15-213 Recitation 6: C and Cache Lab

15 Feb 2016
Ralf Brown and the 15-213 staff
Agenda

- Reminders
- Lessons from Attack Lab
- C Assessment
- Caches
- Cache Lab Overview
- Appendix: Programming Style
- Appendix: valgrind
- Appendix: Contech
Reminders

- Attack Lab is due **tomorrow**!
  - “But if you wait until the last minute, it only takes a minute!” - **NOT**!
- Cache Lab will be released **tomorrow**!

Image credit: pixabay.com
Lessons from Attack Lab

- **Never**, ever use `gets`
  - use `fgets` instead if you need that functionality
- Use functions that pass an explicit buffer length if possible
  - `strncpy/strncat` instead of `strcpy/strcat`, `snprintf` instead of `sprintf`
- Limit `scanf/fscanf` input lengths with `%123s`
- Or use a function that dynamically allocates a large-enough buffer
  - `asprintf` (GNU library) instead of `sprintf`
- If none of those is possible, be very careful about checking input size
- Stack protections make it harder to exploit a buffer overflow – but not impossible
C Assessment

- Can you **easily** answer all of the problems on the following slides?
- If not, please come to the C Bootcamp:
  - Time, Location TBA
- You need this for the rest of the course. **If in doubt, come to the C Bootcamp!**
C Assessment 1: Spot the Errors

```c
int main() {
    int *a = malloc(100 * sizeof(int));
    for (int i=0; i<100; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=0;
    }
    free(a);
    return 0;
}
```
C Assessment 1: Spot the Errors

```c
int main() {  
    int *a = malloc(100 * sizeof(int));  
    for (int i=0; i<100; i++) {  
        if (a[i] == 0) a[i]=i;  
        else a[i]=0;  
    }  
    free(a);  
    return 0;  
}
```

malloc can return NULL – segmentation violation!
C Assessment 1: Spot the Errors

int main() {
    int *a = malloc(100 * sizeof(int));
    for (int i=0; i<100; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=0;
    }
    free(a);
    return 0;
}

malloc can return NULL – segmentation violation!
returned memory is uninitialized – undefined results
int main() {
    int *a = calloc(100, sizeof(int));
    if (a == NULL) {...handle error...}
    for (int i=0; i<100; i++) {
        if (a[i] == 0) a[i]=i;
        else a[i]=0;
    }
    free(a);
    return 0;
}

- **Fixes**
  - use calloc to get zeroed-out memory
  - check a before using it
- **Note:** variable declaration in the “for” statement requires --std=c99 flag to gcc – you'll get an error without it
C Assessment 2: Macros

- What is A?

```c
#define IS_GREATER(a, b) a > b

int is_greater(int a, int b) {
    return a > b;
}

int A = IS_GREATER(1, 0) + 1;
int B = is_greater(1, 0) + 1;
```

- What is B?
C Assessment 2: Macros

- What is A?
  - 0

- What is B?
  - 2

```c
#define IS_GREATER(a, b) a > b
int is_greater(int a, int b) {
    return a > b;
}
int A = IS_GREATER(1, 0) + 1;
int B = is_greater(1, 0) + 1;
```
C Assessment 2: Macros

What is A?
- 0
- int A = 1 > 0 + 1;
- 1 > 1 is false

What is B?
- 2
- is_greater(1,0) returns 1, then we add 1 to that as expected

#define IS_GREATER(a, b) a > b

int is_greater(int a, int b) {
    return a > b;
}

int A = IS_GREATER(1, 0) + 1;
int B = is_greater(1, 0) + 1;
C Assessment 3: Find the Errors

```c
int *foo(int *allocate) {
    int a = 3;
    allocate = malloc(sizeof(int));
    if (allocate == NULL) abort();
    return &a;
}
```
int *foo (int *allocate) {
    int a = 3;
    allocate = malloc(sizeof(int));
    if (allocate == NULL) abort();
    return &a;
}
C Assessment 3: Find the Errors

int *foo(int *allocate) {
    int a = 3;
    allocate = malloc(sizeof(int));
    if (allocate == NULL) abort();
    return &a;
}

