Princeton University
COS 217: Introduction to Programming Systems
Spring 2018 Final Exam Preparation

The exam is a three-hour, closed-book, closed-notes, closed-handouts exam. The exam is cumulative, but emphasizes second-half material. During the exam you may not use a "cheat-sheet." During the exam you may not use computers, calculators, or other electronic devices.

Topics

*You are responsible for all material covered in lectures, precepts, assignments, and required readings. This is a non-exhaustive list of topics that were covered. Topics that were not covered on the midterm exam are in **boldface**.*

1. Number Systems

    Binary, octal, and hexadecimal
    Finite unsigned integers, operations, and overflow
    Finite two's complement signed integers, operations, and overflow
    Floating-point numbers

2. C Programming

    From source to executable: preprocess, compile, assemble, link
    Program structure: multi-file programs with header files
    Process memory layout: text, stack, heap, rodata, **data**, **bss** sections
    Primitive data types
    Variable declarations and definitions
    Variable scope, linkage, and duration/extent
    Constants: `#define`, constant variables, enumerations
    Operators
    Statements
    Function declarations and definitions
    Pointers and arrays
        Call-by-reference, arrays as parameters, strings
        Command-line arguments
    Input/output facilities for standard streams **and files**, and for text **and binary data**
        `getchar()`, `fgetc()`, `putchar()`, `fputc()`, `gets()`, `fgets()`, `puts()`, `fputs()`, `scanf()`, `fscanf()`, `printf()`, `fprintf()`, **`fopen()`**, **`fclose()`**, **`fwrite()`**, **`putc()`**
    Structures
    Dynamic memory management
        `malloc()`, `calloc()`, `realloc()`, `free()`
        Common errors: dereference of dangling pointer, memory leak, double free
    Abstract objects
    Abstract data types; opaque pointers
    Generic data structures and functions
        Void pointers
        Function pointers and function callbacks
    Parameterized macros and their dangers (see King Section 14.3)

3. Programming-in-the-Large

- Modules and interfaces
    - Abstract data types and ADT design in C
    - Heuristics for effective modules: encapsulates data, manages resources, is consistent, has a minimal interface, detects and handles/reports errors, establishes contracts, has strong cohesion, has weak coupling
- Program and programming style
    - Bottom-up design, top-down design, least-risk design
- Building
    - Motivation for make, make fundamentals, non-file targets, macros, implicit rules
- Testing
    - External testing with scripts
    - Internal testing with assertions: validating parameters and return values, checking invariants, checking array subscripts, checking function values
    - Unit testing with scaffolds and stubs
    - Test coverage: statement, path, boundary
- Debugging
    - General heuristics for debugging: understand error messages, think before writing, look for familiar bugs, divide and conquer, add more internal tests, display output, use a debugger, focus on recent changes
    - Heuristics for debugging dynamic memory management: look for common DMM bugs, diagnose seg faults using gdb, manually inspect `malloc()` calls, comment-out `free()` calls, use Meminfo, use Valgrind

- **Performance improvement**
    - **Should you optimize?**
        - **Performance improvement pros and cons, do timing studies**
    - **What should you optimize?**
        - **Use a performance profiler**
    - **Optimization techniques**
        - **Use a better algorithm or data structure, avoid repeated computation, inline function calls, unroll loops, use a lower-level language**

4. Under the Hood: Language Levels Tour

- **Language levels**
    - **High-level vs. assembly vs. machine language**
- **Computer architecture**
    - **The Von Neumann architecture**
        - **RAM**
        - **CPU: control unit, ALU, registers**
    - **Big-endian vs. little-endian byte order**
    - **CISC vs. RISC architectures**
- **x86-64 computer architecture**
    - **General purpose registers: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15**
    - **Sub-registers: RAX, EAX, AX, AH, AL, …**
    - **Special purpose registers: EFLAGS, RIP**
- **x86-64 assembly language**
    - **Instructions: directives and mnemonics**
        - **Defining data**
        - **Transferring data**
        - **Performing arithmetic**
        - **Manipulating bits**
    - **Instruction operands**

