The Memory/Storage Hierarchy and Virtual Memory

Goals of this Lecture

Help you learn about:
• Locality and caching
• The memory / storage hierarchy
• Virtual memory
  • How the hardware and OS give application programs the illusion of a large, contiguous, private address space

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

Recall this iClicker Question?

Q: What effect does virtual memory have on the speed and security of processes?

Recall this iClicker Question?

Q: What effect does virtual memory have on the speed and security of processes?

<table>
<thead>
<tr>
<th>Speed</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>A.</td>
</tr>
<tr>
<td>B.</td>
<td>C.</td>
</tr>
<tr>
<td>C.</td>
<td>no change</td>
</tr>
<tr>
<td>D.</td>
<td>D.</td>
</tr>
<tr>
<td>E.</td>
<td>E.</td>
</tr>
</tbody>
</table>

Let’s see how to make the answer:

<table>
<thead>
<tr>
<th>Speed</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>no change</td>
<td>no change</td>
</tr>
</tbody>
</table>

Agenda

Locality and caching
Typical storage hierarchy
Virtual memory

Storage Device Speed vs. Size

Facts:
• CPU needs sub-nanosecond access to data to run instructions at full speed
• Fast storage (sub-nanosecond) is small (100-1000 bytes)
• Big storage (gigabytes) is slow (15 nanoseconds)
• Huge storage (terabytes) is glacially slow (milliseconds)

Goal:
• Need many gigabytes of memory,
  • but with fast (sub-nanosecond) average access time

Solution: locality allows caching
• Most programs exhibit good locality
• A program that exhibits good locality will benefit from proper caching, which enables good average performance

Locality

Two kinds of locality
• Temporal locality
  • If a program references item X now, it probably will reference X again soon
• Spatial locality
  • If a program references item X now, it probably will reference item at address X±1 soon

Most programs exhibit good temporal and spatial locality
Locality Example

Typical code (good locality)

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

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:

Level k+1:

Blocks copied between levels

Smaller, faster device at level k caches a subset of the blocks from 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: "oracle"
- 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
- (can be) bad for loops
- Expensive to implement
- Often simpler approximations are used
- See Wikipedia "Page replacement algorithm" topic
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

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

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

Factors to consider:
- Capacity
- Latency (how long to do a read)
- Bandwidth (how many bytes/sec can be read)
  - Weakly correlated to latency: reading 1 MB from a hard disk isn’t much slower than reading 1 byte
- Volatility
  - Do data persist in the absence of power?

Typical Storage Hierarchy

Registers
- Latency: 0 cycles
- Capacity: 8-256 registers (31 general purpose registers in AArch64)

L1/L2/L3 Cache
- Latency: 1 to 40 cycles
- Capacity: 32KB to 32MB

Main memory (RAM)
- Latency: ~50-100 cycles
- 100 times slower than registers
- Capacity: GB

Cache / RAM Latency

Disks

Typical Storage Hierarchy

Local secondary storage: disk drives
- Solid-State Disk (SSD):
  - Flash memory (nonvolatile)
  - Latency: 0.1 ms (~300k cycles)
  - Capacity: 128 GB – 2 TB
- Hard Disk:
  - Spinning magnetic platters, moving heads
  - Latency: 10 ms (~30M cycles)
  - Capacity: 1 – 10 TB
Typical Storage Hierarchy

Remote secondary storage (a.k.a. “the cloud”)
- Latency: tens of milliseconds
- Limited by network bandwidth
- Capacity: essentially unlimited

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>128 bytes</td>
</tr>
<tr>
<td>Main memory page</td>
<td>4KB or 64KB</td>
</tr>
<tr>
<td>Disk block</td>
<td>512 bytes to 4KB</td>
</tr>
<tr>
<td>Disk transfer block</td>
<td>4KB (4096 bytes)</td>
</tr>
<tr>
<td></td>
<td>to 64MB (67108864 bytes)</td>
</tr>
</tbody>
</table>

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 cache and main memory)</td>
<td>Compiler, using complex code-analysis techniques</td>
</tr>
<tr>
<td></td>
<td>Assembly lang programmer</td>
</tr>
<tr>
<td>L1/L2/L3 cache (cache of main memory)</td>
<td>Hardware, using simple algorithms</td>
</tr>
<tr>
<td>Main memory (cache of local sec storage)</td>
<td>Hardware and OS, using virtual memory with complex algorithms (since accessing disk is expensive)</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 $\approx 10^{19}$
Uniform: contiguous memory locations from 0 to $2^{64} - 1$

Main Memory: Reality

Memory is divided into pages
At any time some pages are in physical memory, some on disk
OS and hardware swap pages between physical memory and disk
Multiple processes share physical memory
Virtual & Physical Addresses

Question
• How do OS and hardware implement virtual memory?

