Java and C (part II) + Course Wrap-Up
CSE 351 Summer 2020

Instructor: Porter Jones
Teaching Assistants: Amy Xu, Callum Walker, Sam Wolfson, Tim Mandzyuk

https://xkcd.com/1760/
Administrivia

- Questions doc: https://tinyurl.com/CSE351-8-21

- Can still do hw19 (it’s optional/not for credit)
- hw23 due Monday (8/24) – 10:30am
  - Cover most of the material today, a few more things Friday

- Lab 5 and Unit Summary 3 due tonight!(Friday 8/21)
  - Cutoff is tomorrow, Saturday 8/22 @11:59pm (only one late day can be used!)

* Saturday & Sunday office hours*
Course Evaluation Reminder Meme

- Reminder to please fill out your course evaluations!! (you should have received a couple emails with a link to the eval)
Virtual Machine Model

High-Level Language Program (e.g. Java, C)

Bytecode compiler (e.g. javac Foo.java)

Virtual Machine Language (e.g. Java bytecodes)

Virtual machine (interpreter) (e.g. java Foo)

Native Machine Language (e.g. x86, ARM, MIPS)

Ahead-of-time compiler

JIT compiler
Java Bytecode

- Like assembly code for JVM, but works on all JVMs
  - Hardware-independent!
- Typed (unlike x86 assembly)
- Strong JVM protections
JVM Operand Stack

Bytecode:

```
  iload 1    // push 1st argument from table onto stack
  iload 2    // push 2nd argument from table onto stack
  iadd       // pop top 2 elements from stack, add together, and
            // push result back onto stack
  istore 3   // pop result and put it into third slot in table
```

No registers or stack locations! All operations use operand stack

Compiled to (IA32) x86:

```
mov 8(%ebp), %eax
mov 12(%ebp), %edx
add %edx, %eax
mov %eax, -8(%ebp)
```
A Simple Java Method

Method java.lang.String getEmployeeName() {

0  aload 0        // "this" object is stored at 0 in the var table

1  getfield #5 <Field java.lang.String name> 
   // getfield instruction has a 3-byte encoding 
   // Pop an element from top of stack, retrieve its 
   // specified instance field and push it onto stack 
   // "name" field is the fifth field of the object

4  areturn        // Returns object at top of stack

Byte number: 0 1 4

aload_0 | getfield | 00 | 05 | areturn

As stored in the .class file: 2AB4 00 05 B0

Class File Format

- Every class in Java source code is compiled to its own class file

- 10 sections in the Java class file structure:
  - **Magic number**: 0xCAFEBABE (legible hex from James Gosling – Java’s inventor)
  - **Version of class file format**: The minor and major versions of the class file
  - **Constant pool**: Set of constant values for the class
  - **Access flags**: For example whether the class is abstract, static, final, etc.
  - **This class**: The name of the current class
  - **Super class**: The name of the super class
  - **Interfaces**: Any interfaces in the class
  - **Fields**: Any fields in the class
  - **Methods**: Any methods in the class
  - **Attributes**: Any attributes of the class (for example, name of source file, etc.)

- A `.jar` file collects together all of the class files needed for the program, plus any additional resources (e.g. images)
Disassembled Java Bytecode

Compiled from Employee.java

```java
class Employee extends java.lang.Object {
    public Employee(java.lang.String, int);
    public java.lang.String getEmployeeName();
    public int getEmployeeNumber();
}
```

Method Employee(java.lang.String, int)
```
0  aload_0
1  invokespecial #3 <Method java.lang.Object()>  
4  aload_0
5  aload_1
6  putfield #5 <Field java.lang.String name>
9  aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void storeData(java.lang.String, int)>  
20 return
```

Method java.lang.String getEmployeeName()
```
0  aload_0
1  getfield #5 <Field java.lang.String name>
4  areturn
```

Method int getEmployeeNumber()
```
0  aload_0
1  getfield #4 <Field int idNumber>
4  ireturn
```

