Introduction to Computer Systems
Computer Science 210
Fall 2024

Instructor: Jeff Chase

CPS 210 is an introduction to systems software and computer architecture, with programming exercises in the C language on the Linux operating system. *Prerequisite:* CPS 201.

Learning objectives

1. (Overall) Understand how software runs on real machines and operating systems, including implementation of selected programming language constructs (e.g., functions/procedures, loops, conditionals).
2. Understand the key programming language concepts within the C language (e.g., types, operators, control flow) and the surrounding ecosystem (e.g., compiler, linker, debugger).
3. Understand *memory safety* and *memory protection*, to establish concepts underlying computer security and modern safe programming languages.
4. Understand foundations of the instruction set architecture (ISA) for CPU hardware and how C programs are mapped to machine-level instructions.
5. Understand the role of the operating system, OS abstractions provided to user programs (e.g., processes, files), and how the underlying hardware enables them.
6. Implement non-trivial programs in C that interact with details of the underlying machine architecture (e.g., stack layouts) or interfaces provided by the operating system (e.g., threads, mutexes).
7. Understand key factors that impact performance and resource efficiency as a result of hardware and software running underneath your programs (e.g., I/O, caching, multi-core).

Changes. This version of CPS 210 differs from earlier offerings:

- Modular flow follows the online textbook *Dive Into Systems (DIS)*.
- Focus on software and safety. Reduce overlap with CPS 250 (Architecture).
- Introduce architecture topics with a simplified assembly language and machine emulator called Facile. Continue with x86-64, focusing on data representation, procedure call/return, and stack security.

Here is a plan of lectures and readings for the semester. For each lecture session there is a small lab exercise. Some lectures are replaced with *topic review* sessions: examples and exercises for key concepts, with no lab. Any new material on topic review days is out of scope for exams—unless we need a weather makeup day.

1 C: A Systems Programming Language

This module gets you started with C, the foundational language for efficient programs with access to the full power of the machine and operating system. We revisit key ideas from CS 201 from the perspective of C programming. C is a programming language without guardrails: because it lacks (or predates) safety features of languages like Java or Rust, it has certain hazards to lure you into dangerous mistakes, some of which

open security holes for hackers to exploit. Understanding how and why can make you a better programmer with appreciation for the safety of modern languages and the costs of providing that safety.

1. Course intro. The objectives are to introduce the course and get up and running with C. We touch on several topics to discuss in more detail later. Programs and processes. How a computer runs a program: systems programming and the machine. Example: the Facile abstract machine. Overview of programming languages, compilation, libraries, and linking. Similarities between C and Java. Differences: programming without objects. The C standard library (stdlib/libc), I/O channels, and `printf`.
Reading. Facile ASM document (focus on §1, §2, and §3.3). DIS Chapter 1 and/or Appendix 1 (Chapter 16), also see §2.9.1.
Lab: setup, debug. C/Linux development, building with `make`, debugging with `gdb`.

2. Systems programming and memory safety. Memory space: where is your data? Types: numbers, characters (`char`) as byte-size numbers. Compound types: arrays and structs. C constructs and examples: `typedef`, `switch`, `enum`. How C arrays are stored in memory and indexed. Strings as arrays of `char`. Bounds checking and the idea of memory safety. Data items occupy space; memory safety means that operations on an item affect only the space the item occupies. Why C is unsafe. Space for strings: the idea of a byte array as a string buffer. Review from CS 201: strings in Java vs. dynamic `StringBuilder`. Arrays and strings as pointers. Java references vs. C pointers. A peek at the machine model.
Reading. DIS §2.1-§2.3. Reread §1.5 and look into `str` functions in §2.6.
Lab: disassembler. Simple disassembler for Facile. Involves array lookup, `printf`, `switch`, `enum`, external symbols.

3. Functions and the call stack. Recap on pointers and strings. Using `str` functions to work with strings and `strn` functions to do it safely. CS 201 review: functions, APIs, arguments and argument passing, pass-by-value, pass-by-value-of-reference, automatic memory allocation for local variables, scope and lifetime, stack discipline (LIFO) and the call stack, importance of the stack for recursion. Mutability and ownership for reference (pointer) arguments passed to APIs. The stack and memory safety in C: memory reuse, behavior of uninitialized local variables, dangling references.
Reading. DIS §3.1-§3.2. Reread §1.4.
Lab: kvlog: Label Log: a log-based dictionary to map label names. Involves buffering, ownership, working with strings and arrays, `&`.

4. Dynamic data structures: the heap. Heap allocation and deallocation in Java vs. C. CS 201 review: Java new() and automatic memory management in Java. Why C programmers have to use `sizeof`, type casting, and `free`. Common heap mistakes and the heap "contract". Memory safety errors: faults, undefined behavior, memory leaks. Using *valgrind* tool to detect memory safety errors.
Reading. DIS §2.4-§2.7, §3.3.
Lab: pointers. Using C pointers and `malloc`; strings and linked lists in the heap with recursion. Project *KV* out.

