Reminder: Midterm II and Review Session

- This Wednesday in class
  - April 11
  - 1 hr 50 minutes, sharp
  - Everything covered in the course can be on the exam
  - You can bring in two cheat sheets (8.5x11”)
  - Arrive early (12:25pm)

- Midterm II review session
  - Today 4:30-6:30pm
  - Room: PH 125C
Reminder: Homeworks

- Homework 6
  - Due April 16
  - Topics: Main memory, caches, virtual memory
Reminder: Lab Assignments

- Lab Assignment 6
  - Implementing a more realistic memory hierarchy
    - L2 cache model
    - DRAM, memory controller models
    - MSHRs, multiple outstanding misses
  - Due April 23
  - Extra credit: Prefetching
Readings

- Virtual memory
  - Section 5.4 in P&H
  - Optional: Section 8.8 in Hamacher et al.
Review of Last Lecture (I)

- Bloom filters
  - Compact and approximate way of representing set membership
  - Allows easy way of testing set membership

- Memory request scheduling in DRAM
  - Single-core scheduling policies
  - Inter-thread interference and its causes and effects
  - Stall-time fair memory scheduling
  - Parallelism-aware batch scheduling
  - Other approaches
    - Channel partitioning
    - Core/request throttling
Review of Last Lecture (II)

- Virtual memory start
  - Motivation
  - Address translation
  - Page table (per process)
  - Page fault handling
Virtual Memory
Roadmap

- Virtual Memory
  - Purpose: illusion of a large memory and protection
  - Simplified memory management for multiple processes
  - Demand paging, page faults
  - Address Translation
  - TLB
  - Integrating Caches and Virtual Memory
    - Physically indexed caches
    - Virtually indexed caches
    - Virtually indexed, physically tagged caches
    - Synonym/aliasing problem and solutions
Review: Virtual Memory

- Idea: Give the programmer the illusion of a large address space
  - So that he/she does not worry about running out of memory

- Programmer can assume he/she has “infinite” amount of physical memory
  - Really, it is the amount specified by the address space for a program

- Hardware and software cooperatively provide the illusion even though physical memory is not infinite
  - Illusion is maintained for each independent process
Review: Virtual Pages, Physical Frames

- **Virtual** address space divided into pages
- **Physical** address space divided into frames

A virtual page is mapped to a physical frame
- Assuming the page is in memory

If an accessed virtual page is not in memory, but on disk
- Virtual memory system brings the page into a physical frame and adjusts the mapping → demand paging

**Page table** is the table that stores the mapping of virtual pages to physical frames
Review: Address Translation

Example: 8K page size
32-bit virtual address space

VPN → 19 bits
→ $2^{19}$ virtual pages
→ $2^{19}$ PTEs in page table
(for each process)
Review: Page Table Entry

![Diagram of Page Table Entry]

- **Valid bit**: (Is the page present in physical memory?)
- **Dirty bit**:
- **Protection or access control bit(s)** (Can the process access the page? What kind of access?)
- **Reference or access bit**: (Was the page referenced recently?)
- **Physical frame the page is stored in (if valid)**: PTE
VM Address Translation

- Parameters
  - $P = 2^p$ = page size (bytes).
  - $N = 2^n$ = Virtual-address limit
  - $M = 2^m$ = Physical-address limit

Page offset bits don’t change as a result of translation
VM Address Translation

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page

Page table base register

VPN acts as table index

if valid=0 then page not in memory (page fault)

virtual address

$n-1$  $p$  $p-1$

virtual page number (VPN)  page offset

valid access  physical page number (PPN)

$m-1$  $p$  $p-1$

physical frame number (PFN)  page offset

physical address
VM Address Translation: Page Hit

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to L1 cache
5) L1 cache sends data word to processor
VM Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim, and if dirty pages it out to disk
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction.
Page-Level Access Control (Protection)

- Not every process is allowed to access every page
  - E.g., may need supervisor level privilege to access system pages

- Idea: Store access control information on a page basis in the process’s page table

- Enforce access control at the same time as translation

→ Virtual memory system serves two functions today
  
  Address translation (for illusion of large physical memory)
  Access control (protection)
Two Functions of Virtual Memory

