Princeton University
COS 217: Introduction to Programming Systems
Fall 2015 Final Exam Preparation

The exam is a three-hour, closed-book, closed-notes, closed-handouts exam. You may use a one-page (two-sided) summary sheet that you (and only you) have created. The exam is cumulative, but emphasizes second-half material. No laptops, calculators, or other electronic devices are permitted.

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 covered after the midterm exam are in **boldface**.*

1. Number Systems

    The binary, octal, and hexadecimal number systems
    Finite representation of unsigned integers
        Operations on unsigned integers
    Finite representation of signed integers
        Signed magnitude, ones' complement, two's complement
        Operations on signed integers
    Finite representation of rational numbers

2. C Programming

    The program preparation process: preprocess, compile, assemble, link
    Program structure: multi-file programs using header files
    Process memory layout: text, stack, heap, rodata, data, bss sections
    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**
    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

- Testing
  - External testing taxonomy: statement, path, boundary, stress
  - Internal testing techniques: validate parameters, check invariants, check function return values, change code temporarily, leave testing code intact
  - General testing strategies: automate the tests, test incrementally, let debugging drive testing (fault injection)
- Building
  - Separate independent paths before link
  - Motivation for make, make fundamentals, macros, abbreviations, pattern rules
- Program and programming style
  - Bottom-up design, top-down design, least-risk design
- 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`
- Data Structures and algorithms
  - Linked lists
  - Hash tables: hashing algorithms, defensive copies, key ownership
- Modularity
  - History of modularity: non-modular, structured, abstract object, abstract data type programming
  - Module qualities: encapsulates data, is consistent, has a minimal interface, detects and handles/reports errors, establishes contracts, has strong cohesion, has weak coupling
- **Performance improvement**
  - **When to improve performance**
  - **Improving execution (time) efficiency: do timing studies, identify hot spots, use a better algorithm, enable compiler speed optimization, tune the code**
  - **Improving memory (space) efficiency: use a smaller data type, compute instead of storing, enable compiler space optimization**

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**
  - **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, symbol table, relocation records**
- **Machine language after linking**
  - **Resolution: Fetch library code**
  - **Relocation: Use relocation records and symbol table 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
Free-list, doubly-linked free list, bin 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
Standard C I/O
Buffering
Unix I/O
File descriptors, file descriptor tables, file tables
The `creat()`, `open()`, `close()`, `read()`, `write()` system-level functions
Implementing standard C I/O using Unix 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
Unix/Linux shells

7. Tools: The Unix/GNU programming environment

   Unix/Linux
   `bash`
   `emacs`
   `gcc`
   `gdb` for C
   `make`
   `gprof`
   `gdb` for assembly language
   `objdump`

Readings

*As specified by the course "Schedule" Web page. Readings that were assigned after the midterm exam are in **boldface**.*

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**
   Computer Systems (Bryant & O'Hallaron): 1, **3 (OK to skip 3.11), 8.1-5, 9**
   ***Communications of the ACM* "Detection and Prevention of Stack Buffer Overflow Attacks"**
   ***The C Programming Language* (Kernighan & Ritchie) 8.7**

Recommended:

   Computer Systems (Bryant & O'Hallaron): 2, **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)
   GNU GDB Tutorial (website)
   GNU Make Tutorial (website)
   GNU Gprof Tutorial (website)
   "Security as a Class of Interface Guarantee"