Today

- HW3 extension
  - Phew! 😊
- Lab 4?

- Finish up caches, exceptional control flow
Cache Associativity

1-way
8 sets,
1 block each

direct mapped

2-way
4 sets,
2 blocks each

4-way
2 sets,
4 blocks each

8-way
1 set,
8 blocks

fully associative
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block $i$ must be placed in slot $(i \mod \text{size})$), **direct-mapped**
    - if more than one, $n$-way **set-associative** (where $n$ is a power of 2)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time
Types of Cache Misses

- **Cold (compulsory) miss**
  - Occurs on first access to a block

- **Conflict miss**
  - Most hardware caches limit blocks to a small subset (sometimes just one) of the available cache slots
    - if one (e.g., block i must be placed in slot (i mod size)), **direct-mapped**
    - if more than one, n-way **set-associative** (where n is a power of 2)
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
    - e.g., referencing blocks 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - Occurs when the set of active cache blocks (the **working set**) is larger than the cache (just won’t fit)
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

L3 unified cache (shared by all cores)

Main memory

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.
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory
- What is the main problem with that?
What about writes?

- Multiple copies of data exist:
  - L1, L2, possibly L3, main memory

What to do on a write-hit?
- Write-through (write immediately to memory)
- Write-back (defer write to memory until line is evicted)
  - Need a *dirty bit* to indicate if line is different from memory or not

What to do on a write-miss?
- Write-allocate (load into cache, update line in cache)
  - Good if more writes to the location follow
- No-write-allocate (just write immediately to memory)

Typical caches:
- Write-back + Write-allocate, usually
- Write-through + No-write-allocate, occasionally
Where else is caching used?
Software Caches are More Flexible

■ Examples
  ▪ File system buffer caches, web browser caches, etc.

■ Some design differences
  ▪ Almost always fully-associative
    ▪ so, no placement restrictions
    ▪ index structures like hash tables are common (for placement)
  ▪ Often use complex replacement policies
    ▪ misses are very expensive when disk or network involved
    ▪ worth thousands of cycles to avoid them
  ▪ Not necessarily constrained to single “block” transfers
    ▪ may fetch or write-back in larger units, opportunistically
Optimizations for the Memory Hierarchy

- Write code that has locality
  - Spatial: access data contiguously
  - Temporal: make sure access to the same data is not too far apart in time

- How to achieve?
  - Proper choice of algorithm
  - Loop transformations
Example: Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k]*b[k*n + j];
}
Assume:
- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size C << n (much smaller than n)

First iteration:
- \( \frac{n}{8} + n = \frac{9n}{8} \) misses
  (omitting matrix \( c \))

- Afterwards in cache:
  (schematic)
Cache Miss Analysis

Assume:
- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size $C << n$ (much smaller than $n$)

Other iterations:
- Again:
  
  \[
  \frac{n}{8} + n = \frac{9n}{8} \text{ misses}
  \]
  (omitting matrix $c$)

Total misses:
- \[\frac{9n}{8} \times n^2 = \left(\frac{9}{8}\right) \times n^3\]
Blocked Matrix Multiplication

c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                    for (i1 = i; i1 < i+B; i1++)
                        for (j1 = j; j1 < j+B; j1++)
                            for (k1 = k; k1 < k+B; k1++)
                                c[i1*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}
Cache Miss Analysis

- **Assume:**
  - Cache block = 64 bytes = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks fit into cache: $3B^2 < C$

- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix $c$)

- Afterwards in cache (schematic)
Cache Miss Analysis

Assume:
- Cache block = 64 bytes = 8 doubles
- Cache size $C << n$ (much smaller than $n$)
- Three blocks fit into cache: $3B^2 < C$

Other (block) iterations:
- Same as first iteration
- $2n/B \times B^2/8 = nB/4$

Total misses:
- $nB/4 \times (n/B)^2 = n^3/(4B)$
Summary

- No blocking: \((9/8) * n^3\)
- Blocking: \(1/(4B) * n^3\)

- If \(B = 8\) difference is \(4 * 8 * 9 / 8 = 36x\)
- If \(B = 16\) difference is \(4 * 16 * 9 / 8 = 72x\)

- Suggests largest possible block size \(B\), but limit \(3B^2 < C\)!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: \(3n^2\), computation \(2n^3\)
    - Every array element used \(O(n)\) times!
  - But program has to be written properly
Cache-Friendly Code

- **Programmer can optimize for cache performance**
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique

- **All systems favor “cache-friendly code”**
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code
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

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.

