Course Overview

Computer Architecture

Instructor:

Dr. R. Shathanaa

Overview

- **Course theme**
- **Five realities**
- **How the course fits into the curriculum**

Poll

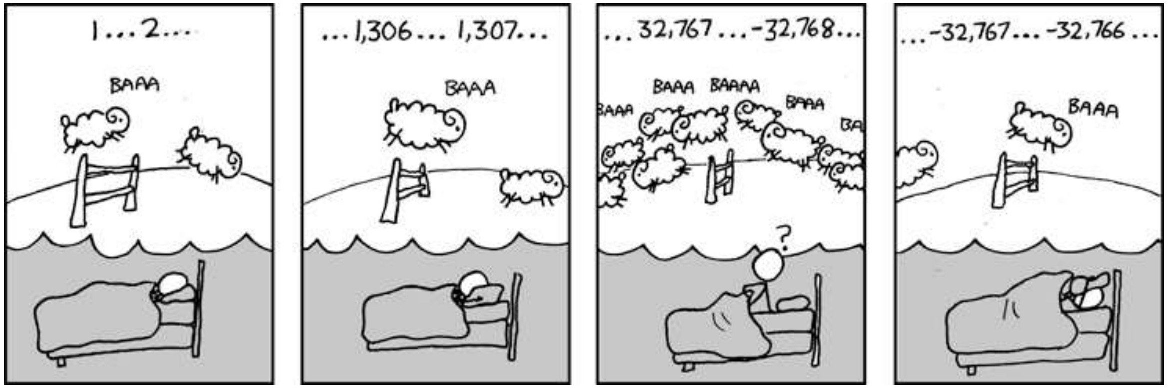
- **What is your career choice?**

Course Theme:
Abstraction Is Good But Don't Forget Reality

- **Most CS courses emphasize abstraction**
  - Abstract data types
  - Asymptotic analysis
- **These abstractions have limits**
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- **Useful outcomes**
  - Become more effective programmers
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune for program performance
  - Prepare for later "systems" classes in CS & ECE
    - Compilers, Operating Systems, Networks, Advanced Computer Architecture, Embedded Systems, Storage Systems, etc.

Great Reality #1:
Ints are not Integers, Floats are not Reals

- **Example 1: Is $x^2 \geq 0$?**
  - Float's: Yes!



  - Int's:
    - 40000 * 40000 = 1600000000
    - 50000 * 50000 = ??
- **Example 2: Is (x + y) + z = x + (y + z)?**
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - (1e20 + -1e20) + 3.14 --> 3.14
    - 1e20 + (-1e20 + 3.14) --> ??

Source: xkcd.com/571

Computer Arithmetic

- **Does not generate random values**
  - Arithmetic operations have important mathematical properties
- **Cannot assume all "usual" mathematical properties**
  - Due to finiteness of representations
  - Integer operations satisfy "ring" properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy "ordering" properties
    - Monotonicity, values of signs
- **Observation**
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

Poll

- **How confident are you in programming?**

Great Reality #2: You've Got to Know Assembly

- **Chances are, you'll never write programs in assembly**
  - Compilers are much better & more patient than you are
- **But: Understanding assembly is key to machine-level execution model**
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