1. Translation
2. Access control (protection)

PTE contains access control bits associated with the virtual page.
VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Page fault handler checks these before remapping
  - If violated, generate exception (Access Protection exception)

![Page Tables Diagram]

- Process i:
  - VP 0: Read? Yes, Write? No, Physical Addr PP 9
  - VP 1: Read? Yes, Write? Yes, Physical Addr PP 4
  - VP 2: Read? No, Write? No, Physical Addr XXXXXXX
- Process j:
  - VP 0: Read? Yes, Write? Yes, Physical Addr PP 6
  - VP 1: Read? Yes, Write? No, Physical Addr PP 9
  - VP 2: Read? No, Write? No, Physical Addr XXXXXXX
Access Control Logic

type of access

privilege level of running process

access allowed?

protection bits of the page to be accessed (in PTE)

type of access: R, W, E, more ordered: Acc, E, R, W

privilege level: specified by ISA

VAX: Kernel (K), Executive (E), Supervisor (S), User (U)

protection bits: specify what type of access can be made to this page & at what privilege level
Privilege Levels in x86

Figure 5-3. Protection Rings
Page Level Protection in x86

Table 5-3. Combined Page-Directory and Page-Table Protection

<table>
<thead>
<tr>
<th>Privilege</th>
<th>Access Type</th>
<th>Privilege</th>
<th>Access Type</th>
<th>Privilege</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>User</td>
<td>Read-Only</td>
<td>User</td>
<td>Read-Only</td>
</tr>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>User</td>
<td>Read-Write</td>
<td>User</td>
<td>Read-Only</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>User</td>
<td>Read-Only</td>
<td>User</td>
<td>Read-Only</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>User</td>
<td>Read-Write</td>
<td>User</td>
<td>Read/Write</td>
</tr>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>User</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>User</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>User</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>User</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Write</td>
<td>User</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Write</td>
<td>User</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Only</td>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read/Write</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read-Write</td>
<td>Supervisor</td>
<td>Read/Write*</td>
</tr>
</tbody>
</table>

*Denotes additional protection.
Aside: Protection w/o Virtual Memory

- Question: Do we need virtual memory for protection

- Answer: No

- Other ways of providing memory protection
  - Base and bound registers
  - Segmentation

- None of these are as elegant as page-based access control
  - They run into complexities as we need more protection capabilities
  - Virtual memory integrates
Base and Bound Registers

In a multi-tasking system

Each process is given a non-overlapping, contiguous physical memory region, *everything belonging to a process must fit in that region*

When a process is swapped in, OS sets **base** to the start of the process’s memory region and **bound** to the end of the region

HW translation and protection check (*on each memory reference*)

\[
PA = EA + \text{base}, \quad \text{provided } (PA < \text{bound}), \text{ else violations}
\]

⇒ *Each process sees a private and uniform address space (0 .. max)*

privileged control registers

Bound can also be formulated as a range
Base and Bound Registers

- Limitations of the base and bound scheme
  - large contiguous space is hard to come by after the system runs for a while---free space may be fragmented
  - how do two processes shared some memory regions but not others?
Segmented Address Space

- segment == a base and bound pair
- segmented addressing gives each process multiple segments
  - initially, separate code and data segments
    - 2 sets of base-and-bound reg’s for inst and data fetch
    - allowed sharing code segments
  - became more and more elaborate: code, data, stack, etc.

SEG # | EA

segment tables must be 1. privileged data structures and 2. private/unique to each process
Segmented Address Translation

- EA: segment number (SN) and a segment offset (SO)
  - SN may be specified explicitly or implied (code vs. data)
  - segment size limited by the range of SO
  - segments can have different sizes, not all SOs are meaningful

Segment translation and protection table
- maps SN to corresponding base and bound
- separate mapping for each process
- must be a privileged structure
How to extend an old ISA to support larger addresses for new applications while remaining compatible with old applications?
Issues with Segmentation

- Segmented addressing creates fragmentation problems:
  - a system may have plenty of unallocated memory locations
  - they are useless if they do not form a contiguous region of a sufficient size

- Page-based virtual memory solves these issues
  - By ensuring the address space is divided into fixed size "pages"
  - And virtual address space of each process is contiguous
  - The key is the use of indirection to give each process the illusion of a contiguous address space
In a Paged Memory System:
- PA space is divided into fixed size “segments” (e.g., 4kbyte), more commonly known as “page frames”
- VA is interpreted as page number and page offset

**Page-based Address Space**

<table>
<thead>
<tr>
<th>Page No.</th>
<th>Page Offset</th>
</tr>
</thead>
</table>

page tables must be 1. privileged data structures and 2. private/unique to each process
Some Issues in Virtual Memory
Virtual Memory Issues (I)

- How large is the page table?

