Introduction to Computer Systems
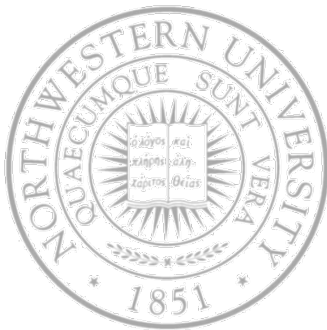
Today:
- Welcome to EECS 213
- Lecture topics and assignments

Next time:
- Bits & bytes
- and some Boolean algebra

Welcome to Intro. to Computer Systems

- Everything you need to know
  http://aqualab.cs.northwestern.edu/classes/EECS213/eecs-213-s12.html

- Your instructor: Fabián E. Bustamante

- Your TA: Zach Bischof

- Communication channels:
  – Course webpage
  – Google group
  – Email (subject "EECS343: <helpful comment>")

- Prerequisites
  – EECS 211 or equivalent
  – Experience with C or C++ - required
  – EECS 311 - useful

Course theme

*Abstraction is good,
but don't forget reality!*

- Courses have so far emphasized abstraction
  - Abstract data types,
  - asymptotic analysis, …
- Abstractions have limits
  - Especially in the presence of bugs
  - Need to understand underlying implementations
- Useful outcomes from taking this course
  - Become more effective programmers
  - Prepare for later "systems" classes in CS & CE

Course perspective

- Most systems courses are builder-centric
  - Operating Systems: Implement portions of an OS
  - Compilers: Write compiler for simple language
  - Networking: Implement and simulate network protocols
- This course is programmer-centric
  - By knowing more about the underlying system, one can be more effective as a programmer
    - Write programs that are more reliable and efficient
    - Incorporate features that require hooks into OS
  - We bring out the hidden hacker in everyone

Some topics covered

- Programs and data
  - Bits arithmetic, assembly, representation of C control …
- Memory hierarchy
  - Memory technology, memory hierarchy, caches, disks, locality
- Linking & exceptional control flow
  - Object files, dynamic linking, libraries, process control, …
- Virtual memory
  - Virtual mem., address translation, dynamic storage allocation
- Concurrency
  - High level & low-level I/O, threads, …
  - …
- Includes aspects of architecture and OS throughout

Course components

- Lectures
  - Higher level concepts
- Labs (4)
  - The heart of the course – in-depth understanding
  - 12.5% of grade each
  - Working on teams of 2
- Homework assignments (4)
  - 10% of grade
- Exams – midterm & final
  - 20% of grade each

Lab rationale

- Teach new skills and concepts
  - Data – Computer arithmetic, digital logic

    Out: Mar. 30 In: Apr. 10

  - Bomb – Assembly language, using a debugger, understanding the stack

    Out: Apr. 13 In: Apr. 24

  - Malloc – Data layout and organization, space/time tradeoffs

    Out: May 5 In: May 15

  - Shell – Processes, concurrency, process control, signals and signal handling

    Out: May 18 In: May 29

Textbooks

- Required
  - Bryant & O'Hallaron, "Computer Systems: A Programmer's Perspective", PH 2010.

- Recommended
  - Kernighan & Ritchie (K&R), "The C Programming Language, Second Edition", PH 1988
  - R. Stevens and S. Rago, "Advanced Programming in the Unix Environment", 2nd Ed. AW 2005

Facilities

- Tlab (Tech F-252, on the bridge to Ford) and Wilkinson Lab (3rd floor).

- You should all have accounts by now; problems? contact root (root@eecs.northwestern.edu)

- Need physical access to labs? Contact Carol Surma (carol@eecs.northwestern.edu)

Policies

- ## Late policy
  - 10% off per day (up to 5 days late)
- ## Cheating
  - What is cheating?
    - Sharing code: either by copying, retyping, looking at, or supplying a copy of a file.
  - What is NOT cheating?
    - Helping others use systems or tools
    - Helping others with high-level design issues
    - Helping others debug their code

Hello World

- What happens and *why* when you run "hello" on your system?

```
/*hello world*/
# include <stdio.h>


int main()
{
    printf("hello, world\n");
}
```

