Storage Management
Jennifer Rexford

Goals of this Lecture
Help you learn about:
• Locality and caching
• Typical storage hierarchy
• Virtual memory
  • How the hardware and OS give applications the illusion of a large, contiguous, private address space

Virtual memory is one of the most important concepts in system programming

Agenda
Locality and caching
Typical storage hierarchy
Virtual memory

Improving Storage Device Performance
Facts:
• CPU performance is improving quickly
• Storage device performance is improving slowly
• Example:
  • Gap between CPU speed and main memory (RAM) performance is widening
  • Main memory (RAM) is performance bottleneck
  • Many programs stall CPU waiting for loads/stores

Conclusion:
• To improve overall performance, must improve storage device performance

Improving Storage Performance
Classes of storage devices:
• Fast access & small capacity
• Slow access & large capacity

We want:
• Fast access & large capacity
• But how???

The key: locality allows caching
• Most programs exhibit good locality
• A program that exhibits good locality will benefit from proper caching

Locality
Two kinds of locality
• Temporal locality
  • If a pgm references item X now, it probably will reference X again soon
• Spatial locality
  • If a pgm references item X now, it probably will reference items in storage nearby X soon

Most programs exhibit good temporal and spatial locality
Locality Example

- **Temporal locality**
  - Data: Whenever the CPU accesses `sum`, it accesses `sum` again shortly thereafter.
  - Instructions: Whenever the CPU executes `sum += a[i]`, it executes `sum += a[i]` again shortly thereafter.
- **Spatial locality**
  - Data: Whenever the CPU accesses `a[i]`, it accesses `a[i+1]` shortly thereafter.
  - Instructions: Whenever the CPU executes `sum += a[i]`, it executes `i++` shortly thereafter.

Typical code (good locality):

```c
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
```

Caching

- **Cache**
  - Fast access, small capacity storage device
  - Acts as a staging area for a subset of the items in a slow access, large capacity storage device

**Good locality + proper caching**

- => Most storage accesses can be satisfied by cache
- => Overall storage performance improved

Caching in a Storage Hierarchy

- Level `k`:
  - Smaller, faster device at level `k` caches a subset of the blocks from level `k+1`.
  - Blocks copied between levels.
- Level `k+1`:
  - Larger, slower device at level `k+1` is partitioned into blocks.

Cache Hits and Misses

- **Cache hit**
  - E.g., request for block 10.
  - Access block 10 at level `k`.
  - Fast!
- **Cache miss**
  - E.g., request for block 8.
  - Evict some block from level `k` to level `k+1`.
  - Load block 8 from level `k+1` to level `k`.
  - Access block 8 at level `k`.
  - Slow!

Caching goal:
- Maximize cache hits.
- Minimize cache misses.

Cache Eviction Policies

- **Best eviction policy**: "clairvoyant" policy
  - Always evict a block that is never accessed again, or...
  - Always evict the block accessed the furthest in the future.
  - Impossible in the general case.
- **Worst eviction policy**
  - Always evict the block that will be accessed next!
  - Causes thrashing.
  - Impossible in the general case.

**Reasonable eviction policy**: LRU policy

- Evict the "least recently used" (LRU) block.
- With the assumption that it will not be used again (soon).
- Good for straight-line code.
- Bad for loops.
- Expensive to implement.
- Often simpler approximations are used.
- See Wikipedia "Page replacement algorithm" topic.

Cache Eviction Policies

- Level `k`:
  - Blocks copied between levels.
- Level `k+1`:
  - Level `k` is a cache for level `k+1`.

Caching goal:
- Maximize cache hits.
- Minimize cache misses.
Locality/Caching Example: Matrix Mult

Matrix multiplication
- Matrix = two-dimensional array
- Multiply n-by-n matrices A and B
- Store product in matrix C

Performance depends upon
- Effective use of caching (as implemented by system)
- Good locality (as implemented by you)

Two-dimensional arrays are stored in either row-major or column-major order

C uses row-major order
- Access in row-major order => good spatial locality
- Access in column-major order => poor spatial locality

for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      for (k=0; k<n; k++)
         c[i][j] += a[i][k] * b[k][j];

Reasonable cache effects
- Good locality for A
- Bad locality for B
- Good locality for C

