Storage Management
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

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 |

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:** “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!
Cache Eviction Policies

**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)
Locality/Caching Example: Matrix Mult

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>row-major</th>
<th></th>
<th>col-major</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0][0]</td>
<td>18</td>
<td>a[0][0]</td>
<td>18</td>
</tr>
<tr>
<td>a[0][1]</td>
<td>19</td>
<td>a[1][0]</td>
<td>21</td>
</tr>
<tr>
<td>a[0][2]</td>
<td>20</td>
<td>a[2][0]</td>
<td>24</td>
</tr>
<tr>
<td>a[1][0]</td>
<td>21</td>
<td>a[0][1]</td>
<td>19</td>
</tr>
<tr>
<td>a[1][1]</td>
<td>22</td>
<td>a[1][1]</td>
<td>22</td>
</tr>
<tr>
<td>a[1][2]</td>
<td>23</td>
<td>a[2][1]</td>
<td>25</td>
</tr>
<tr>
<td>a[2][0]</td>
<td>24</td>
<td>a[0][2]</td>
<td>20</td>
</tr>
<tr>
<td>a[2][1]</td>
<td>25</td>
<td>a[1][2]</td>
<td>23</td>
</tr>
<tr>
<td>a[2][2]</td>
<td>26</td>
<td>a[2][2]</td>
<td>26</td>
</tr>
</tbody>
</table>

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

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++)
      \( c[i][j] += a[i][k] \times b[k][j]; \)
Agenda

Locality and caching

Typical storage hierarchy

Virtual memory
Typical Storage Hierarchy

- **Smaller faster storage devices**
  - registers
  - L1/L2/L3 cache
    - CPU registers hold words retrieved from 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
  - 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**: $\sim 10,000,000$ 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 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>
# 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</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>L1/L2/L3 cache</td>
<td><strong>Hardware</strong>, using simple algorithms</td>
</tr>
<tr>
<td>(cache of main memory)</td>
<td></td>
</tr>
<tr>
<td>Main memory</td>
<td><strong>Hardware and OS</strong>, using virtual memory concept with complex algorithms (since accessing disk is expensive)</td>
</tr>
<tr>
<td>(cache of local sec storage)</td>
<td></td>
</tr>
<tr>
<td>Local secondary storage</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
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
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 $2^{64}$ bytes of virtual memory (per process)
### CourseLab Virtual & Physical Addresses

<table>
<thead>
<tr>
<th>virtual addr</th>
<th>virtual page num</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>52 bits</td>
<td></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>25 bits</td>
<td></td>
<td>12 bits</td>
</tr>
</tbody>
</table>

**On CourseLab:**

- Each offset is 12 bits
  - Each page consists of $2^{12}$ bytes
- Each physical page number consists of 25 bits
  - There are $2^{25}$ physical pages
- Each physical address consists of 37 bits
  - There are $2^{37}$ (128G) bytes of physical memory (per computer)
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>
</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 16386
16386 = …000000000000000001000000000000010B =
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

\[
= 00000000000000000000000000000000010000000000010_B
\]

= 4098

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

Process 1234 accesses mem at virtual addr 8200
8200 = …000000000000000010000000001000\text{B} =
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.)

Process 1234

Virtual Mem

0
1
2
3
4
5
6

Physical Mem

0 VP 3
1 VP 4
2 VP 0
3 VP 2

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>

Disk

X VP 6
Y VP 5

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 = …0000000000000000100000001000_2
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 2 (cont.)

Hardware forms physical addr
Physical page num = 3; offset = 8
= 0000000000000000000011000000001000B
= 12296
Hardware fetches/stores data from/to phys addr 12296
Virtual Memory Example 3

Process 1234 accesses mem at virtual addr 4105
4105 = …000000000000000000000001000000001001_B =
Virtual page num = 1; offset = 9
Virtual Memory Example 3 (cont.)

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VP 3</td>
</tr>
<tr>
<td>1</td>
<td>VP 4</td>
</tr>
<tr>
<td>2</td>
<td>VP 0</td>
</tr>
<tr>
<td>3</td>
<td>VP 2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Disk

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>VP 6</td>
</tr>
<tr>
<td>Y</td>
<td>VP 5</td>
</tr>
</tbody>
</table>

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
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…
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!!!
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
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
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 CourseLab computers:

$ ps 1

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>UID</td>
<td>PID</td>
<td>PPID</td>
<td>PRI</td>
<td>NI</td>
<td>VSZ</td>
<td>RSS</td>
<td>WCHAN</td>
<td>STAT</td>
<td>TTY</td>
<td>TIME</td>
<td>COMMAND</td>
</tr>
<tr>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>--------</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>----------------</td>
</tr>
<tr>
<td>0</td>
<td>42579</td>
<td>9655</td>
<td>9696</td>
<td>30</td>
<td>10</td>
<td>167568</td>
<td>13840</td>
<td>signal</td>
<td>TN</td>
<td>pts/1</td>
<td>0:00</td>
<td>emacs -nw</td>
</tr>
<tr>
<td>0</td>
<td>42579</td>
<td>9696</td>
<td>9695</td>
<td>30</td>
<td>10</td>
<td>24028</td>
<td>2072</td>
<td>wait</td>
<td>SNs</td>
<td>pts/1</td>
<td>0:00</td>
<td>-bash</td>
</tr>
<tr>
<td>0</td>
<td>42579</td>
<td>9725</td>
<td>9696</td>
<td>30</td>
<td>10</td>
<td>11268</td>
<td>956</td>
<td>-</td>
<td>RN+</td>
<td>pts/1</td>
<td>0:00</td>
<td>ps 1</td>
</tr>
</tbody>
</table>

**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