5. The C programming environment. More about program build, launch, and exit. Inside your program file. What the debugger knows: the symbol table. What the operating system knows: program sections and entry points. The truth about `main()`. Program arguments: `argc` and `argv` (vs. multi-dimensional arrays). How to read program input safely.
Reading. DIS §2.8-§2.9, (Save §2.9.3-§2.9.4 about pointer arithmetic for later.)
Lab: ring. Encoder/decoder ring.

2 Data: Binary Representation

One reason to study C is that it enables you to work with memory below the type system. Learning objectives for this unit: understand the abstraction of raw memory space as a resource for storing data of all types ("just bits"); how various data types are represented compactly in memory; limitations and tradeoffs of these representations; how to operate on data at the bit level and why it is useful. This unit also explores how

complex data types are laid out in memory and how code finds what it is looking for.

6. Integers and arithmetic. Number bases: binary and hexadecimal. Unsigned integers. Negative numbers with two's complement. Integer widths and range. Modular addition and overflow. Characters as small integers: ASCII. integer byte ordering.
Reading. DIS §4.1-§4.5.
Lab: binhex. Converting between binary and hexadecimal.

7. Working with bits and boolean operators. Boolean operators and boolean algebra. Bitwise boolean operators in C. Using bitwise operators to manipulate numbers and addresses: alignment, equality, subtraction with negation and addition, using shifts to multiply and divide.
Reading. DIS §4.6.
Lab: bits. Working with bits and boolean operations.

8. Data layout and binary encodings. Array addressing, pointer arithmetic, and `void*`. Blocks and offsets. Padding to preserve alignment. Inside `malloc`. Compact encodings with bit fields. Using shifts and masks to select bits. Example: compact sets as bit arrays. Floating point representation and the problem of precision. Representing media: images.
Reading. DIS §2.9.3-§2.9.4, §4.7-§4.8.
Lab: offsets Struct layout and pointer arithmetic. Project *alloc* out.

9. Topic review: Systems programming in C. Example: representing playing cards. Type casts and type safety revisited. Topic: "endianness", and the need for byte swapping in networks. Discussion of linking and how the linker constructs the parts (sections) of a binary program file.

10. Midterm 1. Friday, September 27. Covers 1-8.

3 Code: Instructions for a CPU Core

The purpose of this unit is to demystify machine-level computation and give you some familiarity with machine code at the assembler (ASM) level, and how it runs. The details are machine-specific and assembler-specific. We introduce concepts using facileASM—a simple abstract machine with certain messy details out of scope. Details from Intel x86-64 ("x64") illustrate and extend the concepts. Learning objectives: (1) understand digital logic as the building blocks of computer hardware; (2) establish ASM concepts and terms that are common across CPU architectures; (3) understand trends in CPU technology and their impact on programming for performance; (4) demonstrate how compiler-generated code implements programming language features (conditionals, loops, functions) at the machine level; (5) develop confidence to "dive into" ASM code when called to it.

11. Computation in hardware and software. The parts of a computer. A peek at digital logic: gates and circuits. Clocked logic and cycle time. Overview of Instruction Set Architecture (ISA) terms and concepts: instructions, opcodes, operands, registers. Memory accesses in the instruction stream: RISC vs. CISC architectures. Performance: latency and throughput, illustrated with multicore. Big-O analysis revisited.
Reading. Review Facile §3. DIS §5.1-§5.5, §5.9.
Note: DIS §5.4 and §5.9.1 are optional and out of scope for this course.
Lab: images. Use file I/O and boolean operations to process image files.

12. The CPU core. Instructions and operands. Registers. The program counter (PC) or instruction pointer (IP) register. The stack pointer (SP) register. Push and pop instructions and stack alignment. Typeless execution: instruction suffixes for data unit size, operand variants for signed vs. unsigned arithmetic. Indirect addressing: addressing local variables and heap blocks. Argument passing. What the compiler knows.
Reading. DIS §5.6. §6.
Lab: mystery. Using gdb to disassemble x64 instructions.

13. Control structures. Branches and conditional branches. Analogy to history: goto statements, labels, and spaghetti code. How to implement basic control structures at the machine level: conditionals, loops.

switch, procedure call and return. Indirect jumps, jump tables, and C function pointers.
Reading. DIS §7.4. Optional: DIS §7/1-7.3.
Lab: asm. Loops and conditionals in facileASM.

14. Topic review: ISA discussion. ISA examples: x64 and ARM. Registers and their usage, basic instruction opcodes, operand variants, data addressing, condition codes. Memory addressing with the infamous x64 `load effective address` instruction.
Reading. Optional: material drawn from DIS §7 (x86-64) and §9 (ARM).
Lab. No lab. Friday before Fall Break: October 11.