- Goal: introduce key concepts, terminology, and components

Information is bits + context

- "hello.c" is a source code
  - Sequence of bits (0 or 1)
  - 8-bit data chunks are called *bytes*
  - Each byte has an integer value, corresponding to some character (ASCII, e.g. '#' → 35)
  - Files made up of ASCII char. → text files
  - All other files → binary files (e.g., 35 is a part of a machine command)
- Context is key
  - Same byte sequence might represent a character string or machine instruction

Programs translated by other programs

```
unix> gcc –o hello hello.c
```

hello.c → **Pre-processor (cpp)** → hello.i → **Compiler (cc1)** → hello.s → **Assembler (as)** → hello.o → **Linker (ld)** → hello

printf.o → (into Linker)

Source program (text) | *Modified source program (text)* | *Assembly program (text)* | *Relocatable object programs (binary)* | *Executable object program (binary)*
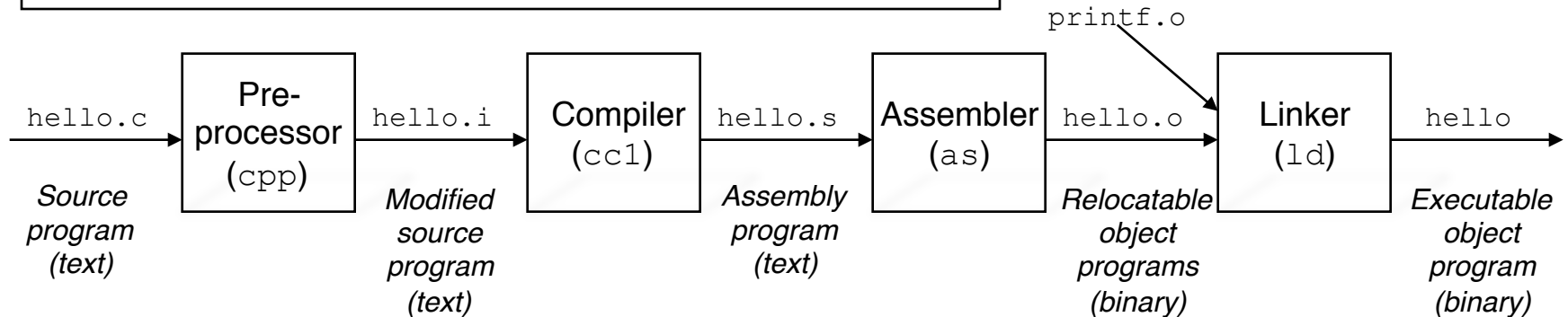
- Pre-processing
  – E.g., #include <stdio.h> is inserted into hello.i
- Compilation (.s)
  – Each statement is an assembly language program
- Assembly (.o)
  – A binary file whose bytes encode mach. language instructions
- Linking
  – Get printf() which resides in a separate precompiled object file
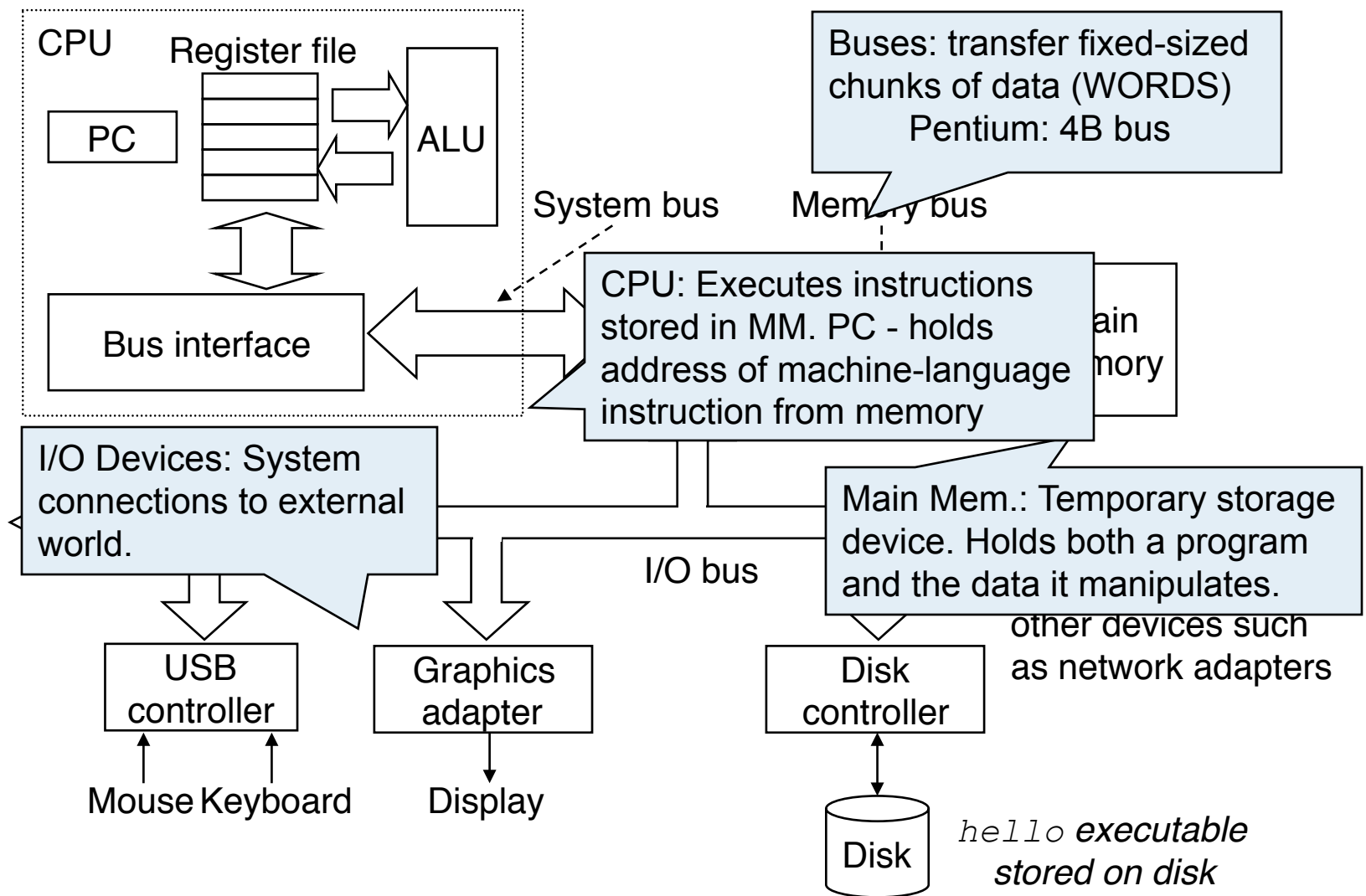
Running `Hello`

