Logistics

• Proj 3 Released, Due on August 12
  • Design Doc Due Wednesday

• HW3 Released, Due on August 13
  • Last Tuesday of the class
Recall: Paging

Process sees contiguous virtual address space
- Transparently divided into virtual *pages*

Physical Memory is Divided Into Chunks Called *Frames*
Implementing Paging

Page Table: *Process-Specific* map for bookkeeping
- Tells us in which physical page frame a virtual page is stored
- Virtual page number is index into table
- Table itself resides in physical memory
- Trigger **fault** if page access is not permitted
Recall: Space Consumption

• OS is paying bookkeeping cost for the entire virtual address range
  • Keep page entry for every virtual page
  • Most entries marked invalid

• But most processes only allocate and use a small fraction of their address space!

• Apply tree structure to virtual memory: *Multilevel Page Tables*
Two-Level Page Tables

- A tree of page tables
- Each "node" has a fixed size
  - x86: 1024 4-byte entries (why?)
- Still just one register to change on context switch
Combining Segments & Pages

- One approach: Each segment has its own page table
- Fragmentation/allocation advantages of pages
Real World: 32-bit x86

• Has both segmentation and paging
• Segmentation different from what we've described
  • Segment identified by instruction, not address

• Note: x86 actually offers multiple modes of memory operation, we'll talk about a common one
Real Example: Intel x86 (Old Days)

80386 Special Registers

<table>
<thead>
<tr>
<th>Segment Registers</th>
<th>Code Seg.</th>
<th>Data Seg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>CS</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>DS</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stack Seg.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SS</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extra Seg.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>ES</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extra Seg.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>GS</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extra. Seg.</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Unused</td>
</tr>
<tr>
<td>30</td>
<td>CR1</td>
</tr>
<tr>
<td>29</td>
<td>0 Flags</td>
</tr>
<tr>
<td>31 30</td>
<td></td>
</tr>
<tr>
<td>Page Directory</td>
<td>Not Used</td>
</tr>
<tr>
<td>Base Register</td>
<td></td>
</tr>
<tr>
<td>31 30</td>
<td></td>
</tr>
<tr>
<td>Page Fault</td>
<td></td>
</tr>
<tr>
<td>Lineart Address</td>
<td>CR2</td>
</tr>
<tr>
<td>31 30</td>
<td></td>
</tr>
</tbody>
</table>

Today: Register points to segment table in physical memory
Intel x86 Segmentation

- Six segments: cs (code), ds (data), ss (stack), es, fs, gs (extras)
- Instructions identify segment to use
  - `mov [es:bx], ax`
  - Some instructions have default segments, e.g. `push` and `pop` always refer to ss (stack)
- Underused in modern operating systems
- In 64-bit x86, only fs and gs support enforcement of base and bound
Intel x86 Segments + Paging

• Only one active page table

• Segment + Offset is called *logical address*

• Result of segmentation lookup is called a *linear address*

• Linear address is used for lookup in page table
x86 Memory Model With Segmentation (32-bit)
Real Page Table Entries

• So far, we've said they contain a physical frame #
  • Or frame # of a page table lower on the "tree"
• But we want some extra information
• Present/valid bit: indicate unallocated regions of physical address space
• Protection bits, e.g. to set certain regions of memory to read-only for sharing
32-bit x86 Page Table Entry

- 10/10/12 Split of Virtual Address
- Top-level page tables called *directories*

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Unused (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PC</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **PFN**: Physical Page number of page or next page table
- **P**: Present Bit (Is page mapping valid?)

