Cache Example, System Control Flow
CSE 351 Winter 2017

happy friday!

http://xkcd.com/292/

Administrivia

- Lab 3 due Monday
- Lab 4 released Monday
- HW 3 released
- Phew! 😊

- Remember to do readings and practice problems on the book
Core i7: Associativity

Processor package

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regs</td>
<td>L1 i-cache</td>
<td>L1 i-cache</td>
<td>L1 i-cache</td>
</tr>
<tr>
<td>L1 d-cache</td>
<td>L2 unified cache</td>
<td>L2 unified cache</td>
<td>L2 unified cache</td>
</tr>
<tr>
<td>L2 unified cache</td>
<td>L3 unified cache (shared by all cores)</td>
<td>L3 unified cache (shared by all cores)</td>
<td>L3 unified cache (shared by all cores)</td>
</tr>
</tbody>
</table>

Main memory

Block/line size:
- 64 bytes for all

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

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

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

slower, but more likely to hit

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 and all caches in-between
  - Write-back: defer write to memory until line is evicted (replaced)
    - Must track which cache lines have been modified ("dirty bit")
- What to do on a write-miss?
  - Write-allocate: ("fetch on write") load into cache, update line in cache
    - Good if more writes or reads to the location follow, example?
  - No-write-allocate: ("write around") just write immediately to memory
- Typical caches:
  - Write-back + Write-allocate, usually
  - Write-through + No-write-allocate, occasionally
Write-back, write-allocate example

Contents of memory stored at address G

In this example we are sort of ignoring block offsets. Here a block holds 2 bytes (16 bits, 4 hex digits).

Normally a block would be much bigger and thus there would be multiple items per block. While only one item in that block would be written at a time, the entire line would be brought into cache.

Write-back, write-allocate example

mov 0xFACE, F
Write-back, write-allocate example

\[ \text{mov 0xFACE, F} \]

\begin{itemize}
  \item \text{Cache:}
  \begin{itemize}
    \item \text{F: 0xFACE}
  \end{itemize}
  \item \text{Memory:}
  \begin{itemize}
    \item \text{F: 0xCAF\text{E}}
    \item \text{G: 0xBE\text{E}}
  \end{itemize}
\end{itemize}

\text{dirty bit}

\text{Step 1: Bring F into cache}

Write-back, write-allocate example

\[ \text{mov 0xFACE, F} \]

\begin{itemize}
  \item \text{Cache:}
  \begin{itemize}
    \item \text{F: 0xFACE}
  \end{itemize}
  \item \text{Memory:}
  \begin{itemize}
    \item \text{F: 0xCAF\text{E}}
    \item \text{G: 0xBE\text{E}}
  \end{itemize}
\end{itemize}

\text{dirty bit}

\text{Step 2: Write 0xFACE to cache only \textbf{and set dirty bit}}
Write-back, write-allocate example

```
mov 0xFACE, F    mov 0xFEED, F
```

Write hit! Write 0xFEED to cache only

Write-back, write-allocate example

```
mov 0xFACE, F    mov 0xFEED, F    mov G, %rax
```

Write hit! Write 0xFEED to cache only
Write-back, write-allocate example

```
mov 0xFACE, F  mov 0xFEED, F  mov G, %rax
```

1. Write F back to memory since it is dirty
2. Bring G into the cache so we can copy it into %rax

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 can you achieve locality?
  - Adjust memory accesses in code (software) to improve miss rate (MR)
    - Requires knowledge of both how caches work as well as your system's parameters
  - Proper choice of algorithm
  - Loop transformations
Example: Matrix Multiplication

\[
C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}
\]

Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row major matrix in memory:
    - COLUMN of matrix (blue) is spread among cache blocks shown in red.
Naïve Matrix Multiply

# move along rows of A
for (i = 0; i < n; i++)
  # move along columns of B
  for (j = 0; j < n; j++)
    # EACH k loop reads row of A, col of B
    # Also read & write c(i, j) n times
    for (k = 0; k < n; k++)
      c[i*n+j] += a[i*n+k] * b[k*n+j];