- Running `hello`

```
unix> ./hello
hello, world
unix>
```

- What's the shell?

- What does it do?
  - prints a prompt
  - waits for you to type command line
  - loads and runs hello program …

Hardware organization

CPU

Register file

PC

ALU

System bus

Memory bus

Bus interface

Buses: transfer fixed-sized chunks of data (WORDS)
Pentium: 4B bus

CPU: Executes instructions stored in MM. PC - holds address of machine-language instruction from memory

Main Memory

I/O Devices: System connections to external world.

Main Mem.: Temporary storage device. Holds both a program and the data it manipulates.

I/O bus

other devices such as network adapters

USB controller

Graphics adapter

Disk controller

Mouse Keyboard

Display

Disk

hello **executable stored on disk**

Running `Hello`

Reading the hello command from the keyboard

Register file

PC

ALU

Bus interface

System bus

I/O bridge

Memory bus

Main memory

"./hello"

I/O bus

Expansion slots for other devices such as network adapters

USB controller

Graphics adapter

Disk controller

Mouse Keyboard

Display

Disk

User types "hello"

Running `Hello`

Register file

PC

ALU

Bus interface

System bus

I/O bridge

Memory bus

Main memory

Shell program loads hello.exe into main memory

"hello,world\n"

`hello` *code*

I/O bus

USB controller

Mouse Keyboard

Graphics adapter

Display

Disk controller

Disk

Expansion slots for other devices such as network adapters

`hello` *executable stored on disk*

Running `Hello`

Register file

PC

ALU

Bus interface

System bus

I/O bridge

Memory bus

Main memory

The processor executes instructions and displays "hello…"

"hello,world\n"