- Immediate vs. register vs. memory
- **Control flow**
  - **Unconditional jumps**
  - **Conditional jumps**
    - **Condition code bits in EFLAGS register**
      - **Set by cmp instruction (and other instructions)**
      - **Examined by conditional jump instructions**
    - **Conditional jumps with signed data**
    - **Conditional jumps with unsigned data**
- **Data structures**
  - **Arrays**
  - **Full form of memory operands**
    - **Direct, indirect, base+displacement, indexed, scaled-indexed addressing**
  - **Structures**
  - **Padding**
- **Local variables**
  - **The stack section and the RSP register**
- **x86-64 function call conventions**
  - **Calling and returning**
    - **The `call` and `ret` instructions**
  - **Passing arguments**
    - **Registers: RDI, RSI, RDX, RCX, R8, R9**
  - **Returning a value**
    - **Register: RAX**
  - **Optimization**
    - **Caller-saved regs: RDI, RSI, RDX, RCX, R8, R9, RAX, R10, R11**
      - **Used for parameters and scratch**
      - **Caller must save, if it wants**
    - **Callee-saved regs: RBX, RBP, R12, R13, R14, R15**
      - **Used for local variables**
      - **Callee must save**

x86-64 machine language
- **Instruction format: prefix, opcode, modR/M, SIB, displacement, immediate fields**
- **Machine language after assembly**
  - **Data section, rodata section, bss section, text section, relocation records**
- **Machine language after linking**
  - **Resolution: fetch library code**
  - **Relocation: use relocation records to patch code**
  - **Output: data section, rodata section, bss section, text section**

5. Under the Hood: Service Levels Tour

Exceptions and processes
- **Exceptions**
  - **Synchronous vs. asynchronous**
  - **Interrupts, traps, faults, and aborts**
- **Traps and system-level functions in x86-64**
- **The process abstraction**
- **The illusion of private address space**
  - **Reality: virtual memory via page faults**
- **The illusion of private control flow**
  - **Reality: context switches during exception handling**

Storage management
- **Locality of reference and caching**

Typical storage hierarchy: registers vs. cache vs. memory vs. local secondary storage vs. remote secondary storage
Virtual memory
Implementation of virtual memory
Virtual addresses vs. physical addresses
Page tables, page faults
Benefits of virtual memory
Dynamic memory management (DMM)
The need for DMM
DMM using the heap section
The `brk()` and `sbrk()` system-level functions
Internal and external fragmentation
Minimal, pad, free-list, doubly-linked free list, bins implementations
DMM using virtual memory
The `mmap()` and `munmap()` system-level functions
Process management
Creating processes
The `getpid()` and `fork()` system-level function
Waiting for (reaping, harvesting) processes
The `wait()` system-level function
Executing new programs
The `execvp()` system-level functions
The `system()` function
I/O management
The file abstraction
Linux I/O
File descriptors, file descriptor tables, file tables
The `creat()`, `open()`, `close()`, `read()`, `write()` system-level functions
Standard C I/O
Buffering
Implementing standard C I/O using Linux I/O
Redirecting standard files
The `dup()` and `dup2()` system-level functions
Pipes
The `pipe()` system-level function
Signals and alarms
Sending signals
Via keystrokes, the kill command, and the `raise()` and `kill()` functions
Handling signals
The `signal()` function
The `SIG_IGN` and `SIG_DFL` arguments to `signal()`
Alarms
The `alarm()` function

6. Applications

De-commenting
Lexical analysis using finite state automata
String manipulation
Symbol tables, linked lists, hash tables
Dynamically expanding arrays
High-precision addition
Buffer overrun attacks
Heap management
Linux shells

7. Tools: The Linux/GNU programming environment

    Linux
    `bash`
    `emacs`
    `gcc`
    `gdb` for C
    `make`
    oProfile
    `gdb` for assembly language
    `objdump`

Readings

As specified by the course "Schedule" Web page.

Required:

- *C Programming* (King): 1, 2, 3, 4, 5, 6, 7, 8, 9, **10**, 11, 12, 13, 14, 15, 16, 17, **18**, 19, 20.1, 22, 24.1, **24.2, 24.3**
- *Computer Systems* (Bryant & O'Hallaron): 1, **3 (OK to skip 3.11)**, **8.1-5**, **9**
- ***The C Programming Language* (Kernighan & Ritchie) 8.7**

Recommended:

- *Computer Systems* (Bryant & O'Hallaron): 2, **5.1-5**, **6**, **7**, **10**
- *The Practice of Programming* (Kernighan & Pike): 1, 2, 4, 5, 6, **7**, 8
- *Unix Tutorial for Beginners* (website)
- *GNU Emacs Tutorial* (website)
- *Linux Pocket Guide* (Barrett)
- *Deterministic Finite Automaton* Wikipedia article (website)
- *GNU GDB Tutorial* (website)
- *GNU Make Tutorial* (website)

Recommended, for reference only:

- ***OProfile Manual* (website)**
- ***Intel 64 and IA-32 Architectures Software Developer's Manual: Vol 1: Basic Architecture***
- ***Intel 64 and IA-32 Architectures Software Developer's Manual: Vol 2: Instruction Set Reference***
- ***Intel 64 and IA-32 Architectures Software Developer's Manual: Vol 3: System Prog. Guide***
- ***Intel 64 and IA-32 Architectures Optimization Reference Manual***
- ***Using As***