Princeton University
Computer Science 217: Introduction to Programming Systems

Goals of this Lecture

Help you learn about:
- Locality and caching
- Typical 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.

Agenda

Locality and caching
- Typical storage hierarchy
- Virtual memory

Storage Device speed vs. size

Facts:
- CPU needs subnanosecond access to memory (else it can’t run instructions fast enough)
- Fast memories (subnanosecond) are small (1000 bytes),
- Big memories (gigabytes) are slow (60 nanoseconds)
- Huge memories (terabytes) are very slow (milliseconds)

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

Solution: 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 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

Locality example

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

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**
- 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
- Blocks copied between levels

**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
- (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

```
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
- Good locality for B
- Good locality for C

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

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

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

**Agenda**
- Locality and caching
- Typical storage hierarchy
- Virtual memory

**Typical Storage Hierarchy**
- CPU registers hold words retrieved from L1/L2/L3 cache
- L1/L2/L3 cache holds cache lines retrieved from main memory
- Main memory holds disk blocks retrieved from local disks
- Local disks hold files retrieved from disks on remote network servers
- Larger slower storage devices
- Local secondary storage (local disks, SSDs)
- Remote secondary storage (distributed file systems, Web servers)
- Virtual memory
- Typical storage hierarchy
- Locality and caching
Typical Storage Hierarchy

Registers
- **Latency:** 0 cycles
- **Capacity:** 8-256 registers
  - 8 general purpose registers in IA-32;
  - 32 in typical RISC machine (ARM, MIPS, RISC-V)

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: disk drives
- **Latency:** ~100,000 cycles
  - 1000 times slower than main mem
  - 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
- **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
Aside: Persistence

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 (67108864 bytes)</td>
</tr>
</tbody>
</table>

Device Managed by:

- Registers (cache of L1/L2/L3 cache and main memory)
  - Compiler, using complex code-analysis techniques
  - Assembly lang programmer
- L1/L2/L3 cache (cache of main memory)
  - Hardware, using simple algorithms
- Main memory (cache of local sec storage)
  - Hardware and OS, using virtual memory with complex algorithms (since accessing disk is expensive)
- Local secondary storage (cache of remote sec storage)
  - End user, by deciding which files to download

Agenda

Locality and caching
典型存储层次结构
虚拟内存

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

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

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Virtual Page Num</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>52 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Addr</th>
<th>Physical Page Num</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On CourseLab:

• Each offset is 12 bits
• Each page consists of 2^12 bytes
• Each virtual page number consists of 52 bits
• There are 2^52 virtual pages
• Each virtual address consists of 64 bits
• There are 2^32 bytes of virtual memory (per process)

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>
Virtual Memory Example 1

Process 1234 accesses mem at virtual addr 16386
16386 = 000000000010B = Virtual page num = 4; offset = 2

Process 1234 Page Table: 0 VP 3, 1 VP 4, 2 VP 0, 3 VP 6

Physical Mem: 0 VP 3, 1 VP 4, 2 VP 0, 3 VP 6

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

Hardware forms physical addr
Physical page num = 1; offset = 2
= 000000010000000000000100B = 4098
Hardware fetches/stores data from/to phys addr 4098

Virtual Memory Example 2

Process 1234 accesses mem at virtual addr 8200
8200 = 00001000B = Virtual page num = 2; offset = 8

Process 1234 Page Table: 0 VP 3, 1 VP 4, 2 VP 0, 3 VP 6

Physical Mem: 0 VP 3, 1 VP 4, 2 VP 0, 3 VP 6

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

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

Virtual Memory Example 2 (cont.)

Process 1234 accesses mem at virtual addr 8200
8200 = 00001000B = Virtual page num = 2; offset = 8

OS gains control of CPU
OS swaps virtual pages 6 and 2
This takes a long while (disk latency), run another process for the time being; then eventually...
OS updates page table accordingly
Control returns to process 1234
Process 1234 re-executes same instruction
Virtual Memory Example 2 (cont.)

Process 1234 accesses mem at virtual addr 8200
8200 = \text{100000001000}_B\text{ = Virtual page num = 2; offset = 8}

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

Virtual Memory Example 3

Process 1234 accesses mem at virtual addr 4105
4105 = \text{100000001001}_B\text{ = Virtual page num = 1; offset = 9}

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

Virtual Memory Example 3 (cont.)
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...

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

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

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 page table for process X, restore contents of registers and page table for process Y

Linking
- Same memory layout for each process
  - E.g., TEXT section always starts at virtual addr 0x08048000
  - E.g., STACK always grows from virtual addr 0x0bfffffff to lower addresses
  - 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
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 CourseLab computers:

```bash
$ 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
```

VSZ (virtual memory size): virtual memory usage
RSS (resident set size): physical memory usage
(both measured in kilobytes)

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