`hello` *code*

I/O bus

Expansion slots for other devices such as network adapters

USB controller

Graphics adapter

Disk controller

Mouse Keyboard

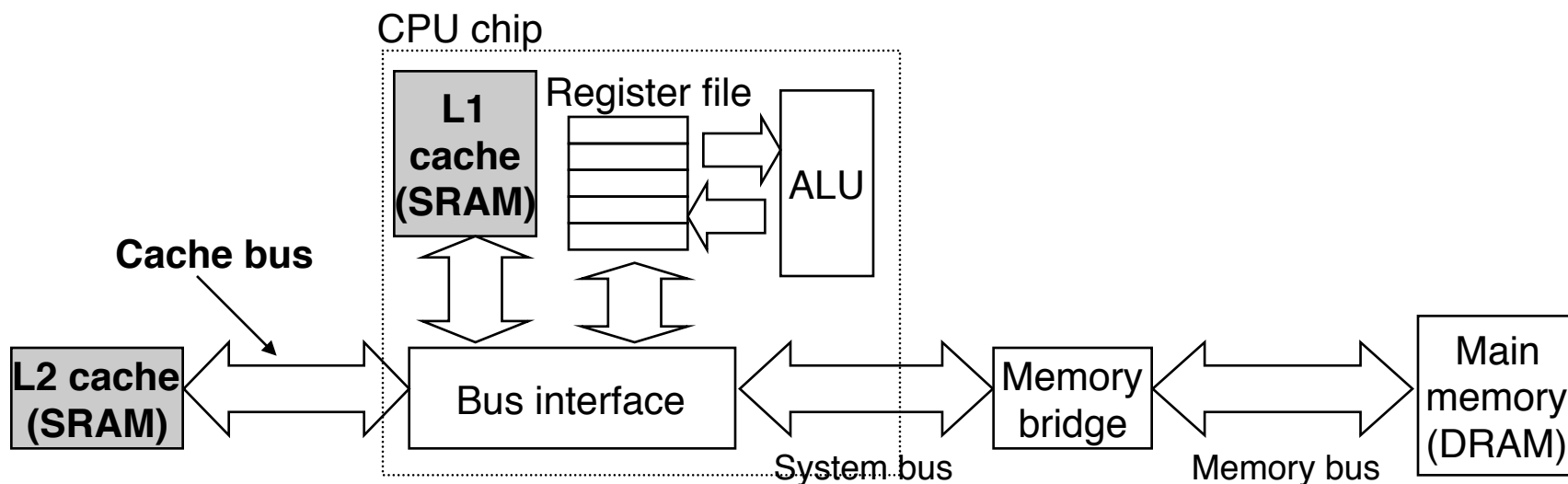Display

"hello,world\n"