*Page Table entry acts as a pointer*
32-bit x86 Page Table Entry

• 10/10/12 Split of Virtual Address
• Top-level page tables called *directories*

```
<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Unused (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PMT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

• **W**: Is this page *writable*?
• **U**: Can we access this page while in user mode?

*Page Table entry acts as record of permissions*
32-bit x86 Page Table Entry

- 10/10/12 Split of Virtual Address
- Top-level page tables called directories

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Unused (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **A**: Set by hardware when page is first accessed
- **D**: Page marked **dirty** if it is ever modified

*Page Table entry tells OS about usage patterns*
32-bit x86 Page Table Entry

- 10/10/12 Split of Virtual Address
- Top-level page tables called *directories*

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Unused (OS)</th>
<th>0</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>UW</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- **PWT**: Write-through cache behavior
- **PCD**: Disable caching for this page (more later)

*Pages for memory-mapped IO treated differently*
What Happens in the MMU?

Option 1: Hardware Traversal (e.g., x86)
• Hardware reads page tables itself for translation
• If we have a problem (e.g., page is marked invalid), invoke page fault handler
• Handler (software) decides how to react
What Happens in the MMU?

Option 2: Software Traversal (e.g., MIPS)

- Look through page tables in software
- Software may generate page fault to run handler
Software vs Hardware Traversal

- Hardware traversal is fast but inflexible
  - Hardware is already complex just to do basic lookup

- Software traversal slower but much more customizable (a "simple" matter of code)
  - But every translation prompts a fault so we can invoke handler to traverse tables

- In either case: lots of memory accesses, particularly for multi-level schemes
Recall: Dual-Mode Operation

- **Process cannot** modify its own page tables
  - Otherwise, it could access all physical memory
  - Even access or modify kernel

- Hardware distinguishes between user mode and kernel mode with special CPU register
  - **Protects Page Table Pointer**
    - Kernel has to ensure contents of page tables themselves are not pointed to by any mapping

- Remember: Page table can mark some frames as off limits for user-mode processes
Synchronous Exceptions

- System calls are an example of a synchronous exception (a "trap" into the kernel)
- Other exceptions: page fault, access fault, bus error
- Handled much like syscalls!
  - Argument passing a bit different
- Special hardware registers for args
  - Example: Virtual address that caused page fault
- Often rerun the triggering instruction
  - After fixing something
  - Relies on precise exceptions
Paging Tricks

• What does it mean if a page table entry doesn't have the valid (present) bit set?
  • Region of address space is invalid or
  • Page is not loaded and ready yet

• When program accesses an invalid PTE, OS gets an exception (a page fault or protection fault)

• Options
  • Terminate program (access was actually invalid)
  • Get page ready and restart instruction
Example Paging Tricks

1. Demand Paging: Swapping for pages
   - Keep only active pages in memory
   - Remember: not common on modern systems, except perhaps when first loading program
   - Response: Load in page from disk, retry operation

2. Copy on Write (remember fork?)
   - Temporarily mark pages as read-only
   - Allocate new pages when OS receives protection fault

3. Zero-Fill on Demand
   - Slow to overwrite new pages with all zeros
   - Page starts as invalid, zero it out when it's actually used
Making Exceptions Transparent

