLB flush
- **Or associate TLB entries with processes**
  - Add an extra field in the TLB tag
    - Identifies the process to which it belongs
  - On context switch, only invalidate entries belonging to suspended process
  - E.g. x86-64, MIPS

Handling TLB Misses

- The TLB is small; it cannot hold all PTEs ☹
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Known as **walking** the page directory/table (i.e. a “page walk”)
    - Significant performance penalty
- Who/what handles TPB misses?
  - Hardware?
  - Software?

Handling TLB Misses

- **Hardware Managed** (e.g. x86, ARM)
  - Hardware does the **page walk**
  - Hardware fetches the PTE and inserts it into the TLB
    - If TLB is full, this entry **replaces** (evicts) another entry
  - All done transparently
- **Software Managed** (e.g. MIPS, SPARC)
  - Hardware raises a “TLB Miss” exception
  - Operating system does the **page walk**
  - Operating system fetches the PTE
  - Operating system inserts/evicts entries in the TLB

Handling TLB Misses

- Hardware Managed TLB

  No exceptions. Instruction stalls (pauses)
  Independent instructions continue
  Small footprint
  Page directory/table organization burnt onto the chip
- Software Managed TLB

  The operating system can design the page directory/table
  Scope for sophisticated TLB replacement policies
  Flushes pipeline
  Performance overhead

Page Faults

- What if a virtual page is <u>not</u> mapped to a physical page?
  - i.e. the virtual page does not have a valid PTE
    - On x86, 0^{th} bit of PDE/PTE is set to 0
- **What would happen if you tried to access that page?**
  - Hardware exception: **page fault**
  - Operating system needs to handle it
    - Page fault handler

What Causes Page Faults?

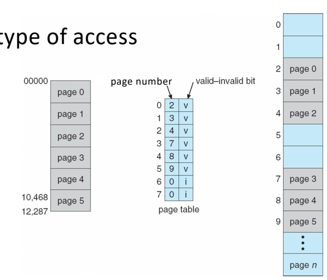
- Program error:

```
int *ptr = random();
int val = *ptr; // segmentation fault!
```

  - Operating system ***cannot*** save you from your own finger fumbles!
- Or the virtual page is mapped to **disk**, not memory:
  - This is the common meaning of “page fault”
  - Operating system <u>can</u> save you
    - Suspend process that caused the page fault
    - Read from disk into a physical page in memory
    - Map the virtual page to the physical page
    - Create appropriate PDE/PTE
    - Resume the process that caused the page fault

Memory Protection

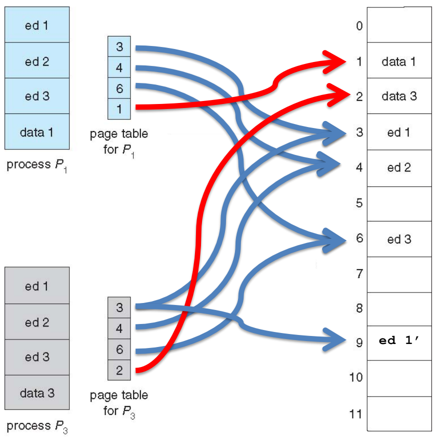
- **Valid-Invalid** bit attached to each entry in the page table
- Checked each time page table is consulted
- Any violations result in a trap to the kernel
- **Downsides of this scheme?**
- Idea led to mechanisms for controlling type of access
- Associate protection bit with each frame to indicate allowable access:
  - read-only or read-write access is allowed
  - also extend to execute-only



Shared Pages

- **Shared Code**
  - One copy of read-only (re-entrant) code shared between processes
  - Similar to multiple threads sharing the same process space
  - Also useful for inter-process communication
- Alternative: **Private Code and Data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space
- **Example**: An editor used by many users
  - Want to share the editor’s executable code, as it is the same for every user
  - Waste of memory to have identical concurrently resident pages
  - First process maps the editor’s virtual memory onto the physical
  - When later editor processes are loaded, the OS notes the pages are already present and re-maps them

Shared Page Example



Will this scheme always work?