Disk
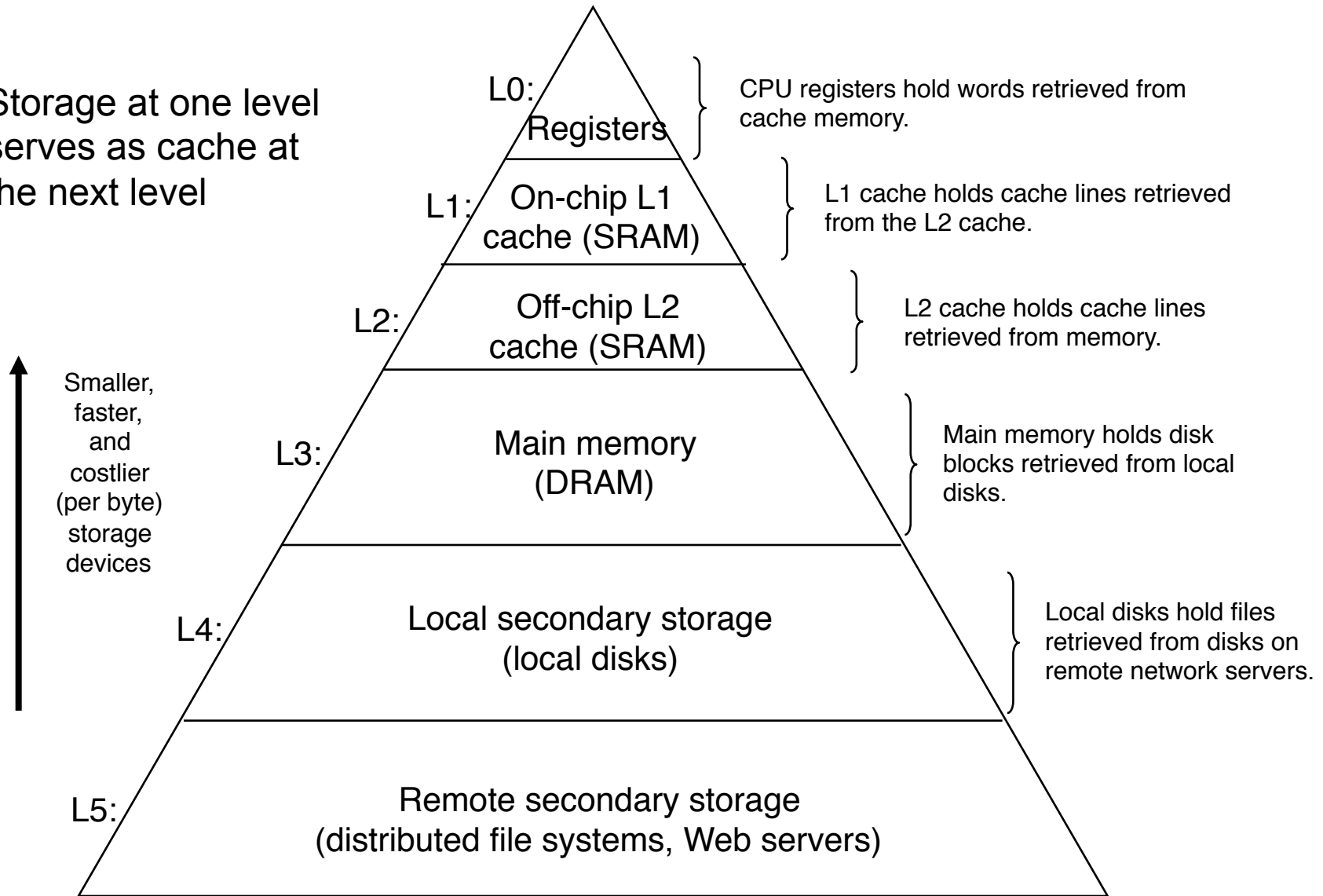
`hello` *executable stored on disk*

Caches matter

- System spends a lot of time moving info. around
- Larger storage devices are slower than smaller ones
  - Register file ~ 100 Bytes & Main memory ~ millions of Bytes
- Easier and cheaper to make processors run faster than to make main memory run faster
 - Standard answer – cache

CPU chip

L1 cache (SRAM) Register file ALU

Cache bus

L2 cache (SRAM)

Bus interface

Memory bridge

Main memory (DRAM)

System bus

Memory bus

Storage devices form a hierarchy

Storage at one level serves as cache at the next level

L0: Registers — CPU registers hold words retrieved from cache memory.

L1: On-chip L1 cache (SRAM) — L1 cache holds cache lines retrieved from the L2 cache.

L2: Off-chip L2 cache (SRAM) — L2 cache holds cache lines retrieved from memory.

L3: Main memory (DRAM) — Main memory holds disk blocks retrieved from local disks.

L4: Local secondary storage (local disks) — Local disks hold files retrieved from disks on remote network servers.

L5: Remote secondary storage (distributed file systems, Web servers)

Smaller, faster, and costlier (per byte) storage devices

Operating system

- OS – a layer of software interposed between the application program and the hardware

Application programs			} Software
Operating system			
Processor	Main memory	I/O devices	} Hardware
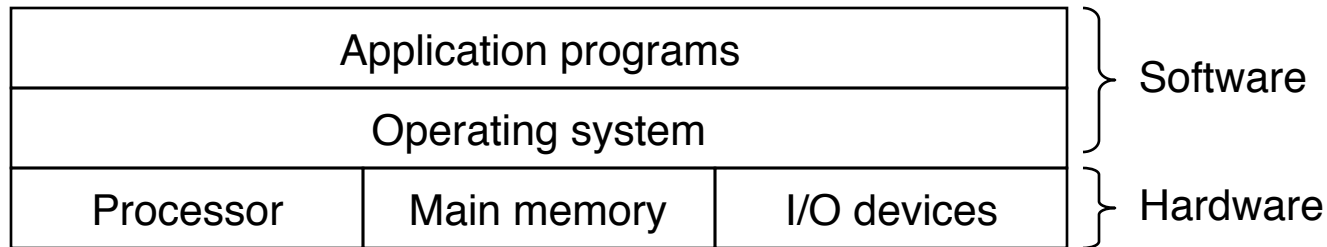
- Two primary goal
  - Protect resources from misuse by applications
  - Provide simple and uniform mechanisms for manipulating low-level hardware devices