Answer (part 1)
• Distinguish between virtual addresses and physical addresses

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

CourseLab Virtual & Physical Addresses

On AArch64:
• Each virtual address consists of 64 bits
• There are $2^{64}$ bytes of virtual memory (per process)
• Each offset is either 12 or 16 bits (determined by OS) – 16 bits on armlab
• Each page consists of $2^{16}$ bytes
• Each virtual page number consists of $64 - 16 = 48$ bits
• There are $2^{48}$ virtual pages

On ArmLab:
• Each physical address consists of 37 bits
• There are $2^{37}$ bytes of physical memory (per computer)
• With 64K pages, each offset is 16 bits
• Each page consists of $2^{16}$ bytes
• Each physical page number consists of $37 - 16 = 21$ bits
• There are $2^{21}$ physical pages

Page Tables

Question
• How do OS and hardware implement virtual memory?

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

Page Table for Process 1234

<table>
<thead>
<tr>
<th>Virtual Page Num</th>
<th>Physical Page Num or Disk Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Physical page 5</td>
</tr>
<tr>
<td>1</td>
<td>(unmapped)</td>
</tr>
<tr>
<td>2</td>
<td>Spot X on disk</td>
</tr>
<tr>
<td>3</td>
<td>Physical page 8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Page table maps each in-use virtual page to:
• A physical page, or
• A spot (track & sector) on disk
Virtual Memory Example 1

Process 1234 accesses mem at virtual addr 262146 (= 0x40002)

iClicker Question coming up . . .

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
= 0x10002
= 65538
Hardware fetches/stores data from/to phys addr 65538

iClicker Question

Q: For virtual address 262146 (= 0x400002), what is the corresponding physical address?

A. 0x140002
B. 0x41002
C. 0x10002
D. 0x10000
E. 0x2

Virtual Memory Example 2

Process 1234 accesses mem at virtual addr 131080
131080 = 0x20008
Virtual page num = 2; offset = 8
Virtual Memory Example 2 (cont.)

Process 1234 accesses mem at virtual addr 131080
131080 = 0x20008
Virtual page num = 2; offset = 8

Hardware forms physical addr
Physical page num = 3; offset = 8
= 0x10000
= 196608
Hardware fetches/stores data from/to phys addr 196608

Virtual Memory Example 3

Process 1234 accesses mem at virtual addr 65545
65545 = 0x10009
Virtual page num = 1; offset = 9

Hardware consults page table
Hardware notes that virtual page 2 maps to phys page 3
Page hit!
Virtual Memory Example 3 (cont.)

<table>
<thead>
<tr>
<th>Virtual Mem</th>
<th>Physical Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 VP 3</td>
<td>0 VP 3</td>
</tr>
<tr>
<td>1 VP 4</td>
<td>1 VP 4</td>
</tr>
<tr>
<td>2 VP 0</td>
<td>2 VP 0</td>
</tr>
<tr>
<td>3 VP 2</td>
<td>3 VP 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Hardware consults page table
Hardware notes that virtual page 1 is unmapped
Page miss!
Hardware generates segmentation fault (Signals lecture!)
OS gains control, (probably) kills process

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
- It hurts! So don't do that, then!
- 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
- Conceptually, 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
- Modern hardware (including ARM) 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!!!

Recall this iClicker Question?

Q: What effect does virtual memory have on the speed and security of processes?

<table>
<thead>
<tr>
<th>Speed</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>↑</td>
</tr>
<tr>
<td>B.</td>
<td>↓</td>
</tr>
<tr>
<td>C.</td>
<td>↑</td>
</tr>
<tr>
<td>D.</td>
<td>↑</td>
</tr>
<tr>
<td>E.</td>
<td>↓</td>
</tr>
</tbody>
</table>

That’s why the real answer is:

Speed Security
no change ↑
Additional Benefits of Virtual Memory

Virtual memory concept facilitates/enables many other OS features; examples…

Context switching (as described last lecture)
- **Illusion**: To context switch from process X to process Y, OS must save contents of registers and memory for process X, restore contents of registers and memory for process Y
- **Reality**: To context switch from process X to process Y, OS must save contents of registers and **virtual memory** for process X, restore contents of registers and **virtual memory** for process Y
- **Implementation**: To context switch from process X to process Y, OS must save contents of registers and **pointer to the page table** for process X, restore contents of registers and **pointer to the page table** for process Y

Additional Benefits of Virtual Memory

Memory protection among processes
- Process’s page table references only physical memory pages that the process currently owns
- Impossible for one process to accidentally/maliciously affect physical memory used by another process

Memory protection within processes
- Permission bits in page-table entries indicate whether page is read-only, etc.
- Allows CPU to prohibit
  - Writing to RODATA & TEXT sections
  - Access to protected (OS owned) virtual memory

Linking
- Same memory layout for each process
- E.g., TEXT section always starts at virtual addr 0x4000000
- Linker is independent of physical location of code

Code and data sharing
- User processes can share some code and data
  - E.g., single physical copy of stdio library code (e.g. printf)
  - Mapped into the virtual address space of each process

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

```
$ ps l
F   UID   PID  PPID PRI  NI
VSZ RSS WCHAN STAT TTY TIME COMMAND
0 42579  9655  9696  30  10
167568 13840 signal TN pts/1 0:00 emacs
0 42579  9696  9695  30  10
24028 2072 wait SNs pts/1 0:00 -bash
0 42579  9725  9696  30  10
11268  956 -RN+ pts/1 0:00 ps l
```
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 operating systems and computer hardware