Cache Miss Analysis (Naïve)

- Scenario Parameters:
  - Square matrix \((n \times n)\), elements are doubles
  - Cache block size \(K = 64\) B = 8 doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

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

- Afterwards in cache:
  - (schematic)
Cache Miss Analysis (Naïve)

- Scenario Parameters:
  - Square matrix \((n \times n)\), elements are doubles
  - Cache block size \(K = 64\) B = 8 doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

- Other iterations:
  - Again:
    \[
    \frac{n}{8} + n = \frac{9n}{8}\] misses

- Total misses:
  \[
  \frac{9n}{8} \times n^2 = \frac{9n^3}{8}\]
  once per element

Linear Algebra to the Rescue (1)

- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix “blocks”)

- For example, multiply two 4x4 matrices:

\[
A = \begin{bmatrix}
    a_{11} & a_{12} & a_{13} & a_{14} \\
    a_{21} & a_{22} & a_{23} & a_{24} \\
    a_{31} & a_{32} & a_{33} & a_{34} \\
    a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
= \begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix}
\]

with \(B\) defined similarly.

\[
AB = \begin{bmatrix}
    (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\
    (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22})
\end{bmatrix}
\]
Linear Algebra to the Rescue (2)

Matrices of size \( n \times n \), split into 4 blocks of size \( r \) (\( n=4r \))

\[
C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k}B_{k2}
\]

- Multiplication operates on small "block" matrices
  - Choose size so that they fit in the cache!
  - This technique called "cache blocking"

Blocked Matrix Multiply

- Blocked version of the naïve algorithm:

```plaintext
# move by rxr BLOCKS now
for (i = 0; i < n; i += r)
  for (j = 0; j < n; j += r)
    for (k = 0; k < n; k += r)
      # block matrix multiplication
        for (ib = i; ib < i+r; ib++)
          for (jb = j; jb < j+r; jb++)
            for (kb = k; kb < k+r; kb++)
              c[ib*n+jb] += a[ib*n+kb]*b[kb*n+jb];
```

- \( r = \) block matrix size (assume \( r \) divides \( n \) evenly)
Cache Miss Analysis (Blocked)

- Scenario Parameters:
  - Cache block size $K = 64$ B = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $r \times r$ fit into cache: $3r^2 < C$

- First (block) iteration:
  - $r^2 / 8$ misses per block
  - $2n/r \times r^2 / 8 = nr / 4$

- Afterwards in cache (schematic)

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

- Total misses:
  - $nr / 4 \times (n/r)^2 = n^3 / (4r)$
Matrix Multiply Summary

- Naïve: \( \frac{9}{8} \times n^3 \)
- Blocked: \( \frac{1}{4r} \times n^3 \)
  - If \( r = 8 \), difference is \( 4 \times 8 \times \frac{9}{8} = 36x \)
  - If \( r = 16 \), difference is \( 4 \times 16 \times \frac{9}{8} = 72x \)

- Blocking optimization only works if the blocks fit in the cache
  - Suggests largest possible block size up to limit \( 3r^2 \leq C \)

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

Matrix Multiply Visualization

- Here \( n = 100 \), \( C = 32 \text{ KiB} \), \( r = 30 \)
- Naïve:
  - \( \approx 1,020,000 \) cache misses
- Blocked:
  - \( \approx 90,000 \) cache misses
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 size, cache block size, associativity, 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

The Memory Mountain

- Core i7 Haswell
  - 2.1 GHz
  - 32 KB L1 d-cache
  - 256 KB L2 cache
  - 8 MB L3 cache
  - 64 B block size

Slopes of spatial locality

Ridges of temporal locality

Aggressive prefetching
Learning About Your Machine

- **Linux:**
  - `lscpu`
  - `ls /sys/devices/system/cpu/cpu0/cache/index0/`
    - `Ex: cat /sys/devices/system/cpu/cpu0/cache/index*/size`
  - `cat /proc/cpuinfo | grep cache | sort | uniq`

- **Windows:**
  - `wmic memcache get <query>` *(all values in KB)*
  - `Ex: wmic memcache get MaxCacheSize`