Memory leak! allocate is a local copy of the pointer that goes away when the function returns.

int *foo(int **allocate) {
    int a = 3;
    *allocate = malloc(sizeof(int));
    if (*allocate == NULL) abort();
    return &a;
}

To return the memory, we need a pointer to a pointer, and an extra dereference on assignment.
C Assessment 3: Find the Errors

```c
int *foo(int *allocate) {
    int a = 3;
    allocate = malloc(sizeof(int));
    if (allocate == NULL) abort();
    return &a;
}
```

returning the address of a local variable yields unpredictable results (why?)
C Assessment

- Did you know the answers to all of the problems? If not, COME TO THE C BOOTCAMP
Memory Hierarchy

- **Reg**: 100s of bytes, 0.2 ns access
- **L1 cache**: 10s of KB, <1 ns random access
- **L2 cache**: 100s of KB, ~1 ns random access
- **L3 cache (SRAM)**: megabytes, 2-5 ns random access
- **Main memory (DRAM)**: gigabytes, 20-50 ns random access
- **Local secondary storage (local disks)**: terabytes, 30,000,000 ns random access
- **Remote secondary storage (e.g. cloud storage)**: exabytes, >100,000,000 ns access

- **smaller, faster, and more expensive per byte**
- **larger, slower, cheaper per byte**
Caching

- Copy a subset of data from slower storage into faster as it is accessed
- If requested data is not yet cached and must first be copied, that is a “cache miss”
- If requested data is already available in the faster storage, that is a “hit”
- If the cache is full, a miss causes an existing entry to be discarded (“evicted”)
Cache Types

- **Fully-associative cache**: any memory location can be stored in any cache line
  - impractical to build in reasonably large size
- **Direct-mapped cache**: each memory location must be stored in a specific cache line
  - easiest to implement, but has poorer performance
- **N-way set-associative cache**: each memory location is associated with a set of $N$ cache lines (typically 2, 4, 8, or 16), and can be stored in any one of the cache lines within that set
  - compromise – easier to implement than fully-associative, better performance than direct-mapped
Direct-Mapped and Set-Associative Caches

Memory Address

- offset in cache line
- set number (line number if direct-mapped)
- stored in tag to be able to find line

Cache Line

- high-order bits of the memory address stored in this line
- is line valid?
- copy of data at the tagged address
- Least-Recently Used line when evicting (not needed for direct-mapped)
- info to determine
Set-Associative Cache: 2-way Example

- Consider the following eight-entry 2-way associative cache with 64 bytes per cache line
- Address bits 0-5 become the index into the line's data
- Address bits 7-6 are the set number
- Remaining address bits become the tag

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set0</td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td>Set1</td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td>Set2</td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td>Set3</td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>-----</td>
</tr>
</tbody>
</table>
Set-Associative Cache: 2-way Example

- Let's read every 128\(^{th}\) byte starting at 0x1521200
- 0x1521200 is
  0001 0101 0010 0001 0010 0000 0000
- that's the first byte of a line in set 0, with tag 0x15212
- it's a miss, so read from main memory and store in an available line in set 0
Set-Associative Cache: 2-way Example

- Next, we read 0x1521280
  0001 0101 0010 0001 0010 1000 0000
- that's the first byte of a line in set 2, with tag 0x15212
- it's a miss, so read from main memory and store in set 2

<table>
<thead>
<tr>
<th>V</th>
<th>LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0x15212</td>
<td>A B C D</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>- - - -</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>- - - -</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x15212</td>
<td>E F G H</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>- - - -</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>- - - -</td>
</tr>
</tbody>
</table>
Set-Associative Cache: 2-way Example

- **0x1521300:**
  - 0001 0101 0010 0001 0011 **0000 0000**
  - that's the first byte of a line in set 0, with tag 0x15213
  - it's once again a miss, so read from main memory and store in an empty line in set 0
  - also update the LRU info
### Set-Associative Cache: 2-way Example

