CMSC 216
Introduction to Computer Systems
Lecture 1
Introduction

Administrivia

- Course home page is at

  http://www.cs.umd.edu/class/summer2011/cmsc216/
- If you don't already have a GLUE account, request one at http://www.oit.umd.edu/new/
- Bring your laptop to discussion section
- Read Chapter 1 of Bryant and O'Hallaron, and Chapter 1 of Reek

Introduction to Computer Systems

- Course objectives
- Expectations
- Course policies
- Discussion sections
- Course projects
- Submit server
- Grades server

Miscellaneous

- Regarding deadline to address grading concerns
  - It will be strictly enforced
  - At the end of the semester we will not address grading concerns for assignments/material already graded
- Email
  - Please follow the guidelines available at:
    - http://www.cs.umd.edu/~nelson/classes/emailingNelson.html

Miscellaneous

- If you are experiencing any problems that affect your performance in this class, please contact us immediately. Usually students wait until the end of the semester when probably nothing could be done
- Work hard from the beginning of the semester in order to avoid the following type of messages at the end of the semester:
  - Is there any extra credit so I can boost my grade?
  - I am .1 from an A; can something be done?
  - I need to pass this class otherwise I:
    - Lose my scholarship
    - Lose my house
    - My parents will not love me
    - ...

Chapter 1, Bryant and O'Hallaron

A Tour of Computer Systems

Storage of Information

- Computers store all data as binary digits, or bits; groups of 8 bits are often called bytes
- How these bits are treated depends on their context
  - the same sequence of bits can be used to represent a character, or an integer, or a floating-point number, or an instruction, or...
  - it's all a matter of interpretation

Instruction-based execution

- Each program on a computer is a sequence of instructions written in machine language
- Processor executes one instruction at a time in a program, then executes the next one in turn
- To study code in this form, it's helpful to use assembly language rather than machine language code

Example assembly program

```
main: mov   #0,sum          ; set sum to 0
      mov   #1,num          ; set num to 1
loop: add   num,sum         ; add num to sum
      add   #1,num          ; add 1 to num
      ble   num,#1000,loop  ; if num <= 1000, go back to 'loop'
      halt                  ; end of program. stop running
```

This is a slightly modified version of the example in Wikipedia's Computer article

- What does this program do?
- Sequence of operations doesn't always go to the next instruction in memory

Computer layout

- Lots of places to store information:
  - CPU registers
  - CPU caches
  - Main memory
  - Hard drives
  - Remote storage
- The farther away from the CPU you go, the longer it takes to access data
- Typical programs have to access data stored on a hard drive, which is quite slow compared to other storage mediums

Caching is important

- Executing a program can mean reading instructions from disk into memory, then moving around data from memory to registers or memory to disk
- Because some devices are much slower (maybe because they're bigger), we can utilize caches to speed up execution time by accessing copies of data
- This can be a major performance gain - properly utilizing caches can increase performance by orders of magnitude

The role of the operating system

- Protect the computer from misuse
- Provide an abstraction for using the hardware so that programs can be written for a variety of different hardware
- Manage the resources to allow for reasonable use by all users and programs on a computer

The UNIX Operating System

- Developed in 1970s at Bell Labs
- Kernel written in C, also developed at the same time
  - C was developed for the purpose of writing UNIX and systems programming
- We will use a variant of UNIX named Linux
  - Do not try working in a Windows environment just because you're more comfortable with it!
  - Other UNIX variants exist, such as Solaris, and the various BSDs (OpenBSD, NetBSD, FreeBSD, OSX)

