Memory Management

Goals of this Lecture

• Help you learn about:
  • The memory hierarchy
  • Spatial and temporal locality of reference
  • Caching, at multiple levels
  • Virtual memory
  • … and thereby …
  • How the hardware and OS give application pgms:
    • The illusion of a large contiguous address space
    • Protection against each other

Virtual memory is one of the most important concepts in systems programming
Motivation for Memory Hierarchy

• Faster storage technologies are more costly
  • Cost more money per byte
  • Have lower storage capacity
  • Require more power and generate more heat

• The gap between processing and memory is widening
  • Processors have been getting faster and faster
  • Main memory speed is not improving as dramatically

• Well-written programs tend to exhibit good locality
  • Across time: repeatedly referencing the same variables
  • Across space: often accessing other variables located nearby

Want the speed of fast storage at the cost and capacity of slow storage. Key idea: memory hierarchy!

Simple Three-Level Hierarchy

• Registers
  • Usually reside directly on the processor chip
  • Essentially no latency, referenced directly in instructions
  • Low capacity (e.g., 32-512 bytes)

• Main memory
  • Around 100-300 clock cycles
  • Constant access time for any memory location
  • Modest capacity (e.g., 512 MB-64GB)

• Disk
  • Around 100,000 times slower than main memory
  • Faster when accessing many bytes in a row
  • High capacity (e.g., 200+ GB)
Widening Processor/Memory Gap

- Gap in speed increasing from 1986 to 2000
  - CPU speed improved ~55% per year
  - Main memory speed improved only ~10% per year
- Main memory as major performance bottleneck
  - Many programs stall waiting for reads and writes to finish
- Changes in the memory hierarchy
  - Increasing the number of registers
    - 8 integer registers in the x86 vs. 128 in the Itanium
  - Adding caches between registers and main memory
    - On-chip level-1&2 caches and (off-chip?) level-3 cache

An Example Memory Hierarchy

- Smaller, faster, and costlier (per byte) storage devices
- Larger, slower, and cheaper (per byte) storage devices
Locality of Reference

• Two kinds of locality
  • Temporal locality: recently referenced items are likely to be referenced in near future
  • Spatial locality: Items with nearby addresses tend to be referenced close together in time.

Locality example

• Program data
  • Temporal: the variable sum
  • Spatial: variable $a[i+1]$ accessed soon after $a[i]$

• Instructions
  • Temporal: cycle through the for-loop repeatedly
  • Spatial: reference instructions in sequence

Locality Makes Caching Effective

• Cache
  • Smaller, faster storage device that acts as a staging area
  • … for a subset of the data in a larger, slower device

• Caching and the memory hierarchy
  • Storage device at level k is a cache for level k+1
  • Registers as cache of L1/L2 cache and main memory
  • Main memory as a cache for the disk
  • Disk as a cache of files from remote storage

• Locality of access is the key
  • Most accesses satisfied by first few (faster) levels
  • Very few accesses go to the last few (slower) levels
Caching in a Memory Hierarchy

Smaller, faster, more expensive device at level k caches a subset of the blocks from level k+1.

Data copied between levels in block-sized transfer units.

Larger, slower, cheaper storage device at level k+1 is partitioned into blocks.

Cache Block Sizes

- Fixed vs. variable size
  - Fixed-sized blocks are easier to manage (common case)
  - Variable-sized blocks make more efficient use of storage

- Block size
  - Depends on access times at the level k+1 device
  - Larger block sizes further down in the hierarchy
  - E.g., disk seek times are slow, so disk pages are larger

- Examples
  - CPU registers: 4-byte words
  - L1/L2 cache: 32-byte blocks
  - Main memory: 4 KB pages
  - Disk: entire files
Cache Hit and Miss

- **Cache hit**
  - Program accesses a block available in the cache
  - Satisfy directly from cache
  - E.g., request for “10”

- **Cache miss**
  - Program accesses a block not available in the cache
  - Bring item into the cache
  - E.g., request for “13”

- Where to place the item?
- Which item to evict?

Three Kinds of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the block hasn’t been accessed before
  - E.g., first time a segment of code is executed
  - E.g., first time a particular array is referenced

- **Capacity miss**
  - Set of active cache blocks (the “working set”) is larger than cache
  - E.g., manipulating a 1200-byte array within a 1000-byte cache

- **Conflict miss**
  - Some caches limit the locations where a block can be stored
  - E.g., block i must be placed in cache location (i mod 4)
  - Conflicts occur when multiple blocks map to the same location(s)
  - E.g., referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time
Cache Replacement

- Evicting a block from the cache
  - New block must be brought into the cache
  - Must choose a “victim” to evict

- Optimal eviction policy
  - Evict a block that is never accessed again
  - Evict the block accessed the furthest in the future
  - Impossible to implement without knowledge of the future

- Using the past to predict the future
  - Evict the “least recently used” (LRU) block
  - Assuming it is not likely to be used again soon

- But, LRU is often expensive to implement
  - Need to keep track of access times
  - So, simpler approximations of LRU are used

Who Manages the Cache?

