CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Virtual Memory

Instructor:
Michael Greenbaum
New-School Machine Structures

- **Software**
  - Parallel Requests
    - Assigned to computer
    - e.g., Search “Katz”
  - Parallel Threads
    - Assigned to core
    - e.g., Lookup, Ads
  - Parallel Instructions
    - >1 instruction @ one time
    - e.g., 5 pipelined instructions
  - Parallel Data
    - >1 data item @ one time
    - e.g., Add of 4 pairs of words
  - Hardware descriptions
    - All gates @ one time

- **Hardware**
  - Software
  - Hardware
  - Harness Parallelism & Achieve High Performance
  - Virtual Memory
  - Computer
    - Core
    - Memory (Cache)
    - Input/Output
    - Instruction Unit(s)
    - A_0+B_0, A_1+B_1, A_2+B_2, A_3+B_3
    - Functional Unit(s)
    - Logic Gates
  - Smart Phone

8/1/2011 Summer 2011 -- Lecture #24
Overarching Theme for Today

“Any problem in computer science can be solved by an extra level of indirection.”

– Often attributed to Butler Lampson (Berkeley PhD and Professor, Turing Award Winner), who in turn, attributed it to David Wheeler, a British computer scientist, who also said “… except for the problem of too many layers of indirection!”

Butler Lampson
Agenda

• Virtual Memory Intro
• Page Tables
• Administrivia
• Translation Lookaside Buffer
• Break
• Demand Paging
• Putting it all Together
• Summary
Review: Memory Management

- The processor’s view of memory (eg, using the C programming language)
  - **Static storage**: global variable storage, basically permanent, entire program run
  - **The Stack**: local variable storage, parameters, return address
  - **The Heap** (dynamic storage): malloc() grabs space from here, free() returns it
The Problem

• Simplest Model: Only one program running on the computer
  – Addresses in the program are exactly the physical memory addresses

• Problems with the simple model:
  – What if less physical memory than full address space?
    • 32 bit addresses => 4 GB address space, RAM hasn’t always been larger than 4 GB.
    • 64 bit addresses => 16 exbibytes.
  – What if we want to run multiple programs at the same time?
The Problem

• Limited physical memory, one or more programs each with their own address space.

Application 1

- Code
- Static data
- Heap
- Stack

Application 2

- Code
- Static data
- Heap
- Stack

Physical Memory
First Idea: Base + Bounds Registers for Location Independence

Location-independent programs
Programming and storage management ease: need for a \textit{base register}

Protection
Independent programs should not affect each other inadvertently: need for a \textit{bound register}

Historically, base + bounds registers were a very early idea in computer architecture
Base and bounds registers are visible/accessible to programmer. Trap to OS if bounds violation detected ("seg fault"/"core dumped")
What prevents programs from accessing each other’s data?
Restriction on Base + Bounds Regs

Want only the Operating System to be able to change Base and Bound Registers

Processors need different execution *modes*

1. **User mode**: can use Base and Bound Registers, but cannot change them

2. **Supervisor mode**: can use and change Base and Bound Registers
   - Also need Mode Bit (0=User, 1=Supervisor) to determine processor mode
   - Also need way for program in User Mode to invoke operating system in Supervisor Mode, and vice versa
As programs come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage. Easy way to do this?
Idea #2: Page Tables to avoid Fragmentation

• Divide memory address space into equal sized blocks, called **pages**
  – Traditionally 4 KB or 8 KB
• Use a *level of indirection* to map program addresses into physical memory addresses
  – One indirection mapping per address space page
• This table of mappings is called a **Page Table**
Paged Memory Systems

- Processor-generated address is split into:
  - Offset indexes which byte in the page.
  - Page # indexes which page in address space.
  - Page Table maps this **Virtual Page Number** (VPN) to a **Physical Page Number** (PPN), a page in physical memory.

