Overview

Fall 2019

Instructor:
Murat Manguoglu (Sections 1 -2)

Unless otherwise noted adapted from slides of the textbook: http://csapp.cs.cmu.edu/
Overview

- Course theme
- Five realities
- How the course fits into the CENG curriculum
- Academic integrity
Course Theme: Abstraction Is Good But Reality Matters

- Most CENG courses emphasize abstraction
  - Abstract data types
  - Asymptotic analysis

- These abstractions have limits
  - Especially in the presence of bugs, performance constraints
  - Need to understand details of underlying implementations

- Useful outcomes of CENG331
  - Become more effective programmers and computer scientists
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune your code for performance
  - Prepare for later “systems” classes in CS/CENG
    - Intro. to Parallel Computing, Compilers, Operating Systems, Networks, Embedded Systems, etc.
Reality #1: Ints are not Integers, Floats are not Reals

Example 1: Is $x^2 \geq 0$?

- Float’s: Yes!
- Int’s:
  - $40000 \times 40000 \rightarrow 1600000000$
  - $50000 \times 50000 \rightarrow ??$

Example 2: Is $(x + y) + z = x + (y + z)$?

- Unsigned & Signed Ints: Yes!
- Floats:
  - $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
  - $1e20 + (-1e20 + 3.14) \rightarrow ??$

Source: xkcd.com/571
Computer Arithmetic

- **Does not generate random values (“most of the time”)**
  - Arithmetic operations have important mathematical properties
  - Random errors, bit flips, etc. are possible on large scale computers

- **Cannot assume all “usual” mathematical properties**
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs

- **Observation**
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers
Reality #2: You’ve Got to Know Assembly (and the processor)

- 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 *(or can not be done)*
    - 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!
Reality #3: Memory Matters
Random Access Memory Is an Unphysical Abstraction

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated

- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space

- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements
Memory Referencing Bug Example

typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}

- Result is system specific

volatile keyword indicates that a value may change between different accesses, even if it does not appear to be modified. This keyword prevents an optimizing compiler from optimizing away subsequent reads or writes and thus incorrectly reusing a stale value or omitting writes.
Memory Referencing Bug Example

typedef struct {
    int a[2];
    double d;
} struct_t;

<table>
<thead>
<tr>
<th>fun(i)</th>
<th>3.14</th>
<th>3.14</th>
<th>3.1399998664856</th>
<th>2.00000061035156</th>
<th>3.14</th>
<th>Segmentation fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun(2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun(4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun(6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Explanation:

<table>
<thead>
<tr>
<th>Critical State</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d7 ... d4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d3 ... d0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Location accessed by fun(i)
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, Python, ML, ...
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)
Reality #4: There’s more to performance than asymptotic complexity

- Constant factors matter too!
- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality
Memory System Performance Example

```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];
}
```

4.3ms  81.8ms

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

2.0 GHz Intel Core i7 Haswell
Why The Performance Differs

![3D graph showing read throughput (MB/s) vs. size (bytes) and stride (x8 bytes).]

- **copyij**
- **copyji**
Figure 8: SpMV Performance Profiles for optimized-BCSR on a dense matrix on a single core. On each platform, each square is an \( r \times c \) implementation, \( 1 \leq r, c \leq 8 \), colored by its performance in MFlops/sec, and labeled by its speedup over OSKI. The top of the speed range for each platform is the performance bound from the Roofline model, in the last column of Table 5.

Reality #5:
Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance

- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues
Course Perspective

- Most Systems Courses are Builder-Centric
  - Operating Systems
    - Implement sample portions of operating system
  - Compilers
    - Write compiler for simple language
  - Networking
    - Implement and simulate network protocols
Course Perspective (Cont.)

- **Our Course is Programmer-Centric**
  - Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer
  - Enable you to
    - Write programs that are more reliable and efficient
    - Incorporate features that require hooks into OS
      - E.g., concurrency, signal handlers
  - Cover material in this course that you won’t see elsewhere
  - Not just a course for dedicated hackers
    - We bring out the hidden hacker in everyone!
Role within CENG Curriculum

CENG331
Databases
Data Reps. Memory Model

CENG334
Operating Systems
Processes Mem. Mgmt

CENG444
Language Processors
Machine Code

CENG371
Scientific Computing
Arithmetic

CENG336
Embedded Systems
Execution Model Memory System & Pipelining/multithreading

CENG478
Parallel Computing

CENG232
Logic Design
Textbook

- Randal E. Bryant and David R. O’Hallaron,
  - [http://csapp.cs.cmu.edu](http://csapp.cs.cmu.edu)
  - This book really matters for the course!
    - How to solve labs
    - Practice problems typical of exam problems
Reference Texts

- **Computer Organization and Design: Hardware/Software Interface** by Patterson and Hennessy

- **Computer Architecture: A Quantitative Approach** by Patterson and Hennessy
Course Components

- **Lectures**
  - Higher level concepts
    - Mondays
      - 9:40-11:30 (1), 15:40-17:30 (2) BMB3
    - Wednesdays
      - 10:40-11:30 (1), 15:40-16:30 (2) BMB3

- **Take-home and/or In-lab exams (4)**
  - Take-home and/or in lab sessions
  - Provide in-depth understanding and hands on experience on an aspect of computing systems; programming and performance measurement

- **Exams (midterm I + midterm II + final)**
  - Test your understanding of concepts & mathematical principles

- **Quizzes**
  - Random quizzes in class, can be done with a group of at most 3 people, graded in ternary {0,1,2}

