New-School Machine Structures

**Big Idea: Memory Hierarchy**

- 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

---

**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!”

---

**Agenda**

- HW Memory Protection
- Administrivia
- Virtual Memory
- Translation Lookaside Buffer
- Technology Break
- Another View of Virtual Memory
- Summary
Review: C Memory Management

- C has three pools of data memory (+ code memory)
  - 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
- Common (Dynamic) Memory Problems
  - Using uninitialized values
  - Accessing memory beyond your allocated region
  - Improper use of free/realloc by messing with the pointer handles returned by malloc
  - Memory leaks: mismatched malloc/free pairs

Simplest Model

- Only one program running on the computer
  - Addresses in the program are exactly the physical memory addresses
- Extensions to the simple model:
  - What if less physical memory than full address space?
  - What if we want to run multiple programs at the same time?

Problem #1: Physical Memory Less Than the Full Address Space

- One architecture, many implementations, with possibly different amounts of memory
- Memory used to very expensive and physically bulky
- Where does the stack grow from then?

Problem #2: Multiple Programs Sharing the Machine’s Address Space

- How can we run multiple programs without accidentally stepping on same addresses?
- How can we protect programs from clobbering each other?

Idea: Level of Indirection to Create Illusion of Large Physical Memory

One table per running application OR swap table contents when switching
Extension to the Simple Model

- Multiple programs sharing the same address space
  - E.g., operating system uses low end of address range shared with application
  - Multiple programs in shared (virtual) address space
    - Static management: fixed partitioning/allocation of space
    - Dynamic management: programs come and go, take different amount of time to execute, use different amounts of memory
- How can we protect programs from clobbering each other?
- How can we allocate memory to applications on demand?

First Idea: Base + Bounds Registers for Location Independence

Location-independent programs Programming and storage management ease: need for a base register

Protection Independent programs should not affect each other inadvertently: need for a bound register

Historically, base + bounds registers were a very early idea in computer architecture

Static Division of Shared Address Space

- E.g., how to manage the carving up of the address space among OS and applications?
- Where does the OS end and the application begin?
- Dynamic management, with protection, would be better!

Simple Base and Bound Translation

Base and bounds registers are visible/accessible to programmer Trap to OS if bounds violation detected (“seg fault”/“core dumped”)

Restriction on Base + Bounds Regs

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

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 Memory 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 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:
  - Page number
  - Offset

• Page table contains the physical address of the base of each page:

12-bit byte address
4096 byte pages

Page number
Offset

Physical Memory

Program consists of
4x 4K Byte pages or
16384 Bytes

Virtual Address Space

Page Table (think of an array of base registers or pointers)

Page tables make it possible to store the pages of a program non-contiguously.

Separate Address Space per Program

• Each program has 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

• Memory addresses called physical addresses
  – Space of all physical addresses called physical memory

Processes and Virtual Memory

• Allow multiple processes to simultaneously occupy memory and provide protection – don’t let one program read/write memory from another
  – Each has own PC, stack, heap
  – Like threads, except processes have separate address spaces

• Address space – give each program the illusion that it has its own private memory
  – Suppose code starts at address 0x40000000. But different processes have different code, both residing at the same (virtual) address. So each program has a different view of memory.

Combine Idea #1 and Idea #2: 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

More Depth on Page Tables

Virtual Address:

Page Table

Page Table located in physical memory
Address Translation & Protection

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*

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 program!)
  - How many bits per page table entry?
  - Too expensive to keep in processor registers!

Page Tables Can Be HUGE: Put Them In Physical Memory

- Caches suggest that there is temporal and spatial locality of data
- Locality of data really means locality of addresses of that data
- What about locality of translations of virtual page addresses into physical page addresses?
- For historical reasons, called *Translation Lookaside Buffer (TLB)*
  - More accurate name is *Page Table Address Cache*

Virtual Memory Without Doubling Memory Accesses

- Idea: Keep page tables in the main memory
  - One reference to retrieve the page base address from table in memory
  - Second memory access to retrieve the data word
  - Doubles the number of memory references!
- Why is this bad news?
Agenda

- HW Memory Protection
- Administrivia
- Virtual Memory
- Translation Lookaside Buffer
- Technology Break
- Another View of Virtual Memory
- Summary

Translation Lookaside Buffers (TLB): Another Layer of Indirection!

Address translation is very expensive!
Each reference becomes 2 memory accesses

Solution: Cache address translations in TLB!
  
