Labs

- Buffer Lab due tonight
- Cache Lab out now!
The Memory Hierarchy

- Faster memory is much more expensive, larger per bit
- Most programs exhibit **locality**
  - Spatial locality: Most likely to access locations near one another – e.g. in an array
  - Temporal locality: Most likely to access locations that have recently accessed (or near one that has been recently accessed)
- Caching recently used memory (and memory near recently used memory) in faster storage offers massive performance improvements
The Memory Hierarchy

- Registers
- SRAM – CPU cache
- DRAM – main memory
- Disk

Values used more frequently stay in faster memory
- Register Allocation (compile-time)
- Caching (runtime, hardware-level)
- Virtual memory (runtime, hardware / OS-level)
Caches – Organization

- Generalized system with several parameters describing size, associativity, block size
  - $m$: number of bits in an address
  - $M = 2^m$: Number of addresses in memory
  - $S = 2^s$: Number of sets (number of bits in a set number)
  - $B = 2^b$: Number of bytes per block (region of memory stored as a unit)
  - $E$: Number of lines each set can hold (associativity)
- Total capacity $C = S \times B \times E$
Caches – Lookup

• Divide address into parts
• Block offset: Low $b$ bits
• Set number: Next $s$ bits
• Tag: Remaining $m - b - s$ bits
• Check each line in set, compare tags
• If one matches and it’s valid, hit!
• If none match, miss. Add block to cache
  • If there’s no room, evict a line from the set
  • LRU – evict the least recently used line to make room for the new one
Cache Lab

• Out now!
• Two parts
• Write a cache **simulator** – not a real cache, but performs lookups / evictions
  • Read trace files
    • list of loads / stores / modifications at addresses
  • Return the number of cache hits / misses / evictions
• Then, write a matrix transposition function optimized for cache performance
Cache Lab – C Programming

- More code than previous labs
- Use good style, document your code!
- Not much base code; it’s up to you to design the structure of your simulator
- Use library functions for parsing trace files / managing memory
Useful Library Functions: getopt()

- `#include <unistd.h>`
- Parses command line arguments
- Call multiple times to parse one argument at a time
- “man -S 3 getopt”
  - there’s a command line program of the same name, and man defaults to that section
  - pass it the C standard library section number explicitly
- Returns the found option character, places the argument in a global variable
Useful Library Functions: File I/O

- #include <stdio.h>
- FILE *f = fopen("path/to/file", "r");
- int x, y;
- char c;
- fscanf(f, "%d %d %c", &x, &y, &c);
- fclose(f);
- Read in two integers and a char from a file
- ALWAYS check return values from file I/O functions
  - See man pages for error codes and full documentation
Useful Library Functions: Memory

- void *malloc(size_t s);
- void free(void *ptr);
- malloc(s) allocates s bytes on the heap, returns a pointer there
- Memory is not zeroed
- Memory is not reused automatically – must manually call free on each pointer returned by malloc when done using it
  - Entire memory space is freed on program exit; don’t bother freeing structures in use at the very end of your program
- You’ll need to write these functions yourself later; Get used to how they work now!
Caches – Example

- 16-byte cache, $S = 4$, $E = 1$ (direct mapped), $B = 4$, $m = 8$

<table>
<thead>
<tr>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>t</td>
</tr>
</tbody>
</table>

- Trace:
  - l 0x3f
  - l 0x3e
  - s 0xc0
  - l 0xde
  - l 0xad