Method void storeData(java.lang.String, int)
```
...
Other languages for JVMs

- JVMs run on so many computers that compilers have been built to translate many other languages to Java bytecode:
  - **AspectJ**, an aspect-oriented extension of Java
  - **ColdFusion**, a scripting language compiled to Java
  - **Clojure**, a functional Lisp dialect
  - **Groovy**, a scripting language
  - **JavaFX Script**, a scripting language for web apps
  - **JRuby**, an implementation of Ruby
  - **Jython**, an implementation of Python
  - **Rhino**, an implementation of JavaScript
  - **Scala**, an object-oriented and functional programming language
  - And many others, even including C!

- Originally, JVMs were designed and built for Java (still the major use) but JVMs are also viewed as a safe, GC’ed platform.
Microsoft’s C# and .NET Framework

- C# has similar motivations as Java
  - Virtual machine is called the *Common Language Runtime*
  - *Common Intermediate Language* is the bytecode for C# and other languages in the .NET framework
We made it! 😊

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:
```
get_mpg:
pushq  %rbp
movq   %rsp, %rbp
...
popq   %rbp
ret
```

Machine code:
```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111
```

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

- Windows 10
- OS X Yosemite
Today

- End-to-end Review
  - What happens after you write your source code?
    - How code becomes a program
    - How your computer executes your code

- Victory lap and high-level concepts (key points)
  - More useful for “5 years from now” than “next week’s final”
C: The Low-Level High-Level Language

C is a “hands-off” language that “exposes” more of hardware (especially memory)

- Weakly-typed language that stresses data as bits
  - Anything can be represented with a number!
- Unconstrained pointers can hold address of *anything*
  - And no bounds checking – buffer overflow possible!
- Efficient by leaving everything up to the programmer
- “C is good for two things: being beautiful and creating catastrophic 0days in memory management.”
  ([https://medium.com/message/everything-is-broken-81e5f33a24e1](https://medium.com/message/everything-is-broken-81e5f33a24e1))
C Data Types

- C Primitive types
  - Fixed sizes and alignments
  - Characters (char), Integers (short, int, long), Floating Point (float, double)

- C Data Structures
  - Arrays – contiguous chunks of memory
    - Multidimensional arrays = still one continuous chunk, but row-major
    - Multi-level arrays = array of pointers to other arrays
  - Structs – structured group of variables
    - Struct fields are ordered according to declaration order
    - *Internal fragmentation*: space between members to satisfy member alignment requirements (aligned for each primitive element)
    - *External fragmentation*: space after last member to satisfy overall struct alignment requirement (largest primitive member)
C and Memory

- Using C allowed us to examine how we store and access data in memory
  - Endianness *(only applies to memory)*
    - Is the first byte (lowest address) the least significant (little endian) or most significant (big endian) of your data?
  - Array indices and struct fields result in calculating proper addresses to access

- Consequences of your code:
  - Affects performance (locality)
  - Affects security

- But to understand these effects better, we had to dive deeper...
How Code Becomes a Program

- **text**: C source code
  - **Compiler** (`gcc -Og -S`)
- **text**: Assembly files
  - **Assembler** (`gcc -c` or `as`)
- **binary**: Object files
  - **Linker** (`gcc` or `ld`)
  - **Static libraries**
- **binary**: Executable program
  - **Loader (the OS)**
- **Hardware**
### Instruction Set Architecture

<table>
<thead>
<tr>
<th>Source code</th>
<th>Compiler</th>
<th>Architecture Instruction set</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different applications or algorithms</td>
<td>Perform optimizations, generate instructions</td>
<td>x86-64 CISC</td>
<td>Intel Pentium 4</td>
</tr>
<tr>
<td>Different applications or algorithms</td>
<td>Perform optimizations, generate instructions</td>
<td>x86-64 RISC</td>
<td>Intel Core 2</td>
</tr>
<tr>
<td>Different applications or algorithms</td>
<td>Perform optimizations, generate instructions</td>
<td>ARMv8 (AAArch64/A64)</td>
<td>Intel Core i7</td>
</tr>
<tr>
<td>Different applications or algorithms</td>
<td>Perform optimizations, generate instructions</td>
<td>ARMv8 (AAArch64/A64)</td>
<td>AMD Opteron</td>
</tr>
<tr>
<td>Different applications or algorithms</td>
<td>Perform optimizations, generate instructions</td>
<td>ARMv8 (AAArch64/A64)</td>
<td>AMD Athlon</td>
</tr>
<tr>
<td>Different applications or algorithms</td>
<td>Perform optimizations, generate instructions</td>
