1 Instructor

Dr. Tom Nurkkala
Associate Professor, Computer Science and Engineering
Director, Taylor Center for Missions Computing
Office: Euler Science Complex 211
E-Mail: `tnurkkala@cse.taylor.edu`
Phone: 8-5163
Hours: MW 2:00-4:00 or by appointment

2 Credit

This course is based on the Introduction to Computer Systems course at Carnegie Mellon University (CMU). Professors Randal Bryant and David O'Hallaron, who developed the course and wrote the textbook (Section 5), have generously made available course material (including notes, syllabi, laboratory exercises, and lecture slides) for use at other institutions. I make considerable use of their excellent materials throughout the course (including in this document!) and wish to credit and thank them.

I have also drawn liberally from previous offerings of this course at Taylor University, and wish to thank Professor Jonathan Geisler for his kind assistance.

3 Course Overview

The aim of the course is to help you become a better programmer by teaching you the basic concepts underlying all computer systems. I want you to learn what really happens when your programs run, so that when things go wrong (as they always do) you will have the intellectual tools to solve the problem.

Why do you need to understand computer systems if you do all of your programming in high-level languages? In most of computer science, we're pushed to make abstractions and stay within their frameworks. But, any abstraction ignores effects that can become critical. As an analogy, Newtonian mechanics ignores relativistic effects. The Newtonian abstraction is completely appropriate for bodies moving at less than $0.1c$, but higher speeds require working at a greater level of detail.

The following "realities" are some of the major areas where the abstractions you've learned in previous classes break down:

1. *An `int` isn't an integer; a `float` isn't a real.* Our finite representations of numbers have significant limitations, and because of these limitations we sometimes have to think in terms of bit-level representations.

2. *You've got to know assembly language.* Even if you never write programs in assembly, the behavior of a program sometimes cannot be understood based purely on the abstraction of a high-level language. Furthermore, understanding the effects of bugs requires familiarity with the machine-level model.

3. *Memory matters.* Computer memory is not unbounded. It must be allocated and managed. Memory referencing errors are especially pernicious. An erroneous updating of one object can cause a change in some logically unrelated object. Also, the

combination of caching and virtual memory provides the functionality of a uniform unbounded address space, but not the performance.

4. *There is more to performance than asymptotic complexity.* Constant factors also matter. There are systematic ways to evaluate and improve program performance.

5. *Computers do more than execute instructions.* They also need to get data in and out and they interact with other systems over networks.

By the end of the course, you will understand these realities in some detail. As a result, you will have learned skills and knowledge that will help you throughout your education and career as a computer scientist.

4 Learning Objectives

By the end of the semester, you should be able to:

1. Describe how a computer represents integers internally.
2. Convert twos complement binary values to decimal.
3. Convert binary to decimal and hexadecimal.
4. Understand how a computer represents fractional values.
5. Convert a fractional decimal value to its IEEE single- or double-precision representation.
6. Read x86 assembly code.
7. Write simple x86 assembly functions.
8. Describe how a computer executes a function call including parameters, return values, and the stack.
9. Describe how compilers convert C code to x86 assembly.
10. Explain how various C data structures are accessed in x86 assembly.
11. Describe the motivation for a memory hierarchy.
12. Explain how a memory hierarchy works.
13. Understand how a cache is organized internally.
14. Create programs that take advantage of the memory hierarchy.
15. Explain how a linker works.
16. Contrast static and dynamic linking.
17. Explain what a system call is and how system calls work.
18. Explain the fork system call.

19. Explain the exec family of system calls.
20. Describe how hardware and the operating system cooperate to deal with exceptional situations.
21. Measure the performance of an application accurately.
22. Improve the performance of an application based on good measurement techniques.
23. Understand virtual memory and how it works.
24. Explain the benefits of virtual memory.
25. Explain the role of the translation look-aside buffer.
26. Describe the virtual memory organization of a process running with Linux on an x86 processor.
27. Explain how a dynamic memory allocator works.
28. Be familiar with garbage collection.
29. Be familiar with common memory bugs and how to avoid them.
30. Explain what concurrent computation is and how to achieve it.
31. Contrast process-based concurrency with I/O multiplexing-based concurrency.
32. Describe why synchronization is necessary.
33. Be familiar with how to achieve synchronization.

5 Texts

The text by Bryant and O'Halaron[1] is required. We will refer to it as `CS:APP`. You are *encouraged* to purchase—or have readily available—the updated edition of "K&R," the classic reference on C.[2]

It is *very important* that you read carefully the assigned readings in the `CS:APP` book. You will get the most benefit if you read assigned passages *before* they are covered in class.

The textbook includes many practice problems throughout the body of each chapter. These are straightforward exercises that help you understand the material you have just read by using it *immediately*. For self-study, solutions for all practice problems appear at the end of each chapter.

