Storage Management
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
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 the CPU waiting for loads and 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 prog references item X now, it probably will reference X again soon

- **Spatial** locality
  - If a prog references item X now, it probably will reference items in storage nearby X 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

Level k:

4  9  10  3

Level k+1:

0  1  2  3
4  5  6  7
8  9 10 11
12 13 14 15

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

Blocks copied between levels

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 \textit{never} accessed again, or…
- Always evict the block accessed the \textit{furthest in the future}
- Impossible in the general case

**Worst** eviction policy
- Always evict the block that will be accessed next!
- Causes \textit{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
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
Locality/Caching Example: Matrix Mult

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
Locality/Caching Example: Matrix Mult

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];

Poor cache effects
- Bad locality for A
- Bad locality for B
- Bad locality for C
Locality/Caching Example: Matrix Mult

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];

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

a b c
Agenda

Locality and caching

Typical storage hierarchy

Virtual memory
Typical Storage Hierarchy

Smaller faster storage devices

- registers
  - CPU registers hold words retrieved from L1/L2/L3 cache

- L1/L2/L3 cache
  - L1/L2/L3 cache holds cache lines retrieved from main memory

- main memory (RAM)
  - Main memory holds disk blocks retrieved from local disks

Larger slower storage devices

- local secondary storage (local disks, SSDs)
  - Local disks hold files retrieved from disks on remote network servers

- remote secondary storage (distributed file systems, Web servers)
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: 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
Typical Storage Hierarchy

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
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 k and k+1 (do data transfer less often)
- Fast data transfer between levels k and k+1
  - => use small block sizes at k and k+1 (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>4 bytes</td>
</tr>
<tr>
<td>L1/L2/L3 cache line</td>
<td>32 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>
### Storage Hierarchy & Caching Issues

**Issue: Who manages the cache?**

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

Locality and caching
Typical storage hierarchy
Virtual memory
Main Memory: Illusion

Each process sees main memory as
Large: $2^{32} = 4$ GB of memory
Uniform: contiguous memory locations from 0 to $2^{32}-1$
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 & 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
Nobel 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>20 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>
</table>

On nobel with gcc217:
- Each offset is 12 bits
  - Each page consists of $2^{12} = 4K = 4096$ bytes
  - Each virtual page number consists of 20 bits
    - There are $2^{20} = 1M = 1,048,576$ virtual pages
  - Each virtual address consists of 32 bits
    - There are $2^{32} = 4G$ bytes of virtual memory (per process)
On nobel with gcc217:

- Each offset is 12 bits
- Each page consists of $2^{12} = 4K = 4096$ bytes
- Each physical page number consists of 21 bits
  - There are $2^{21} = 2M = 2,097,152$ physical pages
- Each physical address consists of 33 bits
  - There are $2^{33} = 8G$ bytes of physical memory (per CPU)
Page Tables

Question
  • How do OS and hardware implement virtual memory?

Answer (part 2)
  • Maintain a page table for each process
Page Tables (cont.)

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>
</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
Virtual Mem

<table>
<thead>
<tr>
<th>VP</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Physical Mem

<table>
<thead>
<tr>
<th>VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Process 1234 accesses mem at virtual addr 16386
16386 =
0000000000000000100000000000010 \_B =
Virtual page num = 4; offset = 2
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
  - \( = 00000000000000000001000000000010 \_B \)
  - \( = 4098 \)

- Hardware fetches/stores data from/to phys addr 4098
Virtual Memory Example 2

Process 1234 accesses mem at virtual addr 8200
8200 = 0000000000000000001000000010000000001000B = 
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
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.)

Process 1234 accesses mem at virtual addr 8200

\[
8200 = 00000000000000001000000001000_B
\]

Virtual page num = 2; offset = 8
Virtual Memory Example 2 (cont.)

<table>
<thead>
<tr>
<th>Virtual Memory</th>
<th>Page Table</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>VP 3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>VP 4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>VP 0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>VP 2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

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

Hardware forms physical addr
  Physical page num = 3; offset = 8
  = 0000000000000000011100000001000B
  = 12296

Hardware fetches/stores data from/to phys addr 12296
Virtual Memory Example 3

Process 1234
Virtual Mem

0
1
2
3
4
5
6
...

Process 1234
Page Table

<table>
<thead>
<tr>
<th>VP</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
</tr>
</tbody>
</table>

Physical Mem

0 | VP 3
1 | VP 4
2 | VP 0
3 | VP 2
...

Disk

X | VP 6
Y | VP 5

Process 1234 accesses mem at virtual addr 4105

4105 =
000000000000000000010000001001B =
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
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
- IA-32 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!!!
Aside: Segmentation

In the early days (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

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

Cons
- Immensely tedious and error-prone
- Compromises the portability of the code
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 page table for process X, restore contents of registers and 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
Additional Benefits of Virtual Memory

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 0x0bffffff 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
Additional Benefits of Virtual 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 nobel computers:

```
$ ps l
F   UID   PID  PPID PRI  NI  VSZ   RSS  WCHAN STAT TTY        TIME COMMAND
0 42579 13082 13081  20   0 112712 2016  wait Ss  pts/0     0:00  -bash
0 42579 13305 13082  20   0 156916 13684 signal T   pts/0     0:00  emacs -nw
0 42579 13517 13082  20   0 11272 892   -      R+  pts/0     0:00  ps l

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