- Exception prompts OS to fix something, e.g. load of finally copy a page
- Can we return just like with a system call?
  - **Not Quite**
  - Need to repeat instruction that caused exception, not advance past it
Precise Exceptions

- **Definition:** Machine's state is as if the program executed up to the offending instruction

- Hardware has to complete previous instructions
  - Remember pipelining
  - May need to revert side effects if instruction was partially executed

- OS relies on hardware to enforce this property
Starting a Program: Steps

1. Allocate Process Control Block
2. Read (some of) program off disk and store in memory
3. Allocate Page Table
   • Set up entries for code so program can execute
4. Set up machine registers
   • Includes page table pointer
5. Set HW to user mode and jump to code
Caching

• **Cache**: Repository for copies that can be accessed more quickly than the originals

• Goal: Improve **performance**

• We'll see lots of applications!
  • Memory
  • Address translations
  • File contents, file name to number mappings
  • DNS records (name to IP addr mappings)
Why Bother with Caching?

Growing latency gap between CPU and Memory

"Moore's Law"

"Less' Law?"

Processor-Memory Performance Gap:
(grows 50% / year)

μProc 60%/yr.
(2X/1.5yr)

DRAM 9%/yr.
(2X/10 yrs)
Why cache? Address Translation

Three DRAM accesses per access?
• Unacceptably slow performance
• Solution: Cache translation mappings
• *Translation Lookaside Buffer (TLB)*
Average Access Time

- **Average Access Time** = 
  \[(Hit \ Rate \times \ Hit \ Time) + (Miss \ Rate \times \ Miss \ Time)\]

- Intuition: Caching is good when **most accesses** are for a **small portion** of the possible items.
- "Most Accesses": High Hit Rate
- "Small Portion": Hits can be faster than misses
  - Smaller storage is faster
Why Does Caching Help? Locality!

- **Temporal Locality** *(Locality in Time)*:
  - Keep recently accessed data items closer to processor

- **Spatial Locality** *(Locality in Space)*:
  - Move contiguous blocks to the upper levels
Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology
Types of Cache Miss

1. **Compulsory** ("cold start"): First access to a block
   • Insignificant in any long-lived process

2. **Capacity**: Not enough space in cache

3. **Conflict**: Memory locations map to same cache location

4. **Coherence** (invalidation): Memory updated externally
   • e.g. multi-core system, or on I/O
Types of Cache Miss

1. **Compulsory** ("cold start"): First access to a block
   - Insignificant in any long-lived process

2. **Capacity**: Not enough space in cache

3. **Conflict**: Memory locations map to same cache location
   - Not affected by cache design (mostly)

4. **Coherence** (invalidation): Memory updated externally
   - e.g. multi-core system, or on I/O
Types of Cache Miss

1. **Compulsory** ("cold start"): First access to a block
   - Insignificant in any long-lived process

2. **Capacity**: Not enough space in cache

3. **Conflict**: Memory locations map to same cache location

4. **Coherence** (invalidation): Memory updated externally
   - e.g. multi-core system, or on I/O

**Improved by Increasing Cache Size**
Types of Cache Miss

1. Compulsory ("cold start"): First access to a block
   • Insignificant in any long-lived process

2. Capacity: Not enough space in cache

3. **Conflict**: Memory locations map to same cache location

4. Coherence (invalidation): Memory updated externally
   • e.g. multi-core system, or on I/O

**Improved by Increasing Associativity**
How is a Block found in a Cache?

- **Block** is the minimum unit of caching (recall spatial locality)
  - Data Select: Which part of block to retrieve
- **Index** is used to Lookup Candidates in Cache
  - Index identifies the set
- **Tag** is used to identify the actual copy
  - If no candidates match, then declare cache miss