In short, the best way to use the textbook to enhance your learning is as follows:

1. Read the assigned readings *before* the corresponding class.
2. Work the practice problems *immediately* as you encounter them in the text.
3. Check your work—and your understanding—with the answer key.

6 Labs

The labs come directly from the textbook authors. They have a very nice system set up so that you can test and submit all your programs online and will already know how you are doing on the assignment prior to submission. This should make it very easy to know when you have a correct solution and when you need to keep working.

Please do not look for solutions online. Doing so constitutes cheating. The authors have introduced randomness into the assignments so solutions posted by others may be wrong anyway.

You will use Linux for all lab work. For some labs, you will be given a binary that works on the CSE machines, but should probably work on any recent vintage of Linux. For others, you are required to develop code for a Linux box. You may use any machine to develop, but *it must run correctly on the CSE machines*. You must check your work on the machines in the laboratories before submitting it.

Most low-level systems code is written in˜C, and you will be required to do the same. You will also be required to read, understand, generate, or otherwise fiddle with x86 assembly on a Linux box.

7 Evaluation

The grading breakdown for the course is as follows:

Deliverable	Weight
Assignments (weekly)	20%
Labs (x6)	40%
Exams (x3)	25%
Final	15%
Total	100%

Refer to the Periodic Table of the Grades (on Moodle) for the grading scheme. I reserve the right to award a higher grade than strictly earned; outstanding attendance and class participation figure prominently in such decisions.

8 Moodle

The Computer Science and Engineering department uses Moodle as our Learning Management System. The URL for Moodle is `https://cms.cse.taylor.edu`. To sign on to the course site for the first time, you will need an enrollment key. The key for this course is `nerds4christ`.

You are responsible for checking Moodle regularly to keep up with assignment due dates and other announcements posted to the site. For due dates, the Moodle calendar is your friend.

9 Classroom Expectations

Following are my expectations about classroom conduct.

9.1 Attendance

Attendance is required. I will be in class each day, and I expect you to be there also. I will log who attends class.

In general, I am very understanding about students who must miss class due to a sanctioned Taylor activity, medical appointment, job interview, family emergency, and the like. If possible, let me know in advance that you will not be in class; I will work with you to arrange make-up instruction, homework, exams, etc.

9.2 Conduct

I expect you to be prepared, awake, aware, and participatory during class. I will not hesitate to ask you to stand or move if you are distracted or sleepy.

I expect you to join in discussions, respond to questions from me and from your colleagues, and ask questions of me. I expect you to hold my feet to the fire if I am being unclear, unkind, or contradictory.

9.3 Gizmos

You may not use a laptop, tablet, or similar device to check e-mail, engage in social networking, surf the web, or any other activity not directly relevant to current classroom activity.

If you use an electronic gizmo during class for legitimate academic purposes (e.g., note taking), be prepared to demonstrate relevant use on demand at any time.

10 Academic Integrity

As a student at an institution whose goal is to honor Christ in all that it does, i expect you to uphold the strictest standards of academic integrity. You must do your own work, cite others when you present their work, and never misrepresent your academic performance in any way. Violation of these standards stains the reputations of you as a student, Taylor as an institution, and Jesus as our Lord. Such a violation may result in your failing the course and other disciplinary action by the University. Refer to the Taylor catalog for the official statement of these ideas.

10.1 What Constitutes Cheating?

For purposes of this course, the following are *non-exhaustive* examples of violations of academic integrity.

1. Sharing code or other electronic files by copying, retyping, looking at, or supplying a copy of a file from this or a previous semester. Be sure to store your work in protected directories, and screen lock or log off a lab machine to prevent others from copying your work without your explicit assistance.

2. Sharing written assignments or exams by looking at, copying, or supplying an assignment or exam.

3. Using other's code. Using code from this or previous offerings of the class, from courses at other institutions, or from any other source (e.g., software found on the Internet).

4. Looking at other's code. Although mentioned above, it bears repeating. Looking at other students' code or allowing others to look at yours is cheating. There is no notion of looking "too much," since no looking is allowed at all.

10.2 What Does Not Constitute Cheating?

In contrast, the following are non-exhaustive examples of activities that do not violate academic integrity.

1. Clarifying ambiguities or vague points in class handouts or textbooks.
2. Helping others use the computer systems, networks, compilers, debuggers, profilers, or other system facilities.
3. Helping others with high-level design issues.
4. Helping others with high-level (not code-based) debugging.
5. Using code from the `CS:APP` website or from the class web pages.

Be sure to store your work in protected directories, and log off when you leave an open cluster, to prevent others from copying your work without your explicit assistance.

References

[1] Randal Bryant and David O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, second edition, 2011. ISBN 978-0-13-610804-7.

[2] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988. ISBN 978-0-13-110362-7.