- TLB hit \implies Single Cycle Translation
- TLB miss \implies Access Page-Table to refill

<table>
<thead>
<tr>
<th>virtual address</th>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical address</td>
<td>PPN</td>
<td>offset</td>
</tr>
</tbody>
</table>

hit?  

TLB Design

- Typically 32-128 entries
  – Usually fully associative: why wouldn’t direct mapped work?

Historical Retrospective: 1960 versus 2010

- Memory used to be very expensive, and amount available to the processor was highly limited
  – Now memory is cheap: approx $20 per Gbyte in April 2011
- 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 diskstore 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 Scheme

- On a page fault:
  – Input transfer into a free page is initiated
  – If no free page available, a page is selected to be replaced (based on usage)
  – Replaced page is written on the disk
    • To minimize disk latency effect, the first empty page on the disk was selected
    • Page table is updated to point to the new location of the page on the disk
Notes on Page Table

• Solves the fragmentation problem: all chunks same size, so any holes can be used
• OS must reserve \textit{Swap Space} on disk for each process
• 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 its 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

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)

Modern Virtual Memory Systems

\textit{Illusion of a large, private, uniform store}

Protection & Privacy
Several users, each with their private address space and one or more shared address spaces
page table = name space

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;
But disk so slow that performance suffers if going to disk all the time (“thrashing”)

Agenda

• HW Memory Protection
• Administrivia
• Virtual Memory
• Translation Lookaside Buffer
• Technology Break
• Another View of Virtual Memory
• Summary

\textbf{QUIET PLEASE}
Agenda

• HW Memory Protection
• Administrivia
• Virtual Memory
• Translation Lookaside Buffer
• Technology Break
• Another View of Virtual Memory
• Summary

Another View of Virtual Memory: Just Another Part of Memory Hierarchy

• Use main memory as a “cache” for secondary (disk) storage
  – Managed jointly by CPU hardware and the operating system (OS)
• Programs share main memory
  – Each gets a private virtual address space holding its frequently used code and data
  – Protected from other programs
• CPU and OS translate virtual addresses to physical addresses
  – VM “block” is called a page
  – VM translation “miss” is called a page fault

Just Another View of Memory Hierarchy

Caching vs. Demand Paging

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

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

Address Translation: Putting it all Together

Address Translation in CPU Pipeline

• Software handlers need restartable exception on TLB fault
• Handling a TLB miss needs a hardware or software mechanism to refill TLB
• 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)
  – Parallel TLB/cache access
Concurrent Access to TLB & Cache

Index L is available without consulting the TLB
Tag comparison is made after both accesses are completed

Cases: \( L + b = k \), \( L + b < k \), \( L + b > k \)

Modern Memory Management

- Slowdown too great to run much bigger programs than memory
  - Called Throttling
  - Buy more memory or run program on bigger computer or reduce size of problem
- Paging system today still used for
  - Translation (mapping of virtual address to physical address)
  - Protection (permission to access word in memory)
  - Sharing of memory between independent tasks

Impact of Paging on AMAT

- Memory Parameters:
  - L1 cache hit = 1 clock cycles, hit 95% of accesses
  - L2 cache hit = 10 clock cycles, hit 60% of L1 misses
  - DRAM = 200 clock cycles (~100 nanoseconds)
  - Disk = 20,000,000 clock cycles (~10 milliseconds)
- Average Memory Access Time (with no paging):
  - \( 1 + 5\%*10 + 5\%*40\%*200 = 5.5 \) clock cycles
- Average Memory Access Time (with paging) = AMAT (with no paging) + ?
  - 5.5 + ?

Nehalem Virtual Memory Details

- 48-bit virtual address space, 40-bit physical address space
- Two-level TLB: L1 + L2
- I-TLB (L1) has shared 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 * 16 KB = 0.25 MB, L2: 512 * 4 KB = 2MB (superpages) L1: 32 * 2-4 MB = 64-128 MB

Impact of TLBs on Performance

- Each TLB miss to Page Table ~ L1 Cache miss
- Page sizes are 4 KB to 8 KB (4 KB on x86)
- TLB has typically 128 entries
  - Set Associative or Fully Associative
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB:
  - 128 * 4 KB = 512 KB = 0.5 MB
- What can you do to have better performance?

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, ...

- Separate Memory Management into orthogonal functions:
  - Translation (mapping of virtual address to physical address)
  - Protection (permission to access word in memory)
  - But most modern systems provide support for all functions with 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
  - Simplifies implementation of some OS features
- Hardware support: User/Supervisor Mode, invoke Supervisor Mode, TLB, Page Table Register

Every instruction and data access needs address translation and protection checks

Good VM design needs to be fast (~ one cycle) and space efficient