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>

32-bit byte address
4096 byte pages
Paged Memory Systems

- **Page table** contains the physical address of the base of each page:

```
Virtual Address Space

<table>
<thead>
<tr>
<th>Code</th>
<th>Static</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>V</td>
</tr>
</tbody>
</table>

\^ | Stack

Page Table (contains VPN => PPN mapping)

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

This Address Space consists of 8x 4K Byte pages or 16768 Bytes

Page tables make it possible to store the pages of a program non-contiguously.
Separate Address Space per Program

- Each program has its own page table
- Page table contains an entry for each program page
Paging Terminology

• Program addresses called *virtual addresses*
  – Space of all virtual addresses called *virtual memory*
  – Divided into pages indexed by VPN.

• Memory addresses called *physical addresses*
  – Space of all physical addresses called *physical memory*
  – Divided into pages indexed by PPN.
Processes and Virtual Memory

• Allows multiple programs to simultaneously occupy memory and provides protection – don’t let one program read/write memory from another
  – Each program called a Process, like a thread but has its own memory space.
  – Each has own PC, stack, heap

• Address space – give each program the illusion that it has its own private memory
  – Suppose code starts at address 0x00400000. But different processes have different code, both residing at the same (virtual) address. So each program has a different view of memory.
Protection via Page Table

• **Access Rights** checked on every access to see if allowed
  – Read: can read, but not write page
  – Read/Write: read or write data on page
  – Execute: Can fetch instructions from page

• **Valid** = Valid page table entry
  – When the virtual page can be found in physical memory.
  – (vs. found on disk [discussed later] or not yet allocated)
Administrivia

• Project 3 released, due 8/7 @ midnight
  – Work individually
• HW4 (Map Reduce) cancelled, reduced to a single lab
• Final Review - Monday, 8/8.
• Final Exam - Thursday, 8/11, 9am - 12pm 2050 VLSB
  – Part midterm material, part material since.
  – Midterm clobber policy in effect
  – Green sheet provided
  – Two-sided handwritten cheat sheet
    • Use the back side of your midterm cheat sheet!
"Only the bolded text in the notes will be on the test"

Everything is bolded

http://www.geekosystem.com/engineering-professor-meme/2/
More Depth on Page Tables

Virtual Address:

- **page no.**
- **offset**

20 bits

12 bits

Page Table

<table>
<thead>
<tr>
<th>V</th>
<th>A.R.</th>
<th>P. P. N.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val-id</td>
<td>Access Rights</td>
<td>Physical Page Number</td>
</tr>
</tbody>
</table>

index into page table

concatenate

Physical Memory Address

8/1/2011

Summer 2011 -- Lecture #24
Every instruction and data access needs address translation and protection checks.
Patterson’s Analogy

• Book title like *virtual address*
• Library of Congress call number like *physical address*
• Card catalogue like *page table*, mapping from book title to call #
• On card, available for 2-hour in library use (vs. 2-week checkout) like *access rights*
Agenda

• Virtual Memory Intro
• Page Tables
• Administrivia
• Translation Lookaside Buffer
• Break
• Demand Paging
• Putting it all Together
• Summary
Where Should Page Tables Reside?

• Space required by the page tables is proportional to the address space, number of users, ...
  – Space requirement is large:
    e.g., $2^{32}$ byte address space, $2^{12}$ byte pages
    = $2^{20}$ table entries
    = $1024 \times 1024$ entries (per process!)
  – How many bits per page table entry?
  – Too expensive to keep in processor registers!
Where Should Page Tables Reside?

• Idea: Keep page tables in the main memory
  – Keep physical address of page table in Page Table Base Register.
  – One access to retrieve the physical page address from table.
  – Second memory access to retrieve the data word
  – *Doubles* the number of memory references!

• Why is this bad news?
Page Tables Can Be HUGE: Put Them In Physical Memory

User 1 Virtual Address Space

User 2 Virtual Address Space

VA1

PT User 1

PT User 2

Physical Memory
Virtual Memory Without Doubling Memory Accesses

• Temporal and Spatial locality of Page Table accesses?
  – (Not a huge amount of spatial locality since pages are so big)

• Build a separate cache for entries in the Page Table!

• For historical reasons, called *Translation Lookaside Buffer* (TLB)
  – More accurate name is *Page Table Address Cache*
TLBs vs. Caches

Data / Instruction Cache

- Memory Address
- Data at memory address
- On miss: Access next cache level / main memory

TLB

- Virtual Page Number
- Physical Page Number
- On miss: Access Page Table in memory
Translation Lookaside Buffers (TLB): More Detail

Cache Page Table Entries in TLB

TLB hit  => *Single Cycle Translation*
TLB miss => *Access Page-Table to fill*

8/1/2011

Summer 2011 -- Lecture #24
TLB Design

• Typically 32-128 entries
  – Usually fully/highly associative: why wouldn’t direct mapped work?
  – Each entry maps a large page, hence less spatial locality across pages

• A memory management unit (MMU) is hardware that walks the page tables and reloads the TLB
Agenda

• Virtual Memory Intro
• Page Tables
• Administrivia
• Translation Lookaside Buffer
• Break
• Demand Paging
• Putting it all Together
• Summary
Agenda

• Virtual Memory Intro
• Page Tables
• Administrivia
• Translation Lookaside Buffer
• Break
• Demand Paging
• Putting it all Together
• Summary
Historical Retrospective: 1960 versus 2010

• Memory used to be very expensive, and amount available to the processor was highly limited
  – Now memory is cheap: <$10 per GByte

