Course Overview

CS 485: Systems Programming
Fall 2015

Instructor:
James Griffioen

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/public/instructors.html)
Course Goals

- **Develop an understanding of computing systems as a whole.**
  - Most programs run in the context of a *computing system* comprised of several distinct components including the underlying hardware, libraries, operating systems, windowing systems, networks interfaces, network services, etc.
  - Most courses are focused on a particular aspect of a computing system (e.g., the programming language). This course is designed to show how all of the components fit together and collectively form the context in which an application executes.
  - Students should come away with a complete, demystified, view of the system.

- **Develop better programmers**
  - Understanding the system as a whole helps programmers identify the causes of problems with their programs.
  - Studying systems from a *programmer’s perspective* allows students to focus on features of the system that are important for programming, as opposed to focusing on the design tradeoffs that go into the building of a particular system component.
  - Experience comes from developing and running programs on real machines.
  - While abstraction is wonderful, it must be grounded in reality.

- **Lay the foundation for upper level classes**
  - Many upper division courses assume the ability to write and debug large programs that interact with a variety of components of a system (e.g., compilers, operating systems, databases, networking, graphics, etc).
Programs are just a small part of the picture

- Even simple programs are part of a larger system.
- They rely on several other system components to “run”.
- In many cases, programming errors arise because of an incorrect assumption about the way some other component of the system actually works (or worse yet, not having thought about the interaction with the other component at all).

*Debugging your program* often requires an understanding of the system in which it is running.

*Writing efficient code* requires an understanding of the system in which it will be run.

*Writing secure code* requires an understanding of the system in which it will be run.

Unfortunately, how a computer system does something is all-too-often pure “magic” to programmers. We need to demystify the system. Computing systems are not magic.

**So what happens when you run a program?**
- Consider a simple program
- Consider a web browser
Abstraction Is Good But Don’t Forget Reality

- Most CS and CE courses emphasize abstraction
  - Abstraction matches the way we (humans) think
  - Abstraction often hides implementation details
  - Abstractions can hide complexity

- However, abstractions are abstractions, not reality
  - The hardware often has limits that the abstraction does not have.
  - Hiding the details of the implementation makes it hard to know if the abstraction can be used in conjunction with other components of the system.
  - Hiding complexity can lead to inefficiency.

- Examples where things can go wrong:
  - Math constructs do not always map onto computer architectures
    - C ints are not the same as integers
    - C floats do not always behave like reals
  - Programming languages hide the instructions that are actually executed
  - Use of memory affects correctness and performance
  - Programs interact with lots of other “stuff”
Expressions are not Math equations: 
Ints are not Integers, Floats are not Reals

- **Example 1:** Is $x^2 \geq 0$?
  - Float’s: Yes!
  - Int’s:
    - $40000 \times 40000 \rightarrow 1,600,000,000$
    - $50000 \times 50000 \rightarrow ??$

- **Example 2:** Is $(x + y) + z = x + (y + z)$?
  - Unsigned & Signed Int’s: Yes!
  - Float’s:
    - $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
    - $1e20 + (-1e20 + 3.14) \rightarrow ??$

Source: xkcd.com/571
High-Level Languages are mapped to Assembly

- Chances are, you’ll never write programs in assembly
  - Compilers are much better & more patient than you are
- But: Understanding assembly is key to machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!
Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free

- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated

- How can I deal with this?
  - Program in Java, Ruby or ML
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) ➞ 3.14
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) ➔ 3.14
fun(1) ➔ 3.14
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) ➞ 3.14
fun(1) ➞ 3.14
fun(2) ➞ 3.1399998664856
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) ➞ 3.14
fun(1) ➞ 3.14
fun(2) ➞ 3.1399998664856
fun(3) ➞ 2.00000061035156
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) ➞ 3.14
fun(1) ➞ 3.14
fun(2) ➞ 3.1399998664856
fun(3) ➞ 2.00000061035156
fun(4) ➞ 3.14, then segmentation fault
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) ➞ 3.14
fun(1) ➞ 3.14
fun(2) ➞ 3.1399998664856
fun(3) ➞ 2.00000061035156
fun(4) ➞ 3.14, then segmentation fault

■ Result is architecture specific
Memory Referencing Bug Example

double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}

fun(0) ➔ 3.14
fun(1) ➔ 3.14
fun(2) ➔ 3.1399998664856
fun(3) ➔ 2.00000061035156
fun(4) ➔ 3.14, then segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Saved State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d7 ... d4</td>
<td>a[1]</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>a[0]</td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
</tr>
</tbody>
</table>

4 3 2 1
Memory System Performance Example

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

```c
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```c
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

21 times slower
(Pentium 4)