Main memory
The Memory Mountain

![Graph showing read throughput (MB/s) against stride (x8 bytes) and working set size (bytes).](image)

- **Intel Core i7**
  - 32 KB L1 i-cache
  - 32 KB L1 d-cache
  - 256 KB unified L2 cache
  - 8M unified L3 cache
- All caches on-chip
**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
1000110100000100
1000100111000010
1100000111111010
1000000111111010
```

### OS:
```
Windows 8
```

## Topics Covered:
- Data & addressing
- Integers & floats
- Machine code & C
- x86 assembly programming
- Procedures & stacks
- Arrays & structs
- Memory & caches
- Exceptions & processes
- Virtual memory
- Memory allocation
- Java vs. C
Control Flow

- So far, we’ve seen how the flow of control changes as a single program executes.
- A CPU executes more than one program at a time though – we also need to understand how control flows across the many components of the system.

- **Exceptional control flow** is the basic mechanism used for:
  - Transferring control between processes and OS
  - Handling I/O and virtual memory within the OS
  - Implementing multi-process applications like shells and web servers
  - Implementing concurrency
Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s control flow (or flow of control)

**Physical control flow**

```
<startup>
inst_1
inst_2
inst_3
...
inst_n
<shutdown>
```
Altering the Control Flow

- Up to now: two ways to change control flow: ... which ones?
Altering the Control Flow

- Up to now: two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return

  Both react to changes in *program state*

- Processor also needs to react to changes in *system state*
  - Like?
Altering the Control Flow

- Up to now: two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  Both react to changes in program state

- Processor also needs to react to changes in system state
  - user hits “Ctrl-C” at the keyboard
  - user clicks on a different application’s window on the screen
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - system timer expires

- Can jumps and procedure calls achieve this?
Altering the Control Flow

- Up to now: two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  Both react to changes in program state

- Processor also needs to react to changes in system state
  - user hits “Ctrl-C” at the keyboard
  - user clicks on a different application’s window on the screen
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - system timer expires

- Can jumps and procedure calls achieve this?
  - Jumps and calls are not sufficient – the system needs mechanisms for “exceptional” control flow!
Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - Exceptions
    - change processor’s in control flow in response to a system event (i.e., change in system state, user-generated interrupt)
  - Combination of hardware and OS software
- Higher level mechanisms
  - Process context switch
  - Signals – you’ll hear about these in CSE451 and CSE466
  - Implemented by either:
    - OS software
    - C language runtime library
An *exception* is transfer of control to the operating system (OS) in response to some *event* (i.e., change in processor state)

- Examples:
  - div by 0, page fault, I/O request completes, Ctrl-C

*How does the system know where to jump to in the OS?*
Interrupt Vectors

- Each type of event has a unique exception number $k$
- $k = \text{index into exception table (a.k.a. interrupt vector)}$
- Handler $k$ is called each time exception $k$ occurs
Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor’s interrupt pin(s)
  - Handler returns to “next” instruction

- **Examples:**
  - I/O interrupts
    - hitting Ctrl-C on the keyboard
    - clicking a mouse button or tapping a touchscreen
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button on front panel
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - Intentional: transfer control to OS to perform some function
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - **Faults**
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), segment protection faults (unrecoverable), integer divide-by-zero exceptions (unrecoverable)
    - Either re-executes faulting (“current”) instruction or aborts
  - **Aborts**
    - Unintentional and unrecoverable
    - Examples: parity error, machine check
    - Aborts current program
Trap Example: Opening File

- User calls: `open(filename, options)`
- Function `open` executes system call instruction `int`
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```c
int a[1000];
main ()
{
    a[500] = 13;
}
```

80483b7:  c7 05 10 9d 04 08 0d  movl $0xd,0x8049d10

User Process

```
User Process

movl

exception: page fault

returns

OS

Create page and load into memory

Page handler must load page into physical memory

Returns to faulting instruction: **mov** is executed again!

Successful on second try
Fault Example: Invalid Memory Reference

```c
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:    c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```

- Page handler detects invalid address
- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”
## Exception Table IA32 (Excerpt)

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-127</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
<tr>
<td>128 (0x80)</td>
<td>System call</td>
<td>Trap</td>
</tr>
<tr>
<td>129-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>

Summary

- Exceptions
  - Events that require non-standard control flow
  - Generated externally (interrupts) or internally (traps and faults)
  - After an exception is handled, one of three things may happen:
    - Re-execute the current instruction
    - Resume execution with the next instruction
    - Abort the process that caused the exception