- Registers
  - Cache of L1/L2 cache and main memory
  - Managed explicitly by the compiler
  - By determining which data are brought in and out of registers
  - Using relatively sophisticated code-analysis techniques

- L1/L2 cache
  - Cache of main memory
  - Managed by the hardware
  - Using relatively simple mechanisms (e.g., “i mod 4”)

- Main memory
  - Cache of the disk
  - Managed (in modern times) by the operating system
  - Using relatively sophisticated mechanisms (e.g., LRU-like)
  - Since reading from disk is extremely time consuming
Manual Allocation: Segmentation

- In the olden days (aka “before the mid 1950s”)
  - Programmers incorporated storage allocation in their programs
  - … whenever the total information exceeded main memory

- Segmentation
  - Programmers would divide their programs into “segments”
  - Which would “overlay” (i.e., replace) one another in main memory

- Advantages
  - Programmers are intimately familiar with their code
  - And can optimize the layout of information in main memory

- Disadvantages
  - Immensely tedious and error-prone
  - Compromises the portability of the code

Automatic Allocation: Virtual Memory

- Give programmer the illusion of a very large memory
  - Large: 4 GB of memory with 32-bit addresses
  - Uniform: contiguous memory locations, from 0 to $2^{32}-1$

- Independent of
  - The actual size of the main memory
  - The presence of any other processes sharing the computer

- Key idea #1: separate “address” from “physical location”
  - Virtual addresses: generated by the program
  - Memory locations: determined by the hardware and OS

- Key idea #2: caching
  - Swap virtual pages between main memory and the disk

One of the greatest ideas in computer systems!
Making Good Use of Memory and Disk

• Good use of the disk
  • Read and write data in large “pages”
  • … to amortize the cost of “seeking” on the disk
  • E.g., page size of 4 KB

• Good use of main memory
  • Even though the address space is large
  • … programs usually access only small portions at a time
  • Keep the “working set” in main memory
    • Demand paging: only bring in a page when needed
    • Page replacement: selecting good page to swap out

• Goal: avoid thrashing
  • Continually swapping between memory and disk

Virtual Address for a Process

• Virtual page number
  • Number of the page in the virtual address space
  • Extracted from the upper bits of the (virtual) address
  • … and then mapped to a physical page number

• Offset in a page
  • Number of the byte within the page
  • Extracted from the lower bits of the (virtual) address
  • … and then used as offset from start of physical page

• Example: 4 KB pages
  • 20-bit page number: $2^{20}$ virtual pages
  • 12-bit offset: bytes 0 to $2^{12} - 1$
Virtual Memory for a Process

Translate virtual page number to physical page number

Virtual Address Space  Physical Address Space

32-bit address

virtual page number

offset in page

physical page number

offset in page

Page Table to Manage the Cache

- Current location of each virtual page
  - Physical page number, or
  - Disk address (or null if unallocated)

- Example
  - Page 0: at location xx on disk
  - Page 1: at physical page 2
  - Page 3: not yet allocated

- Page “hit” handled by hardware
  - Compute the physical address
    - Map virtual page # to physical page #
    - Concatenate with offset in page
  - Read or write from main memory
    - Using the physical address

- Page “miss” triggers an exception…
“Miss” Triggers Page Fault

- Accessing page not in main memory

<table>
<thead>
<tr>
<th>V</th>
<th>Physical or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>xx</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>yy</td>
</tr>
<tr>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Virtual page #2 at location yy on disk!

movl 0002104, %eax

OS Handles the Page Fault

- Bringing page in from disk
  - If needed, swap out old page (e.g., #4)
  - Bring in the new page (page #2)
  - Update the page table entries
VM as a Tool for Memory Protection

- **Memory protection**
  - Prevent process from unauthorized reading or writing of memory

- **User process should not be able to**
  - Modify the read-only text section in its own address space
  - Read or write operating-system code and data structures
  - Read or write the private memory of other processes

- **Hardware support**
  - Permission bits in page-table entries (e.g., read-only)
  - Separate identifier for each process (i.e., process-id)
  - Switching between *unprivileged* mode (for user processes) and *privileged* mode (for the operating system)

---

Sharing Physical Memory

![Diagram showing sharing of physical memory between processes and the operating system.](Diagram.png)
Processes and Page Table Entries

Page Tables in OS Memory...
How to Size Page Tables

• 4GB of virtual address space
• 4KB per page = 1 million entries
• 4 bytes/entry = 4MB per virtual address space
• Lots (hundreds) of processes – huge consumption

How do you fix this?
• Observation: invalid pages have an entry of zero
• Observation: most invalid pages are contiguous

Solution: hierarchical page table
• An array of pointers to arrays
• If lower-level array is all invalid, just zero-out pointer

Question: how many non-zero entries in upper level?

Measuring the Memory Usage

Virtual memory usage
Physical memory usage ("resident set size")
CPU time used by this process so far

Unix

Windows
VM as a Tool for Memory Management

- **Simplifying linking**
  - Same memory layout for each process
    - E.g., text section always starts at 0x08048000
    - E.g., stack always grows down from 0x0bfffffff
  - Linker can be independent of physical location of code

- **Simplifying sharing**
  - User processes can share some code and data
    - E.g., single physical copy of stdio library code (like printf)
  - Mapped in to the virtual address space of each process

- **Simplifying memory allocation**
  - User processes can request additional memory from the heap
    - E.g., using `malloc()` to allocate, and `free()` to deallocate
  - OS allocates *contiguous virtual pages*...
    - … and scatters them *anywhere* in physical memory

Summary

- **Memory hierarchy**
  - Memory devices of different speed, size, and cost
    - Registers, on-chip cache, off-chip cache, main memory, disk, tape
  - Locality of memory accesses making caching effective

- **Virtual memory**
  - Separate virtual address space for each process
  - Provides caching, memory protection, and memory management
  - Implemented via cooperation of the address-translation hardware and the OS (when page faults occur)

- **In Dynamic Memory Management lectures:**
  - Dynamic memory allocation on the heap
  - Management by user-space software (e.g., `malloc()` and `free()`)

30