<td>ARMv8 (AAArch64/A64)</td>
<td>ARM Cortex-A53</td>
</tr>
<tr>
<td>Different applications or algorithms</td>
<td>Perform optimizations, generate instructions</td>
<td>ARMv8 (AAArch64/A64)</td>
<td>Apple A7</td>
</tr>
</tbody>
</table>

**C Language**
- Program A
- Program B
- Your program

**Compilers**
- GCC
- Clang

**Architectures**
- x86-64
- ARMv8 (AArch64/A64)
- Apple A7

**Hardware**
- Intel Pentium 4
- Intel Core 2
- Intel Core i7
- AMD Opteron
- AMD Athlon
- ARM Cortex-A53
- Apple A7
Assembly Programmer’s View

- **Programmer-visible state**
  - **PC**: the Program Counter (\%rip in x86-64)
    - Address of next instruction
  - Named registers
    - Together in “register file”
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte-addressable array
  - Huge *virtual* address space
  - *Private, all to yourself...*
Program’s View

- CPU
  - %rip
  - Registers
  - Condition Codes

- Memory
  - Stack
  - Dynamic Data (Heap)
  - Static Data
  - Literals
  - Instructions

High addresses: $2^{N-1}$

- Local variables; procedure context
- Variables allocated with `new` or `malloc`
- Static variables (global variables in C)
- Large constants (e.g., “example”)

Low addresses: 0
Program’s View

- **Instructions**
  - Data movement
    - `mov`, `movz`, `movz`
    - `push`, `pop`
  - Arithmetic
    - `add`, `sub`, `imul`
  - Control flow
    - `cmp`, `test`
    - `jmp`, `je`, `jgt`, ...
    - `call`, `ret`

- **Operand types**
  - Literal: $8$
  - Register: %rdi, %al
  - Memory: \( D(Rb,Ri,S) = D + Rb + Ri \times S \)
    - `lea`: *not a memory access!*

---

Diagram:
- Memory
- Stack
- Dynamic Data (Heap)
- Static Data
- Literals
- Instructions

- Local variables; procedure context
- Variables allocated with `new` or `malloc`
- *Static* variables (global variables in C)
- Large constants (e.g., “example”)
Program’s View

- **Procedures**
  - Essential abstraction
  - Recursion...

- **Stack discipline**
  - Stack frame per call
  - Local variables

- **Calling convention**
  - How to pass arguments
    - Diane’s Silk Dress Costs $89
  - How to return data
  - Return address
  - Caller-saved / callee-saved registers

---

Memory

Stack

Dynamic Data (Heap)

Static Data

Literals

Instructions

Local variables; procedure context

Variables allocated with *new* or *malloc*

*static* variables (global variables in C)

Large constants (e.g., “example”)

High addresses

2^{N-1}

Low addresses 0

Large constants (e.g., “example”)
Program’s View

- **Heap data**
  - Variable size
  - Variable lifetime

- **Allocator**
  - Balance *throughput* and *memory utilization*
  - Data structures to keep track of free blocks

- **Garbage collection**
  - Must always free memory
  - Garbage collectors help by finding anything *reachable*
  - Failing to free results in *memory leaks*
But remember... it’s all an illusion! 😮

- **Context switches**
  - Don’t really have CPU to yourself

- **Virtual Memory**
  - Don’t really have $2^{64}$ bytes of memory all to yourself
  - Allows for *indirection* (remap physical pages, sharing...)

---

- **CPU**
  - `%rip`
  - Registers
  - Condition Codes
  
- **Memory**
  - **Stack**
  - **Dynamic Data (Heap)**
  - **Static Data**
  - **Literals**
  - **Instructions**

- **Low addresses** 0
  - **High addresses** $2^{N-1}$

- local variables; procedure context
- variables allocated with *new* or *malloc*
- *static* variables (global variables in C)
- Large constants (e.g., “example”)
But remember... it’s all an illusion!

- fork
  - Creates copy of the process
- execv
  - Replace with new program
- wait
  - Wait for child to die (to reap it and prevent zombies)
Virtual Memory

Address Translation

- Every memory access must first be converted from virtual to physical
- *Indirection*: just change the address mapping when switching processes
- Luckily, TLB (and page size) makes it pretty fast
But Memory is Also a Lie! 😐

- **Illusion** of one flat array of bytes
  - But *caches* invisibly make accesses to physical addresses faster!

- **Caches**
  - **Associativity** tradeoff with miss rate and access time
  - **Block size** tradeoff with spatial and temporal locality
  - **Cache size** tradeoff with miss rate and cost
Memory Hierarchy

- **Registers**: <1 ns
- **On-chip L1 cache (SRAM)**: 1 ns
- **Off-chip L2 cache (SRAM)**: 5-10 ns
- **Main memory (DRAM)**: 100 ns
- **Local secondary storage (local disks)**: 150,000 ns
- **Remote secondary storage (distributed file systems, web servers)**: 10,000,000 ns (10 ms)

**Performance Characteristics**
- **Smaller, faster, costlier per byte**
  - registers: <1 ns
  - on-chip L1 cache (SRAM): 1 ns
  - off-chip L2 cache (SRAM): 5-10 ns
- **Larger, slower, cheaper per byte**
  - main memory (DRAM): 100 ns
  - SSD: 150,000 ns
  - Disk: 10,000,000 ns (10 ms)