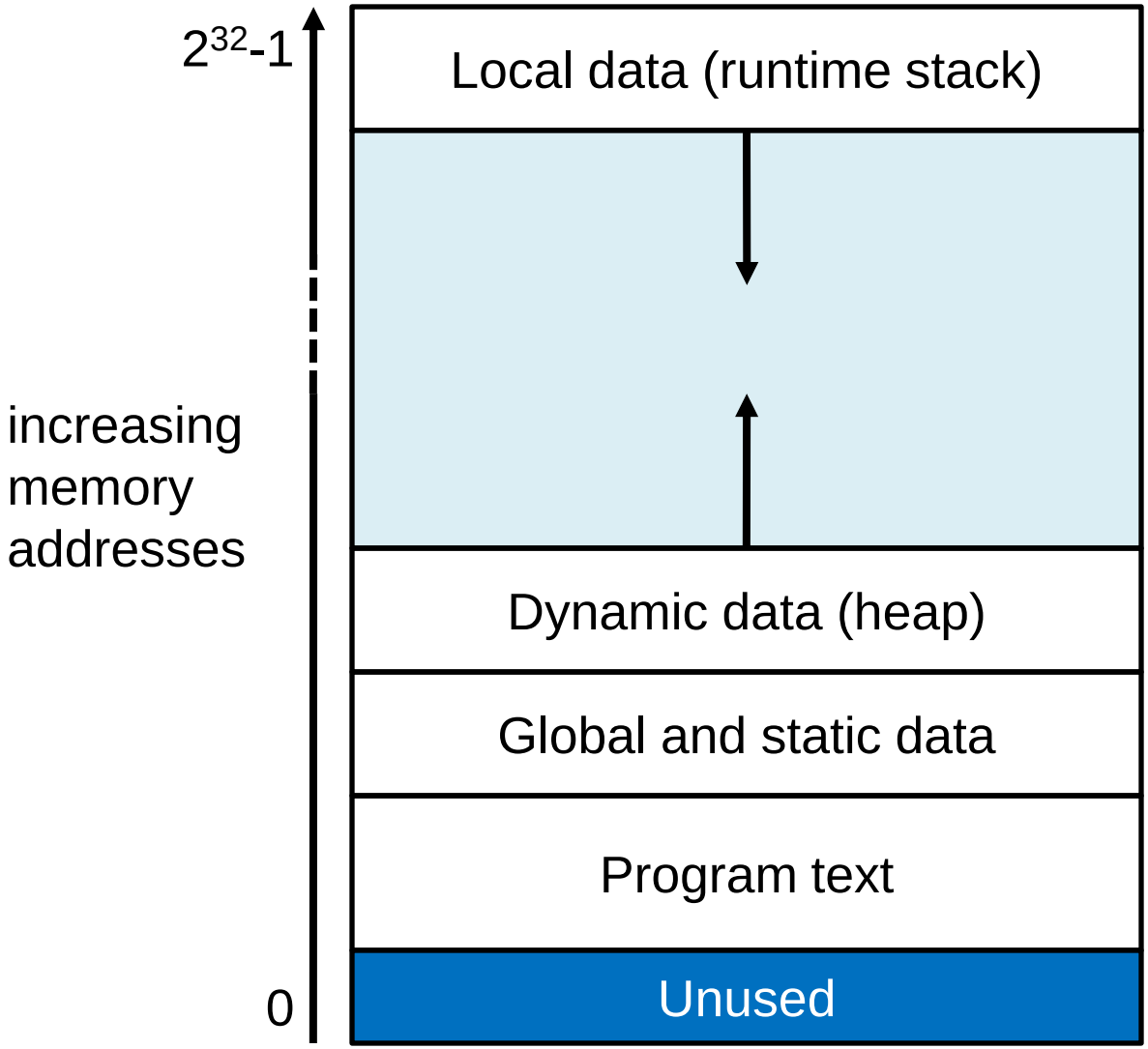
Processes

- Programs are often written as if they are the only things running on a system
- The OS allows them to work this way by providing an abstraction known as a process
- Process is a running program (one or more threads of control), along with all the data associated with it (an **address space**)
- OS uses context switching to give the appearance of multiple processes executing at once on a single processor