- **0x1521380:**
  - 0001 0101 0010 0001 0011 1000 0000
  - That's the first byte of a line in set 2, with tag 0x15213
  - Yet another miss, so read from main memory and store in an empty line in set 2
  - Also update the LRU info

<table>
<thead>
<tr>
<th>Set</th>
<th>V</th>
<th>LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set0</td>
<td>1</td>
<td>1</td>
<td>0x15212</td>
<td>A B C D</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0x15213</td>
<td>I J K L</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>....</td>
</tr>
<tr>
<td>Set1</td>
<td>1</td>
<td>1</td>
<td>0x15212</td>
<td>E F G H</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0x15213</td>
<td>M N O P</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
<td>--</td>
<td>....</td>
</tr>
</tbody>
</table>
Set-Associative Cache: 2-way Example

- **0x1521400:**
  - 0001 0101 0010 0001 0100 0000 0000
  - that's the first byte of a line in set 0, with tag 0x15214
  - missed yet again, so read from main memory and store in an empty line in set 0
  - but set 0 is full, so we need to evict someone
  - tag 0x15212 was least-recently used, so it goes
Set-Associative Cache: 2-way Example

- Note how we had to evict a line even though the cache still has empty entries.

<table>
<thead>
<tr>
<th>Set0</th>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0x15214</td>
<td>Q R S T</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x15213</td>
<td>I J K L</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set1</th>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>---</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set2</th>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0x15212</td>
<td>E F G H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x15213</td>
<td>M N O P</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set3</th>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
<td>---</td>
</tr>
</tbody>
</table>
Cache Lab

- Two parts
  - write a cache simulator
  - optimize some code to minimize cache misses
- Programming style will be graded starting now
  - worth about a letter grade on this assignment
  - a summary slide is included as an appendix to this recitation, but be sure to carefully read the style guide
- Details are in the writeup!
If You Get Stuck

- Please read the writeup. *Please read the writeup. Please read the writeup. Please read the writeup!*
- CS:APP Chapter 6
- View lecture notes and course FAQ at [http://www.cs.cmu.edu/~213](http://www.cs.cmu.edu/~213)
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza
- `man malloc`, `man valgrind`, `man gdb`, `gdb's help command`
KEEP CALM and READ THE WRITEUP
Appendix: Programming Style

- Properly document your code
  - header comments, overall operation of large blocks, any tricky bits
- Write robust code – check error and failure conditions
- Write modular code
  - use interfaces for data structures, e.g. create/insert/remove/free functions for a linked list
  - no magic numbers – use #define
- Formatting
  - 80 characters per line
  - consistent braces and whitespace
- No memory or file descriptor leaks
Appendix: valgrind

- A suite of tools for debugging and profiling memory use, among other things
  - find where memory that wasn't freed was allocated
  - track origin of uninitialized values
  - show heap usage over time
  - detect reads and writes of invalid locations
  - detect illegal and double frees
  - trace individual memory accesses (used for cachelab)
  - report on race conditions in multi-threaded programs (useful later in the semester)
valgrind: Finding Memory Leaks

- valgrind --leak-resolution=high --leak-check=full --show-reachable=yes --track-fds=yes ./my_prog <args>

- your program runs as normal, though much, much slower
  - read/write errors and uses of uninitialized values are reported as they occur
  - un-freed memory is reported on program termination
valgrind: Tracing Memory Accesses

- `valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes <prog> <args>`
- writes a line to stdout for each memory operation the program makes
  - instruction fetches
  - data loads
  - data stores
  - data modifies (read followed by write, e.g. from `add $8, (%rsp)`)
- The writeup has details on the output format
Appendix: Contech

- We are rolling out a new method for generating memory traces
- Contech relies on specially-compiled executables that record their memory accesses to a file
  - this is much faster than Valgrind
  - outputs trace in same format as Valgrind, but omits instruction fetches
- More information on using Contech is coming soon
  - cachelab uses a simplified version of the original, which can be found at http://bprail.github.io/contech/