  - Remote secondary storage (distributed file systems, web servers): 1-150 ms

**Access Times**
- **Registered**: 1-2 min
- **On-chip L1 cache**: 1-2 min
- **Off-chip L2 cache**: 15-30 min
- **Main memory**: 31 days
- **Local secondary storage**: 66 months = 5.5 years
- **Remote secondary storage**: 1-15 years

**Cost Comparison**
- **Remote secondary storage**: distributed file systems, web servers
  - 1-15 years

**Storage Types**
- **Local**: on-chip L1 cache (SRAM), main memory (DRAM), local secondary storage (local disks)
- **Remote**: remote secondary storage (distributed file systems, web servers)
Review of Course Themes

- Review course goals
  - They should make much more sense now!
Big Theme: Abstractions and Interfaces

- Computing is about abstractions
  - (but we can’t forget reality)
- What are the abstractions that we use?
- What do you need to know about them?
  - When do they break down and you have to peek under the hood?
  - What bugs can they cause and how do you find them?
- How does the hardware relate to the software?
  - Become a better programmer and begin to understand the important concepts that have evolved in building ever more complex computer systems
Little Theme 1: Representation

- All digital systems represent everything as 0s and 1s
  - The 0 and 1 are really two different voltage ranges in the wires
  - Or magnetic positions on a disc, or hole depths on a DVD, or even DNA...

- “Everything” includes:
  - Numbers – integers and floating point
  - Characters – the building blocks of strings
  - Instructions – the directives to the CPU that make up a program
  - Pointers – addresses of data objects stored away in memory

- Encodings are stored throughout a computer system
  - In registers, caches, memories, disks, etc.

- They all need addresses (a way to locate)
  - Find a new place to put a new item
  - Reclaim the place in memory when data no longer needed
Little Theme 2: Translation

- There is a big gap between how we think about programs and data and the 0s and 1s of computers
  - Need languages to describe what we mean
  - These languages need to be translated one level at a time

- We know Java as a programming language
  - Have to work our way down to the 0s and 1s of computers
  - Try not to lose anything in translation!
  - We encountered C language, assembly language, and machine code (for the x86 family of CPU architectures)
Little Theme 3: Control Flow

- How do computers orchestrate everything they are doing?

- **Within one program:**
  - How do we implement if/else, loops, switches?
  - What do we have to keep track of when we call a procedure, and then another, and then another, and so on?
  - How do we know what to do upon “return”?

- **Across programs and operating systems:**
  - Multiple user programs
  - Operating system has to orchestrate them all
    - Each gets a share of computing cycles
    - They may need to share system resources (memory, I/O, disks)
  - Yielding and taking control of the processor
    - Voluntary or “by force”?
Course Perspective

- CSE351 will make you a better programmer
  - Purpose is to show how software really works
  - Understanding the underlying system makes you more effective
    - Better debugging
    - Better basis for evaluating performance
    - How multiple activities work in concert (e.g., OS and user programs)
  - Not just a course for hardware enthusiasts!
    - What every CSE major needs to know (plus many more details)
    - See many patterns that come up over and over in computing (like caching)
    - “Stuff everybody learns and uses and forgets not knowing”

- CSE351 presents a world-view that will empower you
  - The intellectual and software tools to understand the trillions+ of 1s and 0s that are “flying around” when your program runs
Topics: What’s Next?

- Even if CSE 351 wasn’t for you, I would encourage you to explore topics that build on its material!
  - I know plenty of people who hated 351 but ended up loving a future topic
- Here are a few topics that build on the material we talked about in this course.
  - UW has many courses that align with these topics, other universities might too!
  - You can also research these on your own, plenty of information online!
- Staying near the hardware/software interface:
  - Digital Design – basic hardware design and circuit logic
  - Computer Architecture – hardware design of CPUs
  - Embedded Systems – software design for microcontrollers
- Systems software
  - Programming Languages and Compilers
  - Data Structures and Parallelism
  - General Systems Programming – building well-structured systems in C/C++
  - Operating Systems
  - Networks
  - Security
Thanks for a great quarter!

- Huge thanks to your awesome TAs!

- Don’t be a stranger!
  - Feel free to send us emails with questions about anything in the future!