- Where do we store it?
  - In hardware?
  - In physical memory? (Where is the PTBR?)
  - In virtual memory? (Where is the PTBR?)

- How can we store it efficiently without requiring physical memory that can store all page tables?
  - **Idea: multi-level page tables**
  - Only the first-level page table has to be in physical memory
  - Remaining levels are in virtual memory (but get cached in physical memory when accessed)
Suppose 64-bit VA and 40-bit PA, how large is the page table? $2^{52}$ entries x $\sim 4$ bytes $\approx 16 \times 10^{15}$ Bytes

and that is for just one process!!?
Multi-Level Page Tables in x86

This page mapping example is for 4-KByte pages and the normal 32-bit physical address size.

*Physical Address
Page Table Access

- How do we access the Page Table?

- Page Table Base Register (CR3 in x86)
- Page Table Limit Register

- If VPN is out of the bounds (exceeds PTLR) then the process did not allocate the virtual page → access control exception

- Page Table Base Register is part of a process’s context
  - Just like PC, PSR, GPRs
  - Needs to be loaded when the process is context-switched in
Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging
More on x86 Page Tables (II): Large Pages

Figure 4-3. Linear-Address Translation to a 4-MByte Page using 32-Bit Paging
Figure 4-4 gives a summary of the formats of CR3 and the paging-structure entries with 32-bit paging. For the paging structure entries, it identifies separately the format of entries that map pages, those that reference other paging structures, and those that do neither because they are “not present”; bit 0 (P) and bit 7 (PS) are highlighted because they determine how such an entry is used.

<table>
<thead>
<tr>
<th>Address of page directory¹</th>
<th>Ignored</th>
<th>P</th>
<th>C</th>
<th>D</th>
<th>Ignored</th>
<th>W</th>
<th>T</th>
<th>Ignored</th>
<th>CR3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 31:22 of address of 2MB page frame</td>
<td>Reserved (must be 0)</td>
<td>Bits 39:32 of address²</td>
<td>P</td>
<td>A</td>
<td>T</td>
<td>Ignored</td>
<td>G</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>Address of page table</td>
<td>Ignored</td>
<td>Q</td>
<td>I</td>
<td>g</td>
<td>n</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>G</td>
<td>P</td>
<td>A</td>
<td>T</td>
<td>D</td>
<td>A</td>
<td>P</td>
<td>C</td>
</tr>
<tr>
<td>Ignored</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging
### X86 PTE (4KB page)

#### Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 4-KByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/Supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>7 (PAT)</td>
<td>If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)¹</td>
</tr>
<tr>
<td>8 (G)</td>
<td>Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise</td>
</tr>
<tr>
<td>11:9</td>
<td>Ignored</td>
</tr>
<tr>
<td>31:12</td>
<td>Physical address of the 4-KByte page referenced by this entry</td>
</tr>
</tbody>
</table>
### X86 Page Directory Entry (PDE)

#### Table 4-5. Format of a 32-Bit Page-Directory Entry that References a Page Table

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to reference a page table</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (depends on CPL and CR0.WP; see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)</td>
</tr>
</tbody>
</table>
Four-level Paging in x86

Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging
A logical processor uses IA-32e paging if CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LME = 1. With IA-32e paging, linear address are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses.\(^1\) Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time.

IA-32e paging uses a hierarchy of paging structures to produce a translation for a linear address. CR3 is used to locate the first paging-structure, the PML4 table. Use of CR3 with IA-32e paging depends on whether process-context identifiers (PCIDs) have been enabled by setting CR4.PCIDE:
Virtual Memory Issues (II)

- How fast is the address translation?
  - How can we make it fast?