```
<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Block Address</td>
</tr>
</tbody>
</table>
```

Set Select

Data Select
Direct Mapped Cache

- Example: $2^{10}$ byte capacity cache, $2^5$ byte blocks
- 32-bit memory addresses
- Need 5 bits for block offset
- Need another 5 bits for index ($2^{10}/2^5 = 2^5$ blocks)
- Remaining 22 bits are tag bits

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ex: 0x50

<table>
<thead>
<tr>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 31</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Byte 63</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Byte 1023</td>
</tr>
<tr>
<td>31</td>
</tr>
</tbody>
</table>
Direct Mapped Cache

• Check index bits to find one potential block
• Compare tag to verify that block contains our data
• Use byte select (offset) to choose byte within block

Ex: 0x50
Ex: 0x01
Ex: 0x00

Valid Bit
Cache Tag
Cache Index
Byte Select

Cache Data

Byte 31
●●
Byte 1
Byte 0

0

Byte 63
●●
Byte 33
Byte 32

2

Byte 1023
●●
Byte 992

31
Set Associative Cache

- **N-way set associative**: $N$ entries per Cache Index
  - $N$ direct mapped caches operating in parallel, or
  - Index gives you a set of possible locations

![Diagram of a set associative cache](image)
Fully Associative Cache

- **Fully Associative:** Any block could hold data for a given memory address
- Compare cache tag bits in all blocks
- No index bits
Where Can a Block Go?

- Example: Block 12 placed in 8 block cache

### Direct mapped:
- Block 12 can go only into block 4 (12 mod 8)

### Set associative:
- Block 12 can go anywhere in set 0 (12 mod 4)

### Fully associative:
- Block 12 can go anywhere
Which block should be replaced on a miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

- Miss rates for a workload:

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way</th>
<th></th>
<th>4-way</th>
<th></th>
<th>8-way</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU Random</td>
<td>LRU Random</td>
<td>LRU Random</td>
<td>LRU Random</td>
<td>LRU Random</td>
<td>LRU Random</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>4.7%</td>
<td>5.3%</td>
<td>4.4%</td>
<td>5.0%</td>
<td></td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>3.5%</td>
<td>1.5%</td>
<td>1.7%</td>
<td>1.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.12%</td>
<td>1.12%</td>
</tr>
</tbody>
</table>
What Happens When We Write to a Cached Address?

• **Write Through**: Update *both* cache block and corresponding location in main memory
  • Simple to implement, but now we wait for writes?

• **Write Back**: Update only the cached copy at time of write. Update main memory when cache block is removed later on.
  • Repeated writes not all sent to RAM
  • A read may trigger a write (evicting a cache block)
  • More complex (need *modified* "dirty" bit)
Summary

• Memory Hierarchy and Locality
  • Temporal Locality: Likely to reference same data soon
  • Spatial Locality: Likely to reference nearby data

• Causes of Cache Misses
  • Compulsory: First Access
  • Conflict: Cache too small, or limited associativity
  • Capacity: Cache is too small
  • Coherence: Something else changed memory location

• Cache Organizations
  • Direct Mapped: Single block could hold address
  • Set associative: Multiple candidate blocks
  • Fully Associative: Data could be anywhere
Caching Applied to Address Translation

- CPU
- Virtual Address
- Physical Memory
- Physical Address
- Data Read or Write (untranslated)
- TLB
- Cached?
  - Yes
  - No
- Translate (MMU)
- Result
  - Save
Caching Address Translations

• Locality in page accesses?

• **Yes**: Spatial locality, just like CPU cache

• **TLB**: Cache of page table entries
TLB and Context Switches

• Do nothing upon context switch?
  • New process could use old process's address space!

• Option 1: Invalidate ("flush") entire TLB
  • Simple, but very poor performance

• Option 2: TLB entries store a process ID
  • Called tagged TLB
  • Requires additional hardware
TLB and Page Table Changes

• Think about what happens we we use **fork**
• OS marks all pages in address space as read-only
• After parent returns from **fork**, it tries to write to its stack
• Triggers a protection fault. OS makes a copy of the stack page, updates page table.
• Restarts instruction.
TLB and Page Table Changes

What if TLB had cached the old read/write page table entry for the stack page?

• OS marks all pages in address space as read-only
• After parent returns from fork, it tries to write to its stack
• Triggers a protection fault. OS makes a copy of the stack page, updates page table.
• Restarts instruction.
How do we invalidate TLB entries?

- Hardware could keep track of where page table for each entry is and monitor that memory for updates…
- Very complicated!
- Especially for multi-level page tables and tagged TLBs

- Instead: The OS must invalidate TLB entries
- So TLB is not entirely transparent to OS
TLB and Page Table Changes

• Think about what happens when we use `fork`.
• OS marks all pages in address space as read-only and tells MMU to clear those TLB entries.
• After parent returns from `fork`, it tries to write to its stack.
• Triggers a protection fault. OS makes a copy of the stack page, updates page table. Also tells MMU to clear that TLB entry.
• Restarts instruction.