The Hardware/Software Interface
CSE351 Spring 2015
Lecture 26

Instructor:
Katelin Bailey

Teaching Assistants:
Roadmap

C:

```c
#include <stdlib.h>

#define CAR_SIZE sizeof(car)

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

Java:

```java
import java.util.*;

public class Car {
    public int miles;
    public int gals;

    public Car() {
        this.miles = 100;
        this.gals = 17;
    }

    public float getMPG() {
        return mpg;
    }
}
```

Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

- OS:
  - Windows 8
  - Mac

- Memory, data, & addressing
- Integers & floats
- Machine code & C
- x86 assembly
- Procedures & stacks
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C
Implementing Programming Languages

- Many choices in how to implement programming models
- We’ve talked about compilation, can also interpret
  - Execute line by line in original source code
  - Simpler/no compiler – less translation
  - More transparent to debug – less translation
  - Easier to run on different architectures – runs in a simulated environment that exists only inside the interpreter process
  - Slower and harder to optimize
  - All errors at run time

- Interpreting languages has a long history
  - Lisp, an early programming language, was interpreted

- Interpreters are still in common use:
  - Python, Javascript, Ruby, Matlab, PHP, Perl, …
Interpreted vs. Compiled in practice

- Really a continuum, a choice to be made
  - More or less work done by interpreter/compiler

- Java programs are usually run by a virtual machine
  - JVMs interpret an intermediate language called Java bytecode
  - Many JVMs compile bytecode to native machine code
    - just-in-time (JIT) compilation
  - Java is sometimes compiled ahead of time (AOT) like C
Virtual Machine Model

High-Level Language Program

- Bytecode compiler
- Virtual machine (interpreter)

Virtual Machine Language

- JIT compiler

Native Machine Language

- Ahead-of-time compiler

 compile time
 run time
Java bytecode

- like assembly code for JVM, but works on all JVMs: hardware-independent
- typed (unlike ASM)
- strong JVM protections
JVM Operand Stack

machine:

variable table

operand stack

constant pool

No registers or stack locations; all operations use 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

compiled to x86:

mov 8(%ebp), %eax
mov 12(%ebp), %edx
add %edx, %eax
mov %eax, -8(%ebp)

‘i’ stands for integer, ‘a’ for reference, ‘b’ for byte, ‘c’ for char, ‘d’ for double, ...

Holds pointer ‘this’

Other arguments to method

Other local variables

‘i’ stands for integer, ‘a’ for reference, ‘b’ for byte, ‘c’ for char, ‘d’ for double, ...

compiled to x86:
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>   // takes 3 bytes
   // 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

In the .class file: 2A B4 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: 0xCAFEBAEBE (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
Compiled from Employee.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
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!
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
Roadmap

C:

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

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
110000011111101000011111

Computer system:

OS:

Bonus topic: parallelism

Memory, data, & addressing
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
When can we execute things in parallel?

**Parallelism:**
Use extra resources to solve a problem faster

**Concurrency:**
Correctly and efficiently manage access to shared resources

thanks to Dan Grossman for the succinct definitions
What is parallel processing?

• Briefly introduction to key ideas of parallel processing
  • instruction level parallelism
  • data-level parallelism
  • thread-level parallelism
Exploiting Parallelism

• Of the computing problems for which performance is important, many have inherent parallelism:

• computer games
  • Graphics, physics, sound, AI etc. can be done separately
  • Furthermore, there is often parallelism within each of these:
    • Each pixel on the screen’s color can be computed independently
    • Non-contacting objects can be updated/simulated independently
    • Artificial intelligence of non-human entities done independently

• search engine queries
  • Every query is independent
  • Searches are (ehm, pretty much) read-only!!
Instruction-Level Parallelism

```
add %r2 <- %r3, %r4
or %r2 <- %r2, %r4
lw %r6 <- 0(%r4)
addi %r7 <- %r6, 0x5
sub %r8 <- %r8, %r4
```

Dependences?
- RAW - read after write
- WAW - write after write
- WAR - write after read

When can we reorder instructions?

```
add %r2 <- %r3, %r4
or %r5 <- %r2, %r4
lw %r6 <- 0(%r4)
sub %r8 <- %r8, %r4
addi %r7 <- %r6, 0x5
```

When should we reorder instructions?

Superscalar Processors:
- Multiple instructions executing in parallel at *same* stage

Take 352 to learn more.
Data Parallelism

- Consider adding together two arrays:

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++ i) {
        C[i] = A[i] + B[i];
    }
}
```

Operating on one element at a time
Data Parallelism

• Consider adding together two arrays:

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++ i) {
        C[i] = A[i] + B[i];
    }
}
```

Operating on one element at a time
Data Parallelism with SIMD

• Consider adding together two arrays:

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++ i) {
        C[i] = A[i] + B[i];
    }
}
```

Operate on MULTIPLE elements

Single Instruction, Multiple Data (SIMD)
Is it always that easy?