Poor cache effects
- Bad locality for A
- Bad locality for B
- Bad locality for C

for (j=0; j<n; j++)
   for (k=0; k<n; k++)
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * b[k][j];

Good cache effects
- Good locality for A
- Good locality for B
- Good locality for C

Agenda

Locality and caching
Typical storage hierarchy
Virtual memory
Typical Storage Hierarchy

Registers
- Latency: 0 cycles
- Capacity: 8-256 registers
  - 8 general purpose registers in IA-32; 128 in Itanium

L1/L2/L3 Cache
- Latency: 1 to 30 cycles
- Capacity: 32KB to 32MB

Main memory (RAM)
- Latency: ~100 cycles
  - 100 times slower than registers
- Capacity: 256MB to 64GB

Local secondary storage (local disks, SSDs)
- Latency: ~100,000 cycles
  - 1000 times slower than main memory
  - Limited by nature of disk
    - Must move heads and wait for data to rotate under heads
    - Faster when accessing many bytes in a row
- Capacity: 1GB to 256TB

Remote secondary storage
- Latency: ~10,000,000 cycles
  - 100 times slower than disk
  - Limited by network bandwidth
- Capacity: essentially unlimited

Aside: Persistence

Another dimension: persistence
- Do data persist in the absence of power?

Lower levels of storage hierarchy store data persistently
- Remote secondary storage
- Local secondary storage

Higher levels of storage hierarchy do not store data persistently
- Main memory (RAM)
- L1/L2/L3 cache
- Registers

Admirable goal: Move persistence upward in hierarchy

Solid state (flash) drives
- Use solid state technology (as does main memory)
- Persistent, as is disk
- Viable replacement for disk as local secondary storage
Storage Hierarchy & Caching Issues

Issue: Block size?
- Slow data transfer between levels $k$ and $k+1$
  - => use large block sizes at level $k$ (do data transfer less often)
- Fast data transfer between levels $k$ and $k+1$
  - => use small block sizes at level $k$ (reduce risk of cache miss)
- Lower in pyramid => slower data transfer => larger block sizes

<table>
<thead>
<tr>
<th>Device</th>
<th>Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>8 bytes</td>
</tr>
<tr>
<td>L1/L2/L3 cache line</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Main memory page</td>
<td>4KB (4096 bytes)</td>
</tr>
<tr>
<td>Disk block</td>
<td>4KB (4096 bytes)</td>
</tr>
<tr>
<td>Disk transfer block</td>
<td>4KB (4096 bytes) to 64MB (6708864 bytes)</td>
</tr>
</tbody>
</table>

Storage Hierarchy & Caching Issues

Issue: Who manages the cache?

<table>
<thead>
<tr>
<th>Device</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers (cache of L1/L2/L3 and main memory)</td>
<td>Compiler, using complex code-analysis techniques</td>
</tr>
<tr>
<td>Main memory (cache of main memory)</td>
<td>Assembly lang programmer</td>
</tr>
<tr>
<td>Main memory (cache of local sec storage)</td>
<td>Hardware, using simple algorithms</td>
</tr>
<tr>
<td>Local secondary storage (cache of remote sec storage)</td>
<td>End user, by deciding which files to download</td>
</tr>
</tbody>
</table>

Agenda

Locality and caching
Typical storage hierarchy
Virtual memory

Main Memory: Illusion

Each process sees main memory as
- Huge: $2^{64} = 16$ EB (16 exabytes) of memory
- Uniform: contiguous memory locations from 0 to $2^{64} - 1$

Main Memory: Reality

Virtual & Physical Addresses

Question
- How do OS and hardware implement virtual memory?
Answer (part 1)
- Distinguish between virtual addresses and physical addresses
Virtual & Physical Addresses (cont.)

Virtual address
- Identifies a location in a particular process’s virtual memory
- Independent of size of physical memory
- Independent of other concurrent processes
- Consists of virtual page number & offset
- Used by application programs

Physical address
- Identifies a location in physical memory
- Consists of physical page number & offset
- Known only to OS and hardware

Note:
- Offset is same in virtual addr and corresponding physical addr

Page Tables

Question
- How do OS and hardware implement virtual memory?

Answer (part 2)
- Maintain a page table for each process

Virtual Memory Example 1

Process 1234 accesses mem at virtual addr 16386
16386 = 00000000000000001010110100001010
Virtual page num = 4; offset = 2
Virtual Memory Example 1 (cont.)