OS Abstractions

- Files – abstractions of I/O devices
- Virtual Memory – abstraction for main memory and I/O devices
- Processes – abstractions for processor, main memory, and I/O devices

Processes

Virtual memory

Files

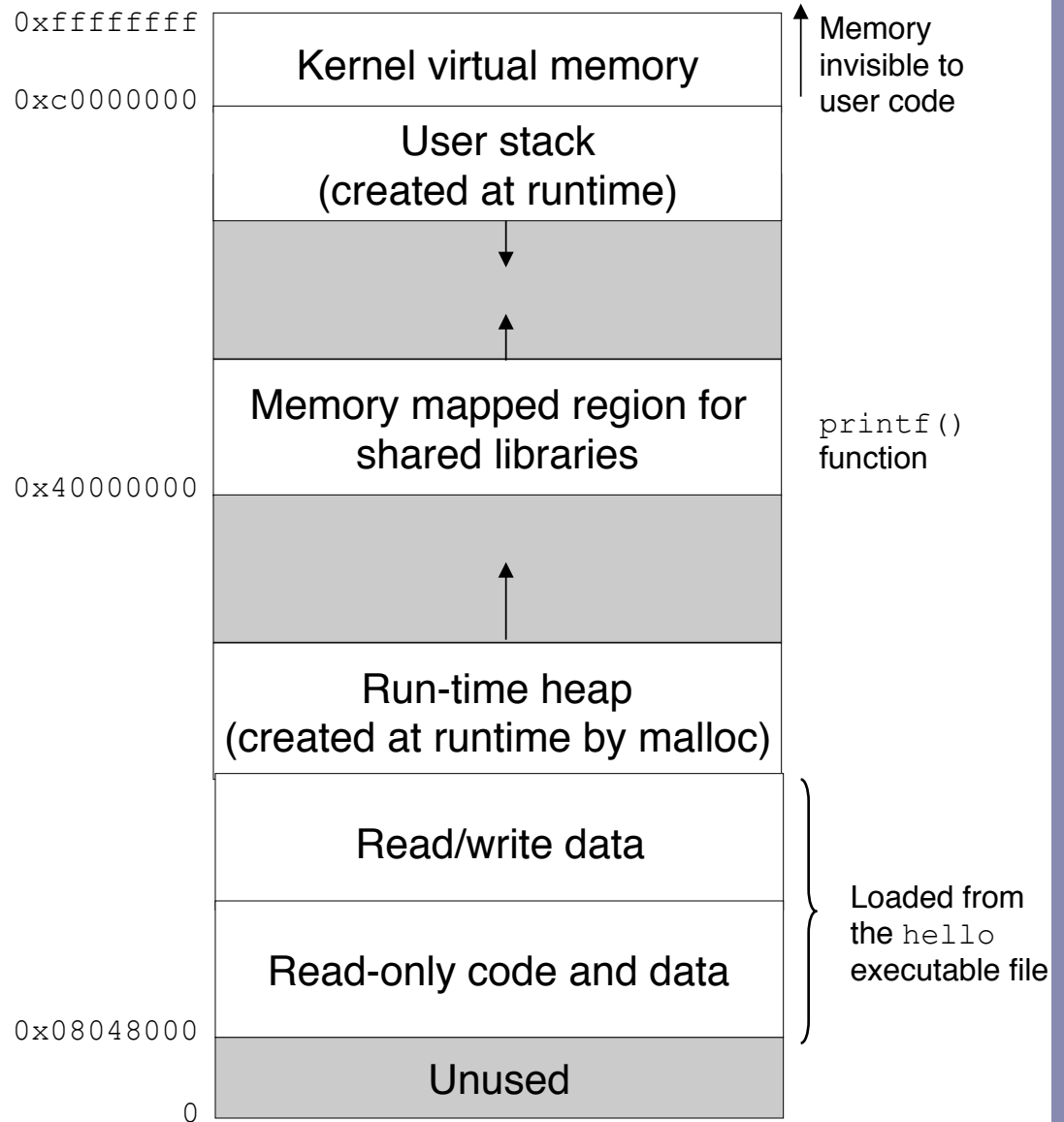| Processor | Main memory | I/O devices |
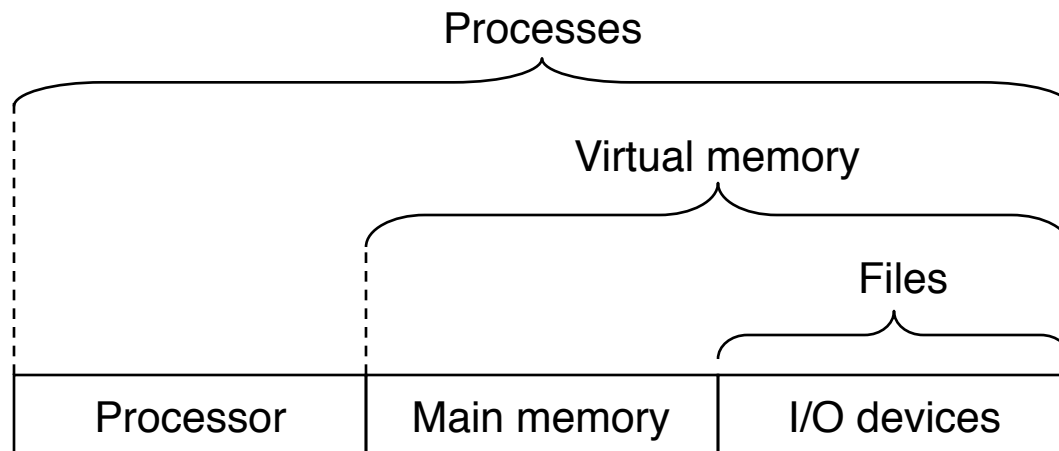
Processes

