August 26, 2019 Lecture

GMU Fall 2019 CS 367

Connor Baker

Compiled on August 27, 2019 at 8:55pm

Course Overview

CS367 is an Introduction to a field of CS called Systems Programming Systems Programming – Programming services to other software.

- Game Engines
- Operating Systems
- Embedded Systems
- Industrial Automation
- Networking
- Optimizing and Reverse Engineering Code

This course will prepare you for CS471 (Operating Systems), as well as a large number of other such courses.

Course Goals

Goals of prior courses:

- CS 112 and CS 211
  - High level programming
  - Creating applications for users
  - Object oriented paradigm
- CS 262
  - Lower level programming
  - Directly accessing memory and hardware
  - Used for embedded systems
- CS 367
  - Remove all the high-level abstractions to understand what happens “under the hood” in programming

Removing Abstractions

Most CS courses focus on adding abstractions

- Abstract data types (like Interfaces in Java)
- Asymptotic Analysis (Big-O)

Abstractions only go so far

- In systems, it’s important to understand the underlying implementation
  - This is doubly true in the presence of bugs!

Outcomes

After taking this course you should

- Become more effective programmers
  - Gain more knowledge about how to find and eliminate bugs *efficiently* (that means no more `printf` statements everywhere)
- Prepare for later systems programming courses in CS
  - CS 440 - Compilers
  - CS 455 - Networking
  - CS 465 - Architecture
  - CS 468 - Secure Programming
  - CS 469 - Security Engineering
  - CS 471 - Operating Systems
  - CS 475 - Concurrent and Distributed Systems

Great Realities in Programming

Observation 1: Same-signed products

File: `int_of.c`.

We know from our previous math courses that $(\forall x \in \mathbb{Z})(x^2 \geq 0)$.

Floating point numbers have this property as part of their definition, so it is true for them by construction.

But what about C's integral data types? Nope!

C's integer data types can very easily suffer from overflows and underflows.

Observation 2: Associative property of addition

File: `fp_of.c`.

We know from our previous math courses that $(\forall x, y, z \in \mathbb{Z})((x + y) + z = x + (y + z))$.

C's integral data types also have this property. The addition may overflow or underflow, but it is associative.

Floating point numbers do not have this property. This is due to the way that floating point numbers were defined – they are meant to represent a great *range* of numbers, but not necessarily with great *accuracy*. That is to say that there are gaps between each numbers.

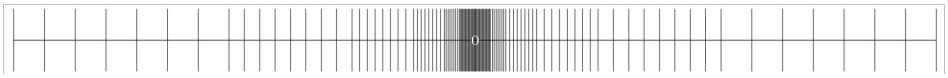

Figure 1: IEEE 754 *tiny* floating point (normal and subnormal) numbers.

Fun math fact: Those gaps above can never be closed. Because the real numbers aren't denumerable, they can never be encoded in a single data type. The rationals are denumerable, but they aren't an order complete field, so they have gaps (the reals are the closure of the rationals, so they are order complete and do not have gaps).

Computer arithmetic: There are a finite number of bits available to represent a number. The way that we encode the number in those bits changes its properties under different operations.

Integers have:

- Commutativity
- Associativity
- Distributivity

Floating point numbers have:

- Monotonicity (for the importance of this property, see **Aside 2** below)
- Values of signs

Omitted: The following code snippets are omitted:

Aside 1 Beware integer underflow and overflow. (See `int_index.c`).

Aside 2 Floating point numbers don't have a uniform distribution – by construction they are clustered around zero.

We know from math that

$$(\forall a \in \mathbb{Q})(\forall b \in \mathbb{Q} : b > 0)(a + b > a).$$

This property does not hold for floating point numbers. Because the numbers are non-uniform, it is possible to add some value to your floating point number such that it doesn't clear the gap between representable values. (See `fp_index_a.c`, `fp_index_b.c`).

With a **`float`** of 16,777,216, adding one to it won't change its value since it doesn't clear the gap between the current value and the next representable number.

Observation 3: Knowing assembly is helpful

Chances are you'll never need to write a program in assembly. However, you'll almost certainly have to look at it at some point to figure out why something isn't working as it's supposed to (see `compiler_a.c` and `compiler_b.c`). You'll have to do this with some frequency if you're doing any of the following:

- Tuning program performance
  - Understanding what optimizations are done or not done by the compiler
  - Sussing out inefficiencies
- Implementing system software
  - Compiles target machine code and will change your code in odd ways
- Creating or fighting malware
  - The x86 ISA is a critical language for reverse engineering

Observation 4: Memory matters

- Memory isn't unbounded
  - It must be allocated and managed by each process
  - Many applications are memory dominated
- Memory reference bugs are extremely problematic
- Memory performance is not uniform
  - Exploiting data locality greatly affects program performance
  - Adjusting the data types and code to fit into a cache can also lead to better performance

Memory Reference Bug

File: `mem_bug.c`.