Hardware consults page table
Hardware notes that virtual page 4 maps to phys page 1
Page hit!

Virtual Memory Example 1 (cont.)

Hardware forms physical addr
Physical page num = 1; offset = 2
= 00000000000000010000000000010
= 4098
Hardware fetches/stores data from/to phys addr 4098

Virtual Memory Example 2

Process 1234 accesses mem at virtual addr 8200
8200 = ...
Virtual page num = 2; offset = 8

Virtual Memory Example 2 (cont.)

Hardware consults page table
Hardware notes that virtual page 2 maps to spot X on disk
Page miss!
Hardware generates page fault

Virtual Memory Example 2 (cont.)

OS gains control of CPU
OS swaps virtual pages 6 and 2
OS updates page table accordingly
Control returns to process 1234
Process 1234 re-executes same instruction
Virtual Memory Example 2 (cont.)

Hardware consults page table
Hardware notes that virtual page 2 maps to phys page 3
Page hit!

Virtual Memory Example 2 (cont.)

Hardware forms physical addr
Physical page num = 3; offset = 8
= 00000000000000010000000000000000
= 12296
Hardware fetches/stores data from/to phys addr 12296

Virtual Memory Example 3

Process 1234 accesses mem at virtual addr 4105
4105 = ...
Virtual page num = 1; offset = 9

Virtual Memory Example 3 (cont.)

Hardware consults page table
Hardware notes that virtual page 1 is unmapped
Page miss!
Hardware generates segmentation fault
(See Signals lecture for remainder!)

Storing Page Tables

Question
• Where are the page tables themselves stored?

Answer
• In main memory

Question
• What happens if a page table is swapped out to disk?

Answer
• OS is responsible for swapping
• Special logic in OS "pins" page tables to physical memory
• So they never are swapped out to disk

Storing Page Tables (cont.)

Question
• Doesn't that mean that each logical memory access requires two physical memory accesses – one to access the page table, and one to access the desired datum?

Answer
• Yes!

Question
• Isn't that inefficient?

Answer
• Not really…
Storing Page Tables (cont.)

Note 1
- Page tables are accessed frequently
- Likely to be cached in L1/L2/L3 cache

Note 2
- X86-64 architecture provides special-purpose hardware support for virtual memory...

Translation Lookaside Buffer

Translation lookaside buffer (TLB)
- Small cache on CPU
- Each TLB entry consists of a page table entry
- Hardware first consults TLB
  - Hit => no need to consult page table in L1/L2/L3 cache or memory
  - Miss => swap relevant entry from page table in L1/L2/L3 cache or memory into TLB; try again
- See Bryant & O’Hallaron book for details

Caching again!!!
Additional Benefits of Virtual Memory

Dynamic 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 memory pages...
    - ... and scatters them anywhere in physical memory

Creating new processes
- Easy for “parent” process to “fork” a new “child” process
  - Initially: make new PCB containing copy of parent page table
  - Incrementally: change child page table entries as required
  - See Process Management lecture for details
  - `fork()` system-level function

Overwriting one program with another
- Easy for a process to replace its program with another program
  - Initially: set page table entries to point to program pages that already exist on disk!
  - Incrementally: swap pages into memory as required
  - See Process Management lecture for details
  - `execvp()` system-level function

Measuring Memory Usage

On FC010 computers:

```
$ ps l
  F   UID   PID  PPID PRI  NI VSZ RSS WCHAN STAT TTY        TIME COMMAND
0 42579  7694 20201  20   0 168040 14328 signal T    pts/0      0:00 emacs –nw
0 42579  7695 20201  20   0  11260  928 -      R+   pts/0      0:00 ps l
0 42579 20201 20200  20   0  24032  2036 wait  sx pte/0  0:00 –bash
```

VSZ (virtual memory size): virtual memory usage
RSS (resident set size): physical memory usage

Summary

Locality and caching
- Spatial & temporal locality
- Good locality => caching is effective

Typical storage hierarchy
- Registers, L1/L2/L3 cache, main memory, local secondary storage
  (esp. disk), remote secondary storage

Virtual memory
- Illusion vs. reality
- Implementation
  - Virtual addresses, page tables, translation lookaside buffer (TLB)
- Additional benefits (many!)

Virtual memory concept permeates the design of modern operating systems and computer hardware