- OS provides the illusion of a dedicated machine per process
- Process
  - OS's abstraction of a running program
- Context switch
  - Saving context of one process, restoring that of another one
 - Distorted notion of time
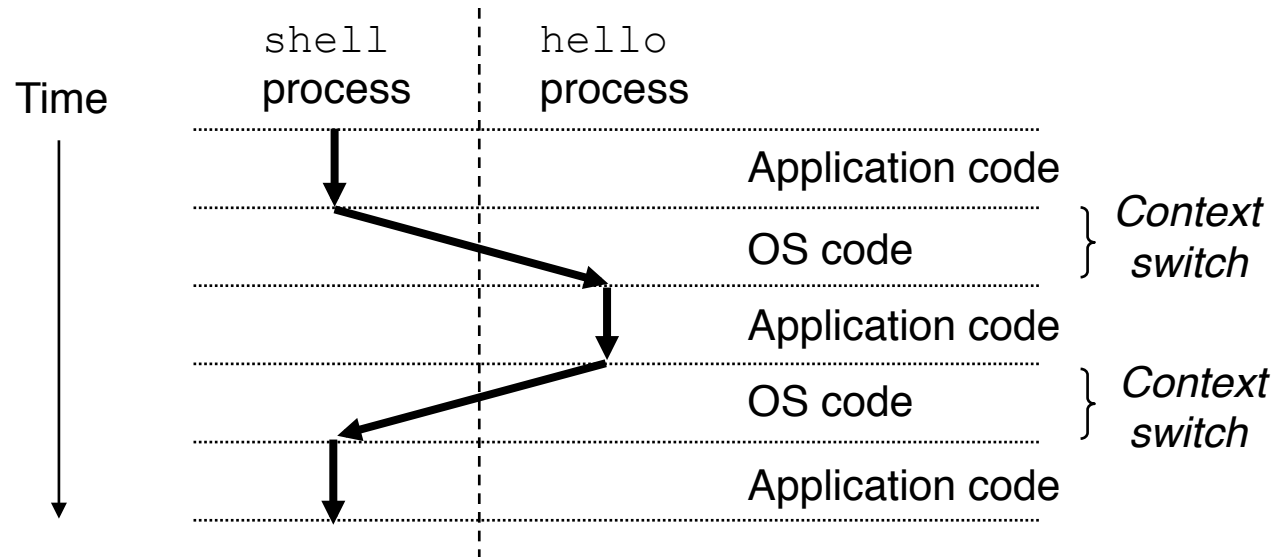
Virtual memory