- Modern processor specs: [http://www.7-cpu.com/](http://www.7-cpu.com/)

Roadmap

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

```java
Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:
```
get_mpg:
pushq %ebp
movq %esp, %ebp
...
popq %ebp
ret
```

Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

OS:
Windows
Mac

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

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)

Control Flow

- **So far:** we’ve seen how the flow of control changes as a single program executes
- **Reality:** multiple programs running *concurrently*
  - How does control flow across the many components of the system?
  - In particular: More programs running than CPUs
- **Exceptional control flow** is basic mechanism used for:
  - Transferring control between *processes* and OS
  - Handling I/O and *virtual memory* within the OS
  - Implementing multi-process apps 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)

```
<startup>
instr_1
instr_2
instr_3
...
instr_n
<shutdown>
```

Physical control flow

time

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
  - Unix/Linux 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?
  - No – the system needs mechanisms for “exceptional” control flow!
Java Digression #1

- Java has exceptions, but they’re *something different*
  - **Examples:** `NullPointerException`, `MyBadThingHappenedException`, ...
  - `throw` statements
  - `try/catch` statements ("throw to youngest matching catch on the call-stack, or exit-with-stack-trace if none")
- Java exceptions are for reacting to (unexpected) program state
  - Can be implemented with stack operations and conditional jumps
  - A mechanism for “many call-stack returns at once”
  - Requires additions to the calling convention, but we already have the CPU features we need
- System-state changes on previous slide are mostly of a different sort (asynchronous/external except for divide-by-zero) and implemented very differently

Exceptional Control Flow

- Exists at all levels of a computer system
- **Low level mechanisms**
  - **Exceptions**
    - Change in processor’s control flow in response to a system event (i.e., change in system state, user-generated interrupt, bugs)
    - Implemented using a combination of hardware and OS software
  - **Higher level mechanisms**
    - **Process context switch**
      - Implemented by OS software and hardware timer
    - **Signals**
      - Implemented by OS software
      - We won’t cover these – see CSE451 and CSE/EE474
Exceptions

- An exception is transfer of control to the operating system (OS) kernel in response to some event (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C

How does the system know where to jump to in the OS?

Exception Table

- A jump table for exceptions (also called Interrupt Vector Table)
  - 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

Exceptions

- Exception Table
  - Exception numbers
  - Exception numbers like jump table in switch statements in x86-64
  - Code for exception handler 0
  - Code for exception handler 1
  - Code for exception handler 2
  - Code for exception handler $n-1$
Exception Table (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>32-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>

Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the 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
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs

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, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - Aborts
    - Unintentional and unrecoverable
    - Examples: parity error, machine check (hardware failure detected)
    - Aborts current program
System Calls

- Each system call has a unique ID number
- Examples for Linux on x86-64:

```
<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
<td>Read file</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>Write file</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
<td>Open file</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>Close file</td>
</tr>
<tr>
<td>4</td>
<td>stat</td>
<td>Get info about file</td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
<td>Create process</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
<td>Execute a program</td>
</tr>
<tr>
<td>60</td>
<td>_exit</td>
<td>Terminate process</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
<td>Send signal to process</td>
</tr>
</tbody>
</table>
```

Traps Example: Opening File

- User calls `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00   mov    $0x2,%eax  # open is syscall 2
e5d7e:   0f 05           syscall  # return value in %rax
e5d80:   48 3d 01 f0 ff ff cmp    $0xffffffffffff001,%rax
...
e5dfa:   c3             retq
```

- `%rax` contains syscall number
- Other arguments in `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
Fault Example: Page Fault

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

\[
\text{int } a[1000]; \\
\text{int } main () \\
\{ \\
\text{a[500]} = 13; \\
\}
\]

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

- Page fault handler must load page into physical memory
- Returns to faulting instruction: mov is executed again!
  - Successful on second try

Fault Example: Invalid Memory Reference

\[
\text{int } a[1000]; \\
\text{int } main () \\
\{ \\
\text{a[5000]} = 13; \\
\}
\]

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

- Page fault handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with “segmentation fault”
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