Today

- Midterm topics end here.

- HW 2 is due Wednesday: great midterm review.

- Lab 3 is posted. Due next Wednesday (July 31)
  - Time to meet your inner computer security ninja...
  - More good practice with assembly, helpful to review for the exam.

- I have a research paper deadline on Wednesday.
  - I will be at my scheduled office hours, but otherwise, I will be very busy until Wednesday 2pm.
Midterm tips

■ Questions will be similar to homeworks and past exams.
  - See webpage for past exams
  - Try practice problems (solutions later in book)

■ Questions will focus on material covered in lectures and labs.
  - If it’s in the book/past exams, but we didn’t cover it in class, it’s not on the exam.

■ Great review: Finish HW 2 or start on Lab 3.

■ BYO questions: 2nd halves of Wednesday’s lecture and Thursday’s section.
  - Maybe later in the day Thursday if there’s interest?
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:

get_mpg:
  pushq %rbp
  movq %rsp, %rbp
  ...
  popq %rbp
  ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
11000001111110100001111

Computer system:

Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

Windows 8
Mac
How does execution time grow with SIZE?

```c
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
    for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
    }
}
```

Plot
Actual Data
An analogy...

- When you get a book from the library, where do you leave it until you finish reading it?
Making memory accesses fast!

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
- Program optimizations that consider caches
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus bandwidth evolved much slower

CPU Reg

Main Memory

Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100 cycles

Problem: lots of waiting on memory
Problem: Processor-Memory Bottleneck

Processor performance doubled about every 18 months

Bus bandwidth evolved much slower

Core 2 Duo:
Can process at least 256 Bytes/cycle

Core 2 Duo:
Bandwidth 2 Bytes/cycle
Latency 100 cycles

Solution: caches
Cache

- **English definition:** a hidden storage space for provisions, weapons, and/or treasures

- **CSE definition:** computer memory with short access time used for the storage of frequently or recently used instructions or data (i-cache and d-cache)

more generally,

used to optimize data transfers between system elements with different characteristics (network interface cache, I/O cache, etc.)
General Cache Mechanics

Cache

Memory

Smaller, faster, more expensive memory caches a subset of the blocks (a.k.a. lines)

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks” or “lines”
General Cache Concepts: **Hit**

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)
Why do caches work?
Why Caches Work

- **Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently
Why Caches Work

- **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.
  - Why is this important?
Why Caches Work

- **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?**
Why Caches Work

- **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.
  
- How do caches take advantage of this?
Example: Locality?

```c
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```
Example: Locality?

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

Data:
- Temporal: `sum` referenced in each iteration
- Spatial: array `a[ ]` accessed in stride-1 pattern
Example: Locality?

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

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern

- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence
Example: Locality?

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

- **Data:**
  - Temporal: `sum` referenced in each iteration
  - Spatial: array `a[]` accessed in stride-1 pattern

- **Instructions:**
  - Temporal: cycle through loop repeatedly
  - Spatial: reference instructions in sequence

- **Being able to assess the locality of code is a crucial skill for a programmer**
Locality Example #1

```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;
}
```

<table>
<thead>
<tr>
<th>a[0][0]</th>
<th>a[0][1]</th>
<th>a[0][2]</th>
<th>a[0][3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
</tr>
</tbody>
</table>
Locality Example #1

```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;
}
```

a[0][0]  a[0][1]  a[0][2]  a[0][3]
a[1][0]  a[1][1]  a[1][2]  a[1][3]

1: a[0][0]
2: a[0][1]
3: a[0][2]
4: a[0][3]
5: a[1][0]
6: a[1][1]
7: a[1][2]
8: a[1][3]
9: a[2][0]
10: a[2][1]
11: a[2][2]
12: a[2][3]

**stride-1**
Locality Example #2

```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 #2

```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;
}
```

stride-N
Local exemplary #3

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

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

    return sum;
}
```

- What is wrong with this code?
- How can it be fixed?
Cost of Cache Misses

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    Cache hit time of 1 cycle
    Miss penalty of 100 cycles

\( \text{cycle} = \text{single fixed-time machine step} \)
Cost of Cache Misses

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider:
    - Cache hit time of 1 cycle
    - Miss penalty of 100 cycles
  - Average access time:
    - 97% hits: 1 cycle + 0.03 * 100 cycles = 4 cycles
    - 99% hits: 1 cycle + 0.01 * 100 cycles = 2 cycles

- **This is why “miss rate” is used instead of “hit rate”**

  *cycle = single fixed-time machine step*
Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses)
    - \( = 1 - \text{hit rate} \)
  - Typical numbers (in percentages):
    - 3% - 10% for L1
    - Can be quite small (e.g., < 1%) for L2, depending on size, etc.

- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - Includes time to determine whether the line is in the cache
  - Typical hit times: 1 - 2 clock cycles for L1; 5 - 20 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
  - Typically 50 - 200 cycles for L2 (trend: increasing!)
Can we have more than one cache?

- Why would we want to do that?
Memory Hierarchies

- Some fundamental and enduring properties of hardware and software systems:
  - Faster storage technologies almost always cost more per byte and have lower capacity
  - The gaps between memory technology speeds are widening
    - True for: registers ↔ cache, cache ↔ DRAM, DRAM ↔ disk, etc.
  - Well-written programs tend to exhibit good locality

- These properties complement each other beautifully

- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**
An Example Memory Hierarchy

- CPU registers hold words retrieved from L1 cache
- L1 cache holds cache lines retrieved from L2 cache
- L2 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

- Smaller, faster, costlier per byte
- Larger, slower, cheaper per byte
Memory Hierarchies

■ **Fundamental idea of a memory hierarchy:**
  - For each \( k \), the faster, smaller device at level \( k \) serves as a cache for the larger, slower device at level \( k+1 \).

■ **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level \( k \) more often than they access the data at level \( k+1 \).
  - Thus, the storage at level \( k+1 \) can be slower, and thus larger and cheaper per bit.

■ **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.
Intel Core i7 Cache Hierarchy

Processor package

Core 0
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

Core 3
- Regs
- L1 d-cache
- L1 i-cache
- L2 unified cache

... (shared by all cores)

L1 i-cache and d-cache:
- 32 KB, 8-way,
- Access: 4 cycles

L2 unified cache:
- 256 KB, 8-way,
- Access: 11 cycles

L3 unified cache:
- 8 MB, 16-way,
- Access: 30-40 cycles

Block size: 64 bytes for all caches.