- Idea: Use a hardware structure that caches PTEs → Translation lookaside buffer

- What should be done on a TLB miss?
  - What TLB entry to replace?
  - Who handles the TLB miss? HW vs. SW?

- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?
Speeding up Translation with a TLB

- Essentially a cache of recent address translations
  - Avoids going to the page table on every reference

- **Index** = lower bits of VPN (virtual page #)
- **Tag** = unused bits of VPN + process ID
- **Data** = a page-table entry
- **Status** = valid, dirty

The usual cache design choices (placement, replacement policy, multi-level, etc.) apply here too.
Virtual to Physical Address Translation

Effective Address

TLB Lookup

hit

Page Table Walk

$\leq 1\ pclk$

miss

Protection Check

$\leq 1\ pclk$

succeed

Update TLB

fail

Permission Fault

Physical Address To Cache

100’s pclk by HW or SW

10000’s pclk
HW-managed vs SW-managed TLB miss

- **HW-managed TLB**
  - No exception raised. Instruction simply stalls.
  - Independent instructions to continue
  - State-machine replaces at most a few lines in D-cache
    - Page table organization etched in stone
    - OS has little flexibility in tailoring the design

- **SW-managed TLB**
  - OS can define page table organization
  - More advanced replacement policy
    - Higher performance cost (TLB Miss handler 100s instructions)
      (even worse if code not in I-cache)
    - Interrupt causes pipeline to be flushed
Issues (III)

- When do we do the address translation?
  - Before or after accessing the L1 cache?

- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache

- What are the issues with a virtually addressed cache?

- Synonym problem:
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data
Homonyms and Synonyms

- **Homonym**: Same VA can map to two different PAs
  - Why?
    - VA is in different processes

- **Synonym**: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, ...

- Do homonyms and synonyms create problems when we have a cache?
  - Is the cache virtually or physically addressed?
Cache-VM Interaction

- CPU
- TLB
- cache
- lower hier.

- CPU
- cache
- lower hier.

- CPU
- cache
- tlb
- lower hier.
Physical Cache
Virtual Cache
Virtual-Physical Cache

Where can the same physical address be in the code?
An Exercise

- Problem 5 from
  - ECE 741 midterm exam, Spring 2009
Virtually-Indexed Physically-Tagged

- If \( C \leq (\text{page\_size} \times \text{associativity}) \), the cache index bits come only from page offset (same in VA and PA)

- If both cache and TLB are on chip
  - index both arrays concurrently using VA bits
  - check cache tag (physical) against TLB output at the end

![Diagram of Virtually-Indexed Physically-Tagged cache and TLB interaction]

TLB hit?  

cache hit?
Virtually-Indexed Physically-Tagged

- If $C > (\text{page\_size} \times \text{associativity})$, the cache index bits include VPN
  $\Rightarrow$ Synonyms can cause problems
  - The same physical address can exist in two locations
- Solutions?

- TLB hit?
- Cache hit?

\[ \text{VPN} \rightarrow \text{PO} \]
\[ \text{IDX} \rightarrow \text{BO} \]
\[ \text{physical} \rightarrow \text{cache} \]
\[ \text{tag} \rightarrow \text{data} \]
Some Solutions to the Synonym Problem

- Limit cache size to page size times associativity
  - get index from page offset

- On a write, search all possible indices that can contain the same physical block, and update/invalidate
  - Used in Alpha 21264, MIPS R10K

- Restrict page placement in OS
  - make sure index(VA) = index(PA)
  - Called page coloring
  - Used in many SPARC processors
## Virtual Memory – DRAM Interaction

- **Operating System** influences where an address maps to in DRAM.

<table>
<thead>
<tr>
<th>Virtual Page number (52 bits)</th>
<th>Physical Frame number (19 bits)</th>
<th>Page offset (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>PA</td>
<td>VA</td>
</tr>
<tr>
<td>PA</td>
<td>PA</td>
<td>PA</td>
</tr>
</tbody>
</table>

- Operating system can control which bank/channel/rank a virtual page is mapped to.

- It can perform page coloring to minimize bank conflicts.
- Or to minimize inter-application interference.