- Illusion that each process has exclusive use of a large main memory
- Example
  - Virtual address space for Linux

Address	Region	Note
0xffffffff	Kernel virtual memory	Memory invisible to user code
0xc0000000	User stack (created at runtime)	
	↓ ↑	
	Memory mapped region for shared libraries	`printf()` function
0x40000000		
	↑	
	Run-time heap (created at runtime by malloc)	
	Read/write data	Loaded from the `hello` executable file
	Read-only code and data	
0x08048000		
0	Unused	

Networking

- Talking to other systems
- Network – seen as another I/O device
- Many system-level issues arise in presence of network
  - Coping with unreliable media
  - Cross platform compatibility
  - Complex performance issues
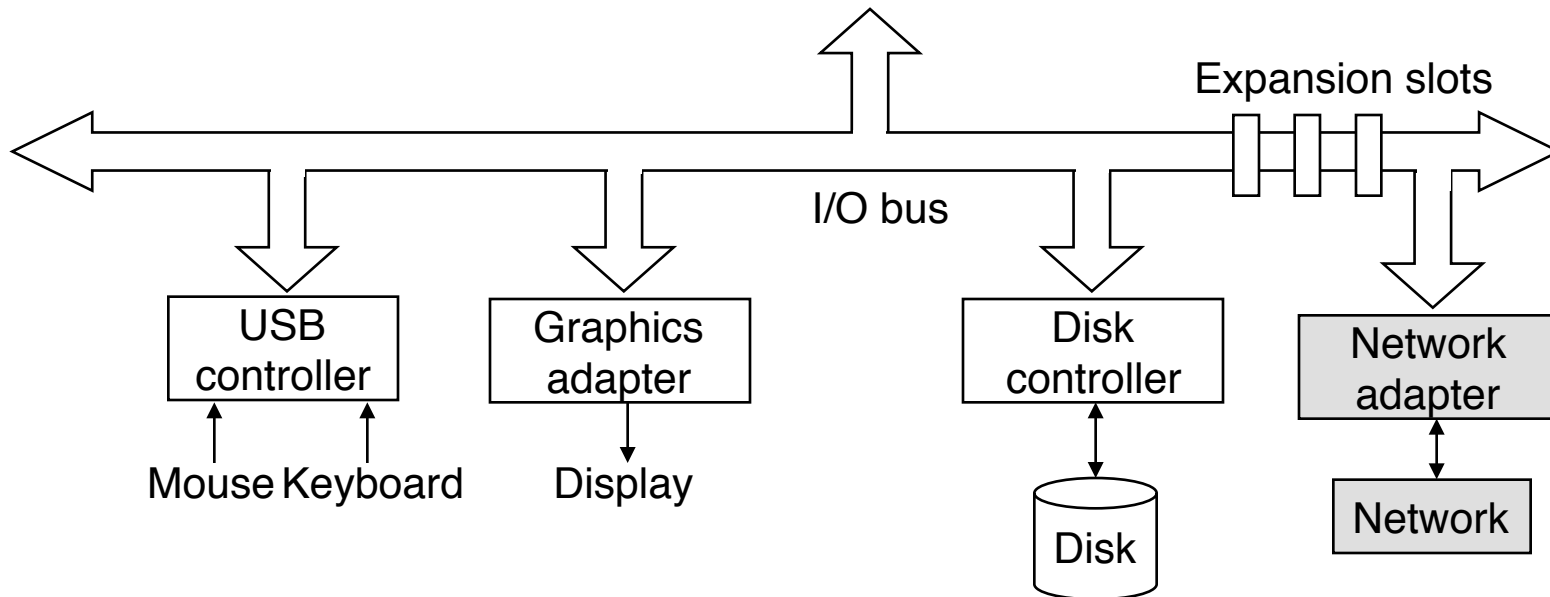
Conclusions

- A computer system is more than just hardware
  - A collection of intertwined HW & SF that must cooperate to achieve the end goal – running applications
- The rest of the course will expand on this