Up to Speed Yet?

- Buflab
  - Due tomorrow, midnight

- Cachelab
  - Out tomorrow, midnight
  - Due Thursday, February 28, 2013, midnight
    - Labs will be going back to regular Thursday due date
  - 10 days to get your C skills back up!
THIS AND THAT AND WHAT

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
BEFORE WE BEGIN...

- Encouraging Female Reverse Engineers
  - Contest to reverse engineer malicious software
  - And document it with utmost understanding

- Prize
  - Ticket to Symposium on Security for Asia Network (SysScan)

- Details
  - [http://addxorrol.blogspot.de/2013/01/encouraging-female-reverse-engineers.html](http://addxorrol.blogspot.de/2013/01/encouraging-female-reverse-engineers.html)
  - Only women can submit (sorry guys)
  - One of the judges went to CMU!
  - Deadline: 24th of March 2013, 23:59 GMT+1
BAM! CIRCULAR STACK!

SPARC (scalable processor architecture) Architecture
SUPER BRIEF ON ALIGNMENT

- **Structs**
  - Align according to the largest alignment requirement
    - Must be multiple of K (largest alignment requirement)
    - System dependent alignments requirements
    - Compilers enforce this
  - Overall structure length a multiple of K
    - Optimize length by declaring largest elements first

- **Unions**
  - Size allocated according to largest element
  - Only one field used at a time
QUICK EXAMPLE FROM LECTURE

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;

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

![Diagram illustrating the layout of union U1 and struct S1 with memory offsets and data types.]
MEMORY HIERARCHY (FROM LECTURE)

- **Registers**: Smaller, faster, costlier per byte
- **L1 cache (SRAM)**: L1 cache holds cache lines retrieved from L2 cache
- **L2 cache (SRAM)**: L2 cache holds cache lines retrieved from main memory
- **Main memory (DRAM)**: 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)**: 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
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)

Cache

| 4 | 9 | 10 | 3 |

Data is copied in block-sized transfer units

Memory

| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

- Larger, slower, cheaper memory viewed as partitioned into “blocks”
- Smaller, faster, more expensive memory caches a subset of the blocks
ADDRESS DIVISION IN CACHES

- On the Sharks, addresses are 64-bits
- Dividing a memory address
  - Block offset: $b$ bits
  - Set index: $s$ bits
CACHES

- A cache is a set of $2^s$ cache sets

- A cache set is a set of $E$ cache lines
  - $E$ is called associativity
  - If $E=1$, it is called “direct-mapped”

- Each cache line stores a block
  - Each block has $2^b$ bytes
**Visual Cache Terminology**

- $E = 2^e$ lines per set
- $S = 2^s$ sets
- $B = 2^b$ bytes per cache block (the data)

Address of word:
- $t$ bits (tag)
- $s$ bits (set index)
- $b$ bits (block offset)

Data begins at this offset
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
  - We do NOT care about block offsets here
  - Your goal: implement the policy and count the number of hits, misses, and evictions
GENERAL SIMULATOR DESIGN HINTS

- 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
FOOD FOR THOUGHT/ OTHER DESIGNS

- 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
  - 5-10 minutes tops
  - The same deal every time
    - Pointers to each node
    - Traversal helper functions
    - Making the right checks
CACHELAB PART B

- Efficient matrix transpose
  - Goal: Increasing locality via blocking

```
1  2  3  4
5  6  7  8
9 10 11 12
13 14 15 16
   1  5  9 13
   2  6 10 14
   3  7 11 15
   4  8 12 16
```
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 files
  - `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

- 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

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;
        }
    }
}

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

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` will be useful in reading from the trace files
**FUNCTION 2: fscanf EXAMPLE**

```c
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

- Read the style guideline
  - “But I already read it!”
  - Good, read it again.

- 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

- Understanding the SPARC Architecture
- Wikipedia: Linked Lists
- C Linked List Example
- getopt from GNU
- fscanf from CPlusPlus.com