Lecture 14/15: Memory Management (1)
Announcements

- Midterm key posted

- HW1 graded -- should be up soon

- Lab 2 was intended to be due today; ok to submit by Monday
  - hope you are winning the battle against xv6 scheduler
  - Lab 3 released, but you will have to read ahead
    » I may have to push back due date…we’ll see
Our plan of action

- Memory/storage technologies and trends
  - Memory wall!

- Locality of reference to the rescue
  - Caching in the memory hierarchy

- Abstraction: Address spaces and memory sharing
- Virtual memory

- Today: background and bird’s eye view – more details to follow later
Random-Access Memory (RAM)

- Key features
  - RAM is traditionally packaged as a chip.
  - Basic storage unit is normally a cell (one bit per cell).
  - Multiple RAM chips form a memory.

- Static RAM (SRAM)
  - Each cell stores a bit with a four or six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to electrical noise (EMI), radiation, etc.
  - Faster and more expensive than DRAM.

- Dynamic RAM (DRAM)
  - Each cell stores bit with a capacitor. One transistor is used for access.
  - Value must be refreshed every 10-100 ms.
  - More sensitive to disturbances (EMI, radiation,…) than SRAM.
  - Slower and cheaper than SRAM.
# SRAM vs DRAM Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>4 or 6</td>
<td>1X</td>
<td>No</td>
<td>Maybe</td>
<td>100x</td>
<td></td>
<td>Cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>Yes</td>
<td>Yes</td>
<td>1X</td>
<td></td>
<td>Main memories, frame buffers</td>
</tr>
</tbody>
</table>
Nonvolatile Memories

- DRAM and SRAM are volatile – lose info without power

- Nonvolatile memories (NVMs) retain value
  - Read-only memory (ROM): programmed during production
  - Programmable ROM (PROM): can be programmed once
  - Eraseable PROM (EPROM): can be bulk erased (UV, X-Ray)
  - Electrically eraseable PROM (EEPROM): electronic erase
  - Flash memory: EEPROMs with partial (sector) erase capability
    » Wears out after about 100,000 erasings.
  - Phase Change Memories (PCMs): also wear out
  - Many exciting NVMs at various stages of development
NVM Uses

- Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems, …)

- Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops, …)

- Caches in high end systems

- Getting better -- many expect Universal memory to come
  - i.e., large replace both DRAM and disk drives
The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.
Locality to the Rescue!

The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as locality.
Today

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy
- Virtual memory and memory sharing
Locality

- **Principle of Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- **Temporal locality**: 
  - Recently referenced items are likely to be referenced again in the near future.

- **Spatial locality**: 
  - Items with nearby addresses tend to be referenced close together in time.
Locality Example

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern).
  - Reference variable \( \text{sum} \) each iteration.

- **Instruction references**
  - Reference instructions in sequence.
  - Cycle through loop repeatedly.

```c
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```
Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Question:** Does this function have good locality with respect to array `a`?

```c
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```
Locality Example

- **Question:** Does this function have good locality with respect to array `a`?

```c
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;
    
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    
    return sum;
}
```
Locality Example

**Question:** Can you permute the loops so that the function scans the 3-d array \( a \) with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];

    return sum;
}
```
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a memory hierarchy.
Today

- Storage technologies and trends
- Locality of reference
- Caching in the memory hierarchy
- Virtual memory and memory sharing
An Example Memory Hierarchy

- **Registers**
  - Larger, slower, cheaper per byte
- **L1 cache (SRAM)**
  - Smaller, faster, costlier per byte
  - CPU registers hold words retrieved from L1 cache
- **L2 cache (SRAM)**
  - L1 cache holds cache lines retrieved from L2 cache
- **Main memory (DRAM)**
  - L2 cache holds cache lines retrieved from main memory
  - Main memory holds disk blocks retrieved from local disks
- **Local secondary storage (local disks)**
  - Local disks hold files retrieved from disks on remote network servers
- **Remote secondary storage (tapes, distributed file systems, Web servers)**

CSE 153 – Lecture 14 – Memory Management
Memory hierarchy

- **Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- Fundamental idea of a memory hierarchy:
  
  For each layer, faster, smaller device caches larger, slower device.

- Why do memory hierarchies work?
  
  Because of locality!
  
  » Hit fast memory much more frequently even though its smaller.
  
  Thus, the storage at level k+1 can be slower (but larger and cheaper!)

- **Big Idea**: The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.
General Cache Concepts

Smaller, faster, more expensive memory caches a subset of the blocks.

Data is copied in block-sized transfer units.

Larger, slower, cheaper memory viewed as partitioned into “blocks”.

Cache