No.

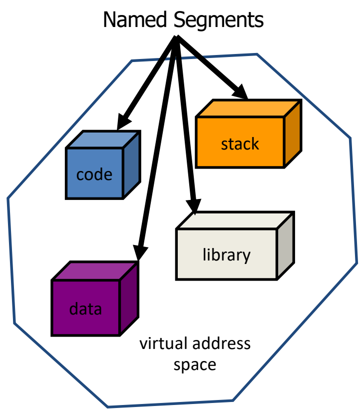
We need a COW (Copy-on-Write)

Segmentation

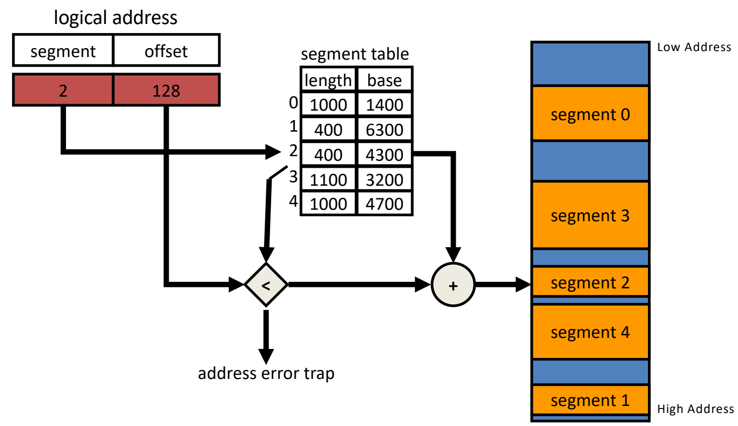
- Memory-management scheme supporting the user view of memory
- To a user a program is simply a collection of memory segments
- A **segment** is a named (or numbered) logical entity such as:
  - the main function
  - function
  - method
  - object
  - local variable
  - global variable
  - stack
  - symbol table
  - array
  - etc.

Segmentation

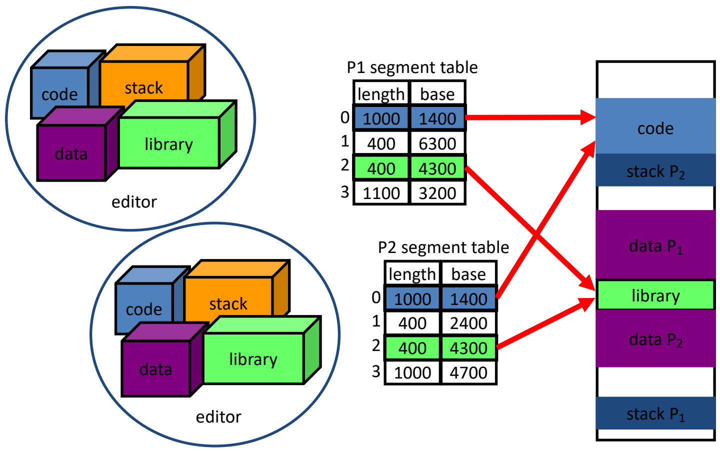
- Reflects logical view of processes
- Addresses: `<segment-number, offset>` map 2-D to 1-D
- Don't care about physical location
- **DO** care about the numbered or named segments
- Facilitates sharing by name or number



Segmentation: Mapping Segments



Segmentation: Sharing Segments



Paging vs. Segmentation

	Paging	Segmentation
Allocation Size	Pages are fixed size	Segments are variable size
Fragmentation	Internal within the pages	External between the segments
Addresses	User address is translated to a page number and offset	User specifies the segment number and offset
Size	Hardware defines page size	User can choose segment size
Mapping	Page tables hold base address of each physical page	Segment table contains segment number and length

