Up to Speed Yet?

- Buflab
  - Due tonight, 11:59 PM EDT

- Cachelab
  - Out tonight, 11:59 PM EDT
  - Due Tuesday, June 25, 2013, 11:59 PM
  - This will be the last one week lab
    - But the labs don’t get any easier
THIS AND THAT AND WHAT’S TODAY

- Exam Talk
- Alignment
- Memory Organization
- Caching
  - Buzzword: locality
  - Cache organization
- Cachelab
  - Part A – Implement a (hardware) cache simulator
  - Part B – Efficient matrix transpose
  - “Bro, do you even C?” – helpful C stuff
Motivation: Why Bother with the ECES?
**STRUCTS, WHAT ARE THEY?**

- An object with sets of (related) values that can be passed around together
- Values not necessarily contiguous in memory
  - Each object may have a different alignment rule
  - There is a constant offset from the beginning of the struct
ALIGNMENT OF STRUCTS

- Entire struct aligns according to the largest alignment constraint of its member
  - Must be multiple of K (largest alignment requirement)
  - Compilers enforce this; different alignments depending

- Overall structure length a multiple of K
  - Optimize length by declaring largest elements first
**Example of a Struct (from Lecture)**

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```
WHAT ARE UNIONS?

- A place in memory used to store data types
- Unlike structs, union elements are not placed “next to each other in memory”
  - Rather they are placed “on top”
- Size is decided by the largest element
- Only one field used at a time
  - Each write to an element overwrites some part of another
- This class does not deal with unions very much
**Union Example (from Lecture)**

```c
union u1 {
    char c;
    int i[2];
    double v;
} *up;
```
**Structs on Exams**

struct stats {
    int num_views;
    short sum;
};

Goal: Align struct system_f according to a 64-bit Linux system

struct system_f {
    char a;
    int* b;
    int c[3];
    long d;
    struct stats e;
    short f;
};

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>X</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Memory Hierarchy (From Lecture)

- **L0:** Registers
  - CPU registers hold words retrieved from L1 cache
- **L1:** L1 cache (SRAM)
  - L1 cache holds cache lines retrieved from L2 cache
- **L2:** L2 cache (SRAM)
  - L2 cache holds cache lines retrieved from main memory
- **L3:** Main memory (DRAM)
  - Main memory holds disk blocks retrieved from local disks
- **L4:** Local secondary storage (local disks)
  - Local disks hold files retrieved from disks on remote network servers
- **L5:** Remote secondary storage (tapes, distributed file systems, Web servers)

**Smaller, faster, costlier per byte**

**Larger, slower, cheaper per byte**
SRAM vs DRAM

- **SRAM**
  - Faster (L1 Cache: 1 CPU cycle)
  - Smaller (L1 in kilobytes; L2 in megabytes)
  - More expensive and “energy-hungry”

- **DRAM (Main memory)**
  - Relatively slower (hundreds of CPU cycles)
  - Larger (Gigabytes)
  - Cheaper
HARDWARE INSIGHT

- Picture from 18-447 slides
Locality

- Temporal locality
  - Recently referenced items are likely to be referenced again in the near future.
  - After accessing address X in memory, save the bytes in cache for future access.

- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time.
  - After accessing address X, save the block of memory around X in cache for future access.
GENERAL CACHING (FROM LECTURE)

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 “blocks”.

Cache

Memory
ADDRESS DIVISION IN CACHES

- On the Shark machines, addresses are 64-bits
- Dividing a memory address
  - Block offset: b bits
  - Set index: s bits
  - Tag bits: address size – b – s
Cache Parameters

- A cache is a set of $S = 2^s$ cache sets
- A cache set is a set of $E$ cache lines
  - $E$ is called associativity
  - If $E = 1$, the cache is “direct-mapped”
- Each cache line stores a block
  - Each block has $B = 2^b$ bytes
- Total capacity $C = S \times B \times E$
VISUAL CACHE TERMINOLOGY

E lines per set

$S = 2^s$ sets

Address of word:

- **t bits**
- **s bits**
- **b bits**

- **tag**
- **set index**
- **block offset**

data begins at this offset

valid bit

$B = 2^b$ bytes per cache block (the data)
CACHE LOOKUP STEPS

- Divide address into parts
  - Block offset: Low b bits
  - Set number: Next s bits
  - Tag: Remaining ((address size) – b – s) bits

- Check each line in a set, compare tags
  - If one matches and it’s valid, it’s a hit!
  - If none match, it’s a miss. Add block to cache
    - If there’s no room, evict a line from the set
CACHE EVICTION

Observations
- Each address block has a specified set it belongs to
- Each block has a specific tag for that set
- If we need to add items to a set and it’s full, we have to evict via an eviction policy

Least-recently used (LRU)
- Main eviction policy for 15-213
- Evict (remove) the least recently used block from the cache to make room for the next block
CACHE LAB PART A