---

<table>
<thead>
<tr>
<th>Technique</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elaborative interrogation</td>
<td>Moderate</td>
</tr>
<tr>
<td>Self-explanation</td>
<td>Moderate</td>
</tr>
<tr>
<td>Summarization</td>
<td>Low</td>
</tr>
<tr>
<td>Highlighting</td>
<td>Low</td>
</tr>
<tr>
<td>The keyword mnemonic</td>
<td>Low</td>
</tr>
<tr>
<td>Imagery use for text learning</td>
<td>Low</td>
</tr>
<tr>
<td>Rereading</td>
<td>Low</td>
</tr>
<tr>
<td>Practice testing</td>
<td>High</td>
</tr>
<tr>
<td>Distributed practice</td>
<td>High</td>
</tr>
<tr>
<td>Interleaved practice</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

*Improving Students’ Learning With Effective Learning Techniques Promising Directions From Cognitive and Educational Psychology, John Dunlosky, Katherine A. Rawson, Elizabeth J. Marsh, Mitchell J. ,Nathan and Daniel T. Willingham, *Physiological Science in the Public Interest*
Getting Help

- Odtuclass: [http://odtuclass.metu.edu.tr](http://odtuclass.metu.edu.tr)
  - Course discussion forums, announcements, exams, assignments, grades

- Lecture notes (posted before the lecture)
  [www.ceng.metu.edu.tr/~manguoglu/ceng331](http://www.ceng.metu.edu.tr/~manguoglu/ceng331)

- Email communication:

  If you have a specific question that is not beneficial to others, you can send an e-mail to the instructor or to your teaching assistants. However make sure that the subject line starts with CENG331-Section#, state your first, last name and ID # to get faster reply.
Email communication:

- **Specific questions:**
  
  Büşra Akarsu  
  Çağrı Utku Akpak  
  Erbil Yakışkan  
  Merve Taplı  
  
  (busra@ceng.metu.edu.tr)  
  (capaki@ceng.metu.edu.tr)  
  (erbil@ceng.metu.edu.tr)  
  (tapli@ceng.metu.edu.tr)  

- **My Office hours:**
  
  Murat Manguoglu: Mondays 14:00-15:00 - A407  
  Email: manguoglu@ceng.metu.edu.tr
Policies: Take-home/in-lab assignments, exams and quizzes

■ Study groups
  ▪ You are encouraged to study in groups

■ Work groups
  ▪ You must work alone on all take-home, in-lab, midterm, final exams.
  ▪ You can work in groups (of at most 3) for quizzes

■ Midterm and Final Exams
  ▪ Written exams
  ▪ You can bring one A4 size handwritten notesheet, they are collected at the end of the exam and will be graded in binary \{0,1\}
Makeups

There are no makeups unless:

- Major illness (sometimes this includes Flu), death in family, ...
- Submit an official report to the instructor as soon as possible
Policies: Grading

- Written Exams (65%): midterm I (20%), midterm II (20%), final (25%)

- Assignments (Take-home and/or In-lab) (30%)

- Quizzes (5%)
Programs and Data

- **Topics**
  - Bits operations, arithmetic, assembly language programs
  - Representation of C control and data structures
  - Includes aspects of architecture and compilers

- **Take-home and/or in-lab assignments**
  - Bomblab: Defusing a binary bomb
  - Attacklab: The basics of code injection attacks
Processor Architecture

- **Topics**
  - Y86-64 architecture
    - Pipelining and hazards
    - Control structures

- **Take-home and/or in-lab assignments**
  - Architecture
Code optimization and Memory Hierarchy

Topics

- Code optimization
- Memory technology, memory hierarchy, caches, disks, locality
- Includes aspects of architecture and OS

Take-home and/or in-lab assignments

- Performance: Improve the performance of a kernel which is a bottleneck in an application
Virtual Memory

Topics

- Virtual memory, address translation
- Includes aspects of architecture and OS
Other topics (if time permits)

Topics

- Tensor Processing Units (TPUs)
- Graphical Processing Units (GPUs)
- Quantum Processors
- Multicore Architectures
- Multithreading
- Very Large Instruction Word Machines
In-lab and Take-home exam Rationale

- Each assignment has a well-defined goal such as solving a puzzle or winning a contest

- Doing the lab should result in new skills and concepts

- We try to use competition in a fun and healthy way
  - Set a reasonable threshold for full credit
  - Post intermediate results (anonymized)
Cheating: Description

What is cheating?

- Sharing code: by copying, retyping, **looking at**, or supplying a file
- Describing: verbal description of code from one person to another.
- Coaching: helping your friend to write a lab, line by line
- Searching the Web for solutions
- Copying code from a previous course or online solution
  - You are only allowed to use code we supply, or from the CS:APP website

What is NOT cheating?

- Explaining how to use systems or tools
- Helping others with high-level design issues
Cheating: Consequences

- Penalty for cheating:
  - Disciplinary action

- Detection of cheating:
  - We have sophisticated tools for detecting code plagiarism
  - And other forms of cheating

- Don’t do it!
  - Start early
  - Ask the staff for help when you get stuck
METU Honor Code

Every member of METU community adopts the following honor code as one of the core principles of academic life and strives to develop an academic environment where continuous adherence to this code is promoted.

"The members of the METU community are reliable, responsible and honourable people who embrace only the success and recognition they deserve, and act with integrity in their use, evaluation and presentation of facts, data and documents."

METU Academic Integrity Guide for Students:

https://cow.ceng.metu.edu.tr/Courses/download_courseFile.php?id=8737
Welcome and Enjoy!