• Many apps’ data could not fit in main memory, e.g., payroll
  – Paged memory system reduced fragmentation but still required whole program to be resident in the main memory
  – For good performance, buy enough memory to hold your apps

• Programmers moved the data back and forth from the disk store by overlaying it repeatedly on the primary store
  – Programmers no longer need to worry about this level of detail anymore
Demand Paging in Atlas (1962)

“A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor.”

*Tom Kilburn*

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage
Demand Paging

- What if required pages no longer fit into Physical Memory?
  - Think running out of RAM on a machine
- Physical memory becomes a cache for disk.
  - Page not found in memory => *Page Fault*, must retrieve it from disk.
  - Memory full? Invoke replacement policy to swap pages back to disk.
Demand Paging Scheme

- On a *page fault*:
  - Allocate a free page in memory, if available.
  - If no free pages, invoke replacement policy to select page to swap out.
  - Replaced page is written to disk
  - *Page table is updated* - The entry for the replaced page is marked as invalid. The entry for the new page is filled in.
Demand Paging

• OS must reserve Swap Space on disk for each process
  – Place to put swapped out pages.
• To grow a process, ask Operating System
  – If unused pages available, OS uses them first
  – If not, OS swaps some old pages to disk
    – (Least Recently Used to pick pages to swap)
• How/Why grow a process?
Impact on TLB

• Keep track of whether page needs to be written back to disk if it's been modified
• Set “Page Dirty Bit” in TLB when any data in page is written
• When TLB entry replaced, corresponding Page Dirty Bit is set in Page Table Entry
Modern Virtual Memory Systems

*Illusion of a large, private, uniform store*

**Protection & Privacy**

Several users, each with their private address space and one or more shared address spaces.

**Demand Paging**

- Provides ability to run programs larger than the primary memory.
- Hides differences in machine configurations.

*Price is address translation on each memory reference;*

*And disk so slow that performance suffers if going to disk all the time (“thrashing”)*
Hardware/Software Support for Memory Protection

• Different tasks can share parts of their virtual address spaces
  – But need to protect against errant access
  – Requires OS assistance

• Hardware support for OS protection
  – Privileged supervisor mode (aka *kernel mode*)
  – Privileged instructions
  – Page tables and other state information only accessible in supervisor mode
  – System call exception (e.g., syscall in MIPS)
Just Another Level in the Memory Hierarchy

Virtual Memory:
- Pages
- Blocks
- L2 Cache
- Cache
- Regs

Upper Level: Faster
Lower Level: Larger

Memory Hierarchy:
- Files
- Pages
- Blocks
- Instr. Operands
- Regs

Date: 8/1/2011
Lecture: #24
Caching vs. Demand Paging

**Caching**
- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled in *hardware*

**Demand paging**
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in *software*
Agenda

• Virtual Memory Intro
• Page Tables
• Administrivia
• Translation Lookaside Buffer
• Break
• Demand Paging
• Putting it all Together
• Summary
Virtual Memory and Caching

- Two options:
  - Perform address translation BEFORE caching
    - Cache operates on physical address.
    - This is mostly what we’ll talk about.
  - Perform address translation AFTER caching
    - Cache operates on virtual address.
    - Must either clear cache on context switch or store Process ID with the Tag.
Virtual Memory and Caching

Virtual Address

TLB Lookup

Restart instruction

Page Table Walk

Page Fault
(OS loads page)

Update TLB

Protection Fault

Protection Check

hit

miss

Permitted

denied

≠ Memory

∈ memory

hardware

hardware or software

software

Physical Address (to cache)

SEGFAULT

8/1/2011

Summer 2011 -- Lecture #24
Address Translation in CPU Pipeline

- Need mechanisms to cope with the additional latency of a TLB:
  - Slow down the clock
  - Pipeline the TLB and cache access
  - Virtual address caches (indexed with virtual addresses)
Third Option: Concurrent Access to TLB & Phys. Addressed Cache

Index I is available without consulting the TLB

⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

**Cases:**

\[ I + O = k \quad I + O < k \quad I + O > k \]
Nehalem Virtual Memory Details

- 48-bit virtual address space, 34-bit physical address space
- Two-level TLB: L1 + L2
- I-TLB (L1) has 128 entries 4-way associative for 4KB pages, plus 7 dedicated fully-associative entries per SMT thread for large page (2/4MB) entries
- D-TLB (L1) has 64 entries for 4KB pages and 32 entries for 2/4MB pages, both 4-way associative, dynamically shared between SMT threads
- Unified L2 TLB has 512 entries for 4KB pages only, also 4-way associative
- Data TLB Reach
  (4 KB only) L1: 64*4 KB =0.25 MB, L2:512*4 KB= 2MB (superpages) L1: 32 *2-4 MB = 64-128 MB
Using Large Pages from Application?

• Difficulty is communicating from application to operating system that want to use large pages
• Linux: “Huge pages” via a library file system and memory mapping; beyond 61C
  – See http://lwn.net/Articles/375096/
• Max OS X: no support for applications to do this (OS decides if should use or not)
And in Conclusion, ...

- Virtual Memory supplies two features:
  - *Translation* (mapping of virtual address to physical address)
  - *Protection* (permission to access word in memory)
  - Most modern systems provide support for all functions with a single page-based system

- All desktops/servers have full demand-paged Virtual Memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks

- Hardware support: User/Supervisor Mode, invoke Supervisor Mode, TLB, Page Table Register