- Not always… a more challenging example:

```c
unsigned sum_array(unsigned *array, int length) {
    int total = 0;
    for (int i = 0; i < length; ++i) {
        total += array[i];
    }
    return total;
}
```

- Is there parallelism here?
- Each loop iteration uses data from previous iteration.
Restructure the code for SIMD...

// one option...
unsigned sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    // chunks of 4 at a time
    for (i = 0 ; i < length & ~0x3 ; i += 4) {
        temp[0] += array[i];
        temp[1] += array[i+1];
        temp[2] += array[i+2];
        temp[3] += array[i+3];
    }
    // add the 4 sub-totals
    return total;
}
What are threads?

- Independent “thread of control” within process
- Like multiple processes within one process, but sharing the same virtual address space.
  - logical control flow
    - program counter
    - stack
  - shared virtual address space
    - all threads in process use same virtual address space
- Lighter-weight than processes
  - faster context switching
  - system can support more threads
Thread-level parallelism: Multicore Processors

- Two (or more) complete processors, fabricated on the same silicon chip
- Execute instructions from two (or more) programs/threads at the same time

IBM Power5
Multicores are everywhere

- **Laptops, desktops, servers**
  - Most any machine from the past few years has at least 2 cores

- **Game consoles:**
  - Xbox 360: 3 PowerPC cores; Xbox One: 8 AMD cores
  - PS3: 9 Cell cores (1 master; 8 special SIMD cores);
    PS4: 8 custom AMD x86-64 cores
  - Wii U: 2 Power cores

- **Smartphones**
  - iPhone 4S, 5: dual-core ARM CPUs
  - Galaxy S II, III, IV: dual-core ARM or Snapdragon
  - …
Why Multicores Now?

• Number of transistors we can put on a chip growing exponentially…
• But performance is no longer growing along with transistor count.
• So let’s use those transistors to add more cores to do more at once…
As programmers, do we care?

- What happens if we run this program on a multicore?

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++i) {
        C[i] = A[i] + B[i];
    }
}
```
What if we want one program to run on multiple processors (cores)?

- We have to explicitly tell the machine exactly how to do this
  - This is called parallel programming or concurrent programming

- There are many parallel/concurrent programming models
  - We will look at a relatively simple one: fork-join parallelism
How does this help performance?

- Parallel speedup measures improvement from parallelization:
  \[
  \text{speedup}(p) = \frac{\text{time for best serial version}}{\text{time for version with } p \text{ processors}}
  \]

- What can we realistically expect?
Reason #1: Amdahl’s Law

- In general, the whole computation is not (easily) parallelizable.
- Serial regions limit the potential parallel speedup.
Reason #1: Amdahl’s Law

- Suppose a program takes 1 unit of time to execute serially.
- A fraction of the program, s, is inherently serial (unparallelizable).

\[
\text{New Execution Time} = \frac{1-s}{p} + s
\]

For example, consider a program that, when executing on one processor, spends 10% of its time in a non-parallelizable region. How much faster will this program run on a 3-processor system?

\[
\text{New Execution Time} = \frac{.9T}{3} + .1T
\]
Reason #2: Overhead

- Forking and joining is not instantaneous
  - Involves communicating between processors
  - May involve calls into the operating system
  - Depends on the implementation

\[
\text{New Execution Time} = \frac{1-s}{p} + s + \text{overhead}(P)
\]
Multicore: what should worry us?

• Concurrency: what if we’re sharing resources, memory, etc.?
• Cache Coherence
  • What if two cores have the same data in their own caches?
    How do we keep those copies in sync?
• Memory Consistency, Ordering, Interleaving, Synchronization…
  • With multiple cores, we can have truly concurrent execution of threads.
    In what order do their memory accesses appear to happen?
    Do the orders seen by different cores/threads agree?
• Concurrency Bugs
  • When it all goes wrong…
  • Hard to reproduce, hard to debug
Summary

- **Multicore: more than one processor on the same chip.**
  - Almost all devices now have multicore processors
  - Results from Moore’s law and power constraint

- **Exploiting multicore requires parallel programming**
  - Automatically extracting parallelism too hard for compiler, in general.
  - But, can have compiler do much of the bookkeeping for us

- **Fork-Join model of parallelism**
  - At parallel region, fork a bunch of threads, do the work in parallel, and then join, continuing with just one thread
  - Expect a speedup of less than P on P processors
    - Amdahl’s Law: speedup limited by serial portion of program
    - Overhead: forking and joining are not free