Virtual memory

- Each process is presented with the appearance of having 4 GB of available memory (on a 32-bit system) - this is virtual memory
- Physical memory ≠ virtual memory
  - Computer may not even have 4 GB of memory!
- Memory is organized in a particular manner; from bottom to top (in terms of addresses):
  - program code and data
  - heap
  - stack

Virtual address space (simplified)



Files in UNIX

- A file is a sequence of bytes - not a magical container holding the bytes, but the bytes themselves
- In UNIX, all I/O devices are modeled as a file
  - input from keyboard
  - output to screen
  - input/output from/to disk
  - input/output from/to network port
- Specific details of file organization can vary from OS to OS, and even filesystem to filesystem

Why learn about computer systems?

- Getting your programs to work correctly requires an understanding of how the computer does its work
- Making the computer do what you want can require in-depth knowledge of the OS
- For example, this Java method runs incredibly slowly, and it's entirely the programmer's fault:

```
public static int sumByColumns(int[][] array) {
  int sum = 0;
  for (int j = 0; j < COLS; j++)
    for (int i = 0; i < ROWS; i++)
      sum += array[i][j];
  return sum;
}
```

Chapter 1, Reek

A Quick Start with C

Comparison between C and Java

- C is procedural, not object-oriented
- C is fully compiled (to machine code), not to bytecode
- C allows direct manipulation of memory via pointers
- C does not have garbage collection
- Many of the basic language constructs in C act in similar ways to the way they work in Java
- C has many important, yet subtle, details

An example C program

- The following is C's version of the "Hello world" program:

```
#include <stdio.h>
int main() {
  printf("Hello world\n");
  return 0;
}
```

- How does it accomplish its goal?
  - includes library header file with function declarations (so call to **`printf()`** compiles OK)
  - provides definition of **`main()`** function, where all C programs begin
  - returns from **`main()`** to end program - the value returned can be checked by the program that invoked this one

Important caveats

- You will be writing C code that conforms to the C90 standard - this means a few things have to be kept in mind:
  - All comments must be of the `/* */` variety; C90 does not recognize `//` (single-line) comments
  - All variables must be declared at the beginning of a block (immediately after an opening brace); failure to do this will trigger "mixed declarations and code" error messages

Terrible style

```
if (condition) do_something();
  always_do_this();

if (x) y = j;

while (condition) x++;

if (c) {
  f();
  g();}

/* this funtion do stuf */

/* call f with value of 10 */
f(5);
```

- These examples are all bad style and you will lose credit for using them
  - statements executed conditionally should be put on separate lines from the condition
  - closing braces should be first thing on a line
  - comments should be spelled properly, and have correct and useful information

C function declarations

- Function prototype declarations provide information about a function's return type and parameters, but do not define the function
- Using function declarations allows the compiler to check your function calls for correctness
- Examples:

```
void foo(int, int, double);
int bar(double x, double y);
```

Declarations and the Preprocessor

- **`#include`**
  - provides ability to include declarations from other files
    - usually have the file name extension .h
  - generally only include function prototypes, and type and variable declarations
  - actual code is kept in a different file, usually with the extension .c
    - files with code are compiled individually to *object* code
    - and then linked together to create a file with *executable* code
- Two ways to use **`#include`**

  `#include <stdio.h>`
    - for standard system files

  `#include "swap.h"`
    - for user-written files

Compiling a C program

- C programs must be compiled to be executed
- Use the **`gcc`** program to build your programs
  - invocation: **`gcc`** options source-files
  - common options

`-g`	enable debugging
`-Wall`	warn about common things that may be problems
`-o filename`	place output in filename
`-c`	only compile to object file, don't link

  - a very simple compilation command

    ```
    gcc file.c
    ```

  - a simple compilation command

    ```
    gcc -g -Wall -o prog1 prog1.c tools.c
    ```

  - executable is run in UNIX by just typing its name (sometimes preceded by "**`./`**", like this: **`./prog1`**)
- There are programs to make the compiling process easier