15. Procedures and recursion. ISA features for procedure/function call and return. The return address on the stack. Passing arguments and return value. Modularity reconsidered: calling conventions, register spills, and register usage conventions. Recursion example for x86-64 from DIS §7.6. In scope: stack instructions `pushq` and `popq` and their use for register spills; `callq`, and `retq` and their use for function call/return; effects of these instructions on the stack; accessing arguments and local variables within a procedure; recognize situations that call for register spills.
Reading. DIS §7.5 and §7.6 (x86-64).
Lab: interpret. Function pointers and 8-bit stack machine.

4 Software on the Machine

This module focuses on the system environment for your programs, and its impact on performance and security.

16. Stack buffer overflow and security. Vulnerabilities and Remote Code Execution (RCE) attacks. Overflowing a local variable. Smashing the return address. Code injection. How an attacker uses knowledge of address space layout to exploit an overflow vulnerability and mount an RCE attack. Defenses: no-execute (NX) data segments, address space layout randomization (ASLR). Cyberwar examples: the story of EternalBlue.
Reading. DIS §7.10 (x86-64).
Lab: binary. Linking or DIS Guessing Game (TBD). Project *attack* out.

17. Storage. The memory/storage hierarchy or "pyramid". Latency and throughput. The von Neumann bottleneck. I/O operations and DMA. Blocks and block I/O. Volatile vs. non-volatile storage. The problem of crash recovery. SSDs vs. spinning disks. Overhead and the I/O bottleneck. The impact of transfer size on throughput. Memory as a cache over storage. Spatial locality and temporal locality.
Reading. DIS §11.1-§11.3.
Lab: locality. Loops in a 3D array and their effect on performance. See DIS §11.3.1.

18. Caching. Locality revisited. Caching: hit or miss. Hit/miss ratio and performance. Mechanics: blocks, lines, and slots. Cold/compulsory misses and capacity misses in a "perfect" (fully associative) cache. The impact of spatial locality on cold misses. The impact of temporal locality on capacity misses. Keeping track of the cache contents. Basics of cache mapping: Block number, slot index, byte offset. Hash tables in software and hardware. Direct-mapped caches. Conflict misses in a direct-mapped cache. Cache analysis and cachegrind.
Reading. DIS §11.4-§11.6.
Lab: casim. Direct-mapped cache simulator. See also *cachechecker* for tag/index/block combinations.

19. The process and the kernel. The kernel: protected CPU mode and space. Booting the kernel. Getting into it: system calls and faults as examples of Exceptional Control Flow (ECF), the foundation of modern operating systems. Processes and the process ID. Process blocking and context switch. Process creation: parent and child. Where `argv` comes from. The abstraction of virtual memory.
Reading. DIS §13.1-§13.4. (We gloss over `fork/exec`.)
Lab: files. File I/O (TBD).

20. Virtualization. How to share a machine transparently among many processes? Sharing the processors: timeslicing. Sharing machine memory: pages, page translation, and page protections. Translation at hardware speeds: the MMU. Mapping again: page tables to map virtual addresses to physical addresses. Protection faults and segmentation faults revisited. Role of the kernel in handling faults.
Reading. DIS §13.3-§13.5.
Lab: vasim. Virtual page simulator. Project *KVcache* out.

21. Topic Review: the Machine. OS topics: Unix/Linux `fork/exec/exit/wait`. Topics for linking and address spaces: relocation, position-independent code (PIC), dynamic linking for shared libraries (DLLs).

22. Midterm exam 2. Friday, November 8. Covers lectures in range 1-20, with emphasis on 11-20.

5 Parallelism and Concurrency

23. Parallelism and coordination. Pipelining: doing the laundry. Pipelining in the CPU. Pipelining with Inter-process communication (IPC). The bounded buffer abstraction and its use for pipes and sockets. Logical concurrency (timeslicing) and physical concurrency (e.g., multicore) revisited. Coordination needs for pipes and sockets. CPU utilization as a metric of efficiency.
Reading. Parallelism in the CPU: DIS §5.7 and §5.9. IPC: DIS §13.4. (Skim §13.4.1; we won't discuss signals).
Lab: amdahl. Performance for parallel tasks. [TBD]

24. Threads. Processes vs. threads. Posix threads (pthreads) API. Why use threads? What metrics show the benefit? Three application template examples or design patterns. (1) Parallel computing: speedup and Amdahl's Law. (2) Servers: request throughput and response time. (3) Graphical user interface: events with background processing. Coordination in shared memory. Why threads run with a non-deterministic schedule and interleaving. The problem of data races.
Reading. DIS §14.1-§14.2.
Lab: threads. Thread intro.

25. Synchronization. Concurrency control with mutexes and conditions in the pthreads API. Example: the soda machine as a bounded buffer.
Reading. DIS §14.3. (Skim 14.3.2: we won't discuss semaphores.)
Lab: lock. Mutexes for safe array access.

26. Using synchronization. Barrier and other examples. Thundering herds. Deadlock. The Dining Philosophers. Starvation.
Reading. DIS §14.3.3.
Lab: wait. Using conditions to wait for events.

27. Topic Review: Concurrency. Threads backlash: reactive programming and the need for state.

28. Wrapup: LDOC.