- **Cache Simulator**
  - Implement for variable s, b, and E values
    - Values read in from a trace file (at runtime)
  - Least Recently Used (LRU) Policy

- **Cache Simulator != Cache**
  - This simulator does NOT store memory contents
    - Only performs lookups/ evictions for various addresses
  - We do NOT care about block offsets here
  - Your goal: count the number of hits, misses, and evictions
    - Read addresses from files and return these numbers
A cache is just 2D array of cache lines:
- struct cache_line cache[S][E];
- $S = 2^s$ is the number of sets
- $E$ is associativity

Each cache_line has:
- Valid bit
- Tag
- LRU “counter”
ANITA’S FAVORITE DATA STRUCTURE

- Linked lists
  - “The only data structure you will ever need”
  - (Heavily) used in cache and malloc lab
  - A lesson on linked list in the credits page
How necessary is the LRU counter?
- We have the power to insert nodes wherever we want
  - So why use a counter?

As a C programmer, implementing a linked list should be second nature
- The same deal every time
  - Pointers to each node
  - Traversal helper functions
  - Checking invariants
**CACHELAB PART B**

- Efficient matrix transpose
  - Goal: Increasing locality via blocking
  - Involves careful analysis of cache element placement

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>9</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
CACHELAB PART B

Cache:
- 1 kilobytes of cache
- Directly mapped (E=1)
- Block size is 32 bytes (b=5)
- S = 32 sets (s=5)

Test Matrices:
- 32 x 32, 64 x 64, 61 x 67
- You only need to optimize for these sizes
“Bro, Do You Even C?”

In this section:

- Warnings are errors
- Headers
- Useful C functions
WARNINGS ARE ERRORS

- Strict compilation flags
  - Avoid potential errors that are hard to debug
  - Learn good habits from the beginning

- Add “-Werror” to your compilation flags

- DO NOT ignore the compiler errors
WHAT ABOUT HEADERS?

- Remember to include files that we will be using functions from

- If function declaration is missing
  - Find corresponding header file
  - `unix> man function-name`
    - Skim the man pages, they’ll tell you what you need to know
**FUNCTION 1: getopt**

- *getopt* automates parsing elements on the unix command line
  - Typically called in a loop to retrieve arguments
  - Use a switch statement to handle options
  - Returns -1 when there are no more arguments

- **Must include the header file** `unistd.h`
FUNCTION 1: getopt USAGE

- Switch statement used on the (local) variable holding the return value from getopt
  - Each command line input can be handled separately
  - optarg – Points to the value of the option argument
    - This is set by the getopt function

- Food for thought
  - How do we handle invalid inputs?
FUNCTION 1: GETOPT EXAMPLE

Suppose we had an executable called “foo”
- Example call from shell: `unix> ./foo -x 1`

Next slide: Parsing the argument to the x option
- Notice: We passed in an int which is read as a char *
- We use `atoi` to convert the string to an int
FUNCTION 1: getopt EXAMPLE CONT.

int main(int argc, char** argv){
    int opt, x;

    /* looping over arguments */
    while(-1 != (opt = getopt(argc, argv, "x:"))){
        /* determine which argument it's processing */
        switch(opt) {
            case 'x':
                x = atoi(optarg);
                break;
            default:
                printf("wrong argument\n");
                break;
        }
    }
}
**FUNCTION 2: fscanf**

- The `fscanf` function is just like `scanf/sscanf`
  - But it can specify a stream to read from
  - `scanf` always reads from stdin
  - `sscanf` reads from a string

- **Parameters:**
  - File pointer
  - Format string with information on how to read file
  - Variable number of pointers to with locations for storing data from file

- Typically use in a loop until it hits the end of file
- `fscanf` is useful in reading from the trace files
**Function 2: fscanf Example**

FILE *pFile; // pointer to FILE object

/* open file for reading */
pFile = fopen("myfile.txt", "r");

int x, y;
char c;

/* read two ints and a char from file */
while(fscanf(pFile, "%d %d %c", &x, &y, &c) > 0){
    // Do stuff
}

fclose(pFile); // remember to close file when done
FUNCTION 3 AND 4: MALLOC/FREE

- Use `malloc` to allocate memory on the heap
  - Returns a pointer to location in memory

- Always free what you `malloc`
  - Or you’ll suffer from memory leaks

- Example usage:
  - `int *pointer = malloc( sizeof(int) );`
  - `free(pointer);`

- **DO NOT** free memory you didn’t allocate
  - This includes double free-ing
STYLE AND TIPS FOR LIFE

- Check for failures and errors ALWAYS
  - Functions don’t always succeed
  - What happens when a system call fails?

- Common cases of failure:
  - Not checking the return of `malloc`
  - Not handling invalid inputs
  - Generally, not checking returns of functions
I Stole From These Places

- Upside down CPU Cache Pyramid
- Memory Bank Organization from 18-447
- Wikipedia: Linked Lists
- C Linked List Example
- getopt from GNU
- fscanf from CPlusPlus.com