```
4  9 10  3
```

Memory

```
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
```
General Cache Concepts: Hit

Request: 14

Data in block b is needed

Block b is in cache: Hit!
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)
General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.

- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
Summary so far

- The speed gap between CPU, memory and mass storage continues to widen.

- Well-written programs exhibit a property called locality.

- Memory hierarchies based on caching close the gap by exploiting locality.
Sharing Memory

- Rewind to the days of “second-generation” computers
  - Programs use physical addresses directly
  - OS loads job, runs it, unloads it

- Multiprogramming changes all of this
  - Want multiple processes in memory at once
    » Overlap I/O and CPU of multiple jobs
  - How to share physical memory across multiple processes?
    » Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
    » A program can run on machine with less memory than it “needs”
Virtual Addresses

- To make it easier to manage the memory of processes running in the system, we’re going to make them use **virtual addresses** (logical addresses)
  - Virtual addresses are independent of the actual physical location of the data referenced
  - OS determines location of data in physical memory
- Instructions executed by the CPU issue virtual addresses
  - Virtual addresses are translated by hardware into physical addresses (with help from OS)
  - The set of virtual addresses that can be used by a process comprises its **virtual address space**
Virtual Addresses

- Many ways to do this translation…
  - Need hardware support and OS management algorithms

- Requirements
  - Need protection – restrict which addresses jobs can use
  - Fast translation – lookups need to be fast
  - Fast change – updating memory hardware on context switch
Fixed Partitions

- Physical memory is broken up into fixed partitions
  - Size of each partition is the same and fixed
  - Hardware requirements: base register
  - Physical address = virtual address + base register
  - Base register loaded by OS when it switches to a process

Physical Memory

- P1
- P2
- P3
- P4
- P5
Fixed Partitions

How do we provide protection?
Fixed Partitions

- **Advantages**
  - *Easy to implement*
    - Need base register
    - Verify that offset is less than fixed partition size
  - *Fast context switch*

- **Problems?**
  - *Internal fragmentation*: memory in a partition not used by a process is not available to other processes
  - *Partition size*: one size does not fit all (very large processes?)
Variable Partitions

- Natural extension – physical memory is broken up into variable sized partitions
  - Hardware requirements: base register and limit register
  - Physical address = virtual address + base register

- Why do we need the limit register?
  - Protection: if (virtual address > limit) then fault
Variable Partitions

Virtual Address

Offset

Base Register

P3's Base

Limit Register

P3's Limit

Virtual Address + Offset

Yes?

No?

Protection Fault

P1

P2

P3
Variable Partitions

- Advantages
  - *No internal fragmentation*: allocate just enough for process

- Problems?
  - *External fragmentation*: job loading and unloading produces empty holes scattered throughout memory
Paging

- New Idea: split virtual address space into multiple partitions
  - Each can go anywhere!

Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory

But need to keep track of where things are!
Page Lookups

Virtual Address
- Page number
- Offset

Page Table
- Page frame

Physical Address
- Page frame
- Offset

Physical Memory
Paging Advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
    » All pages of the same size

- Simplifies protection
  - All chunks are the same size
  - Like fixed partitions, don’t need a limit register

- Simplifies virtual memory – later
Paging Limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of a page

- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - What can we do?

- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = $2^{20}$ PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - What can we do?
Segmentation

- Segmentation: partition memory into logically related units
  - Module, procedure, stack, data, file, etc.
  - Units of memory from user’s perspective

- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
  - Fixed partition : Paging :: Variable partition : Segmentation

- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table
  - Virtual addresses become <segment #, offset>
Segment Lookups

Virtual Address

Segment #  Offset

Segment Table

Yes?

Physical Memory

No?

Protection Fault

<

limit  base

+