Consider the following function `double_fun` which is passed a number from the command line.

```
1  void double_fun(int index) {
2      double d[1] = {3.14};
3      long a[2];
4      a[index] = BIG_NUM;
5      printf("Setting a[%ld] = %ld; d[0] = %lf\n", index, BIG_NUM, d[0]);
6  }
```

Running it yields:

```
kandrea@zeus-2$ ./mem_bug 0
Setting a[0] = 1073741824; d[0] = 3.140000 kandrea@zeus-2$ ./mem_bug 1
Setting a[1] = 1073741824; d[0] = 3.140000 kandrea@zeus-2$ ./mem_bug 2
Setting a[2] = 1073741824; d[0] = 0.000000 kandrea@zeus-2$ ./mem_bug 3
Setting a[3] = 1073741824; d[0] = 3.140000 kandrea@zeus-2$ ./mem_bug 4
Setting a[4] = 1073741824; d[0] = 3.140000 Segmentation fault
kandrea@zeus-2$ ./mem_bug 5
Setting a[5] = 1073741824; d[0] = 3.140000 Segmentation fault
```

The stack as the program executes looks like:

Array	Stack
`a[5]`	Saved state
`a[4]`	Return address
`a[3]`	Saved state
`a[2]`	`d[0]`
`a[1]`	`a[1]`
`a[0]`	`a[0]`

Memory Referencing Errors

C (and C++) do not provide any memory protection

- They don't perform out of bounds checking for arrays or references
- They don't check for point values before dereferencing
- They don't check for abuses of `malloc` or `free`

This can lead to really nasty bugs

- You can easily corrupt unrelated objects which reside in memory (like `a[3]` being assigned to something affecting the value of `d[0]`)
- Sometimes the effects of these bugs are only observed long after they are generated

Observation 5: Performance is more than just Asymptotic Complexity

When dealing with Big-O notation constant factors are ignored. However, in practice constant factors are usually a *deciding* factor in choosing which algorithm to use.

In general, we can't assume that the operation count is a prediction of performance.

- Performance is impacted by memory access, caching, paging, etc
  - You should know how to optimize at multiple levels
  - To do this requires an understanding of how programs are compiled and executed, how to measure a program's performance, and how to improve performance without destroying modularity or generality

Memory Performance Example

Files: `array_access.c`, `Readme.txt`, and `prof_results.txt`.

Consider the following `for` loops:

```
for(i = 0; i < SIZE; i++) {
  for(j = 0; j < SIZE; j++) {
    dst[i][j] = src[j][i];
  }
}
```

and

```
for(j = 0; j < SIZE; j++) {
  for(i = 0; i < SIZE; i++) {
    dst[i][j] = src[j][i];
  }
}
```

Functionally, they are equivalent. However, profiling them reveals something surprising:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 58.75      4.37     4.37      500     8.73     8.73  transpose_b
 41.93      7.48     3.12      500     6.23     6.23  transpose_a
  0.14      7.49     0.01                             main
```

Just changing the order in which we iterate through the array net us a 28.6% performance improvement. That's the magic of exploiting repeated local memory access.

Observation 6: Computers do more than just Execute Programs

In addition to running your program, you also need to worry about putting data into it and getting data out of it.

There's a whole slew of issues that need to be taken into account:

- I/O is system critical to program reliability and performance

- Networking
  - Concurrent Operations by Autonomous Processes
  - Coping with Unreliable Media (eg. wireless)
  - Cross-Platform Compatibility
  - Complex Performance Issues

Course Perspective

Most Systems courses are “Builder-Centric.”

- Computer Architecture (CS465)
  - Design a pipelined processor
- Operating Systems (CS471)
  - Design portions of an operating system
- Language processors (CS440)
  - Write a compiler for a simple language
- Networking (CS455)
  - Implement networking protocols

This course is “Programmer-Centric.”

The purpose of this course is to show how, by knowing the underlying system, you can be a more effective programmer.

This course will enable you to

- Write programs that are more reliable and efficient
- Incorporate features that tie into the OS (like concurrency and signals)
- use debuggers effectively
- Reverse engineer programs (or at the least provide a good introduction to this)

Course Prerequisites

In addition to the listed course prerequisites, you should know how to

- Write programs in C
- Know C data types
- Know the C standard library
- Know bit-wise operators

- Know how to use pointers
- Know how to create structures
- Know how dynamic memory allocation works
- Know how to make/read a Makefile
- Know the basic process of compiling a program
- Know basic debugging via gdb
- Know basic unix commands

Course Recitations (7.5% of the grade)

We will be doing small, ungraded exercises from time to time in lecture. The main time for exercises is in Recitation.

- You are all in one of the Recitation sections that meet on Fridays
  - Your Recitation instructor is also your GTA for project grading issues.
- Each Recitation is exercise based.
  - These exercises will be given to you at the beginning of the recitation and you will work in small groups on these.
  - You will not be graded on how well you do, only on participating.
  - The goal is working together on problem solving.

Course Projects (30% of the grade)

There will be at least 3 projects this semester.

- Each Project will be on a central topic for the course. (eg. Floating Point Arithmetic, Dynamic Memory Allocation, x86-64 Assembly.)
- Each Project is an individual effort.
  - The recitations are all for group work.
  - The projects are for individual work. There should be no collaboration on the project design and implementation.
  - We will run plagiarism detection software on all submitted projects.
- All Projects will be Tested and Graded on zeus.vse.gmu.edu
  - Make sure to compile and test your code on Zeus
  - Make sure to verify what you submit are the right files!

Academic Integrity

The Honor Codes are listed in the Syllabus and both apply to all work. For Projects:

- You may not share code (copying, retyping, looking at other code)
- You may not coach anyone on writing projects.
- You may not copy or purchase any code from a previous course or from anywhere online.

The Penalty for Cheating is a Failing Grade in the course.

You can explain how to use the systems or tools, help explain the specifications, and discuss output.

Getting Help

If you feel lost, overwhelmed, or just generally behind…

- Come in to any of the Professor or GTA office hours.
- Post questions on Piazza
  - If the question is on the Project design or includes code, post it as private.
- Email me (kandrea@gmu.edu)
  - I can add extra material to the next lecture to answer questions.
  - I can also make video supplemental lectures as time permits to answer.

Piazza is our Forum for the course. Instructions will be forthcoming. Our Blackboard pages (Lecture and Recitation) will have all:

- Slides, Recitation Assignments, and Projects

Homework

Reading:

1. Read through Chapter 1 of the textbook
2. Review C (K&R Chapters 1, 5, 6, and 7)