Paging vs. Segmentation

	Paging	**Segmentation**
Does the program need to be aware the technique is being used?	No	No
How many linear address spaces are there	1	Many
Can the total address space exceed the physical memory?	Yes	Yes
Can executable code and data be distinguished and separately protected?	No (**hardware now does this**)	Yes
Can tables whose size varies be accommodated easily	No	Yes
Is sharing of functions between users facilitated?	No (**dynamic libraries do this**)	Yes
Principle idea behind the technique?	To get a large linear address space.	To abstract the structure of programs in memory, to permit sharing, and allow protection

Paging vs. Segmentation

- Linux Paging (with Copy-on-Write)
- BSD 4.4 Paging (Least Recently Used replacement)
- FreeBSD Paging (Least Actively Used replacement)
- Unix SVR4 Paging
- SCO Unix Paging
- macOS Paging
- iOS Paging
- IBM AIX Paging
- Android Paging
- Windows Paging
- OS400 Segmentation

The Nature of Computer Programs

- **Non-trivial programs**:
  - Have many rarely used features
  - Include code to handle rare edge cases (i.e. exceptional conditions)
  - Sometimes allocate more memory than they need

```
int buffer[1024];
for (int i = 0; i < smallNumberFromUserInput; i++)
    buffer[i] = i;
```

- **Virtual address spaces**:
  - Are **implicitly sparse**, with holes for growth, shared code, etc.
  - System libraries shared by mapping into the virtual address space
  - Shared memory by mapping pages read-write into virtual address space
  - Pages can be shared during `fork()`, speeding up process creation

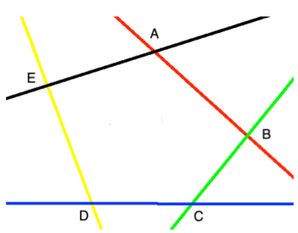
Assignment 4 - Introduction

- Basic / Manpage
  - Seven paragraphs from a manual page assigned at random to seven threads
  - Need to display them in the correct order
  - Synchronize activity of the threads so they show their paragraphs in order

```
$./manpage

A semaphore S is an unsigned-integer-valued variable.
Two operations are of primary interest:

P(S): If processes have been blocked waiting on this semaphore,
 wake one of them, else S <- S + 1.

V(S): If S > 0 then S <- S - 1, else suspend execution of the calling process.
 The calling process is said to be blocked on the semaphore S.

A semaphore S has the following properties:

1. P(S) and V(S) are atomic instructions. Specifically, no
 instructions can be interleaved between the test that S > 0 and the
 decrement of S or the suspension of the calling process.

2. A semaphore must be given an non-negative initial value.

3. The V(S) operation must wake one of the suspended processes. The
 definition does not specify which process will be awakened.
```


Assignment 4 - Introduction

- Advanced / Cartman
  - Tracks: Red, Green, Blue, Yellow, Black & Junctions: A, B, C, D, E
  - CARTs (Continuous Automated Rolling Trolleys) must cross critical sections of track only when they have *exclusive access* to the junctions at both ends
  - How many critical sections of track can *simultaneously* have a CART on them?




```
$ ./cartman -s
** Simple Test **
CART 0 on track 0 between 0 and 1
CART 1 on track 3 between 3 and 4
CART 2 on track 1 between 1 and 2
CART 3 on track 4 between 4 and 0
CART 4 on track 2 between 2 and 3
PASS All carts crossed safely
```

```
$ ./cartman -d
** Deadlock Test **
CART 0 on track 0 between 0 and 1
CART 3 on track 3 between 3 and 4
CART 1 on track 1 between 1 and 2
CART 4 on track 4 between 4 and 0
CART 2 on track 2 between 2 and 3
PASS All carts crossed safely
```


Next Lecture

- Demand Paging
- Frame Allocation
- Assignment 4 - Secret Sauce