Great Reality #3: Memory Matters

Random Access Memory Is an Unphysical Abstraction

- **Memory is not unbounded**
  - It must be allocated and managed
  - Many applications are memory dominated
- **Memory referencing bugs especially pernicious**
  - Effects are distant in both time and space
- **Memory performance is not uniform**
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  ➙  3.14
fun(1)  ➙  3.14
fun(2)  ➙  3.1399998664856
fun(3)  ➙  2.00000061035156
fun(4)  ➙  3.14
fun(6)  ➙  Segmentation fault
```

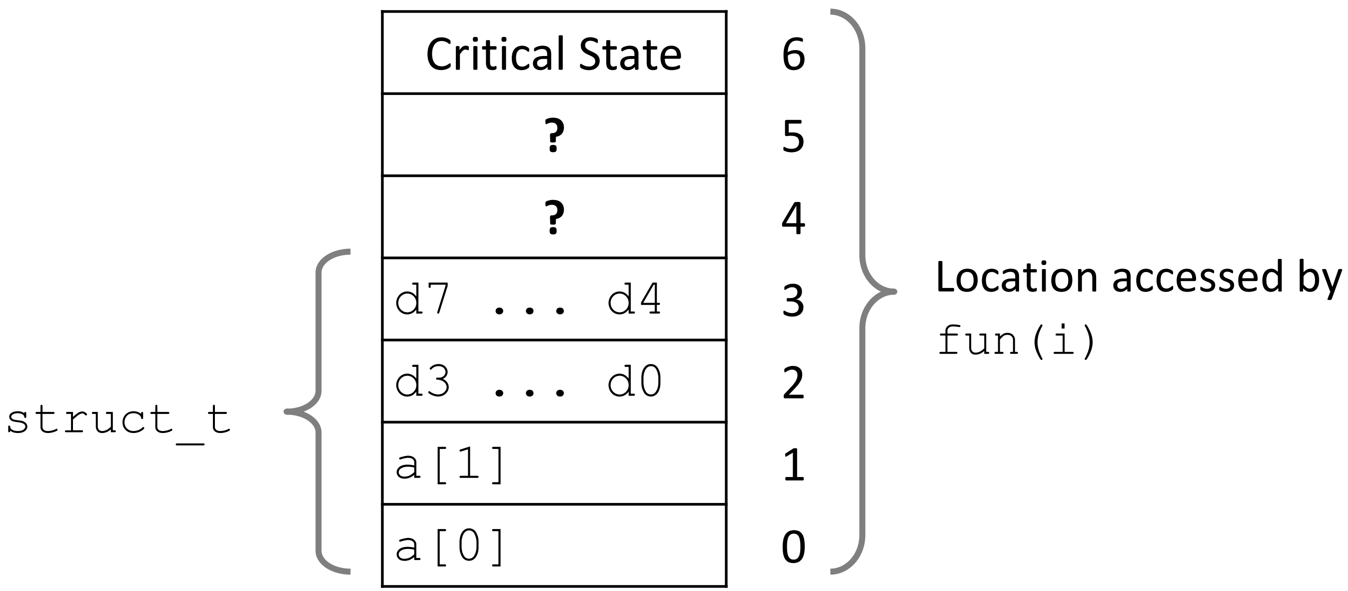
- Result is system specific

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault
```

Explanation:



Memory Referencing Errors

- **C and C++ do not provide any memory protection**
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- **Can lead to nasty bugs**
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- **How can I deal with this?**
  - Program in Java, Ruby, Python, ML, …
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

Great Reality #4: There's more to performance than asymptotic complexity

- **Constant factors matter too!**
- **And even exact op count does not predict performance**
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Memory System Performance Example

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

4.3ms **2.0 GHz Intel Core i7 Haswell** **81.8ms**

- **Hierarchical memory organization**
- **Performance depends on access patterns**
  - Including how step through multi-dimensional array

Great Reality #5: Computers do more than execute programs

- **They need to get data in and out**
  - I/O system critical to program reliability and performance

- **They communicate with each other over networks**
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

Course Perspective

- **Most Systems Courses are Builder-Centric**
  - Computer Architecture
    - Design pipelined processor in Verilog
  - Operating Systems
    - Implement sample portions of operating system
  - Compilers
    - Write compiler for simple language
  - Networking
    - Implement and simulate network protocols

Course Perspective (Cont.)

- **Our Course is Programmer-Centric**
  - Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer
  - Enable you to
    - Write programs that are more reliable and efficient
    - Incorporate features that require hooks into OS
      - E.g., concurrency, signal handlers
  - Cover material in this course that you won't see elsewhere

Now, why take this course?

- End of Moore's law
- Multiprocessor architectures
- Emergence of new platforms

- To be better, a hardware designer, compiler designer, OS designer AND a software developer need to know about the basic hardware.

Textbooks

- **Randal E. Bryant and David R. O'Hallaron,**
  - *Computer Systems: A Programmer's Perspective*, **Third Edition** (CS:APP3e), Pearson, 2016
  - http://csapp.cs.cmu.edu
  - This book really matters for the course!
    - How to solve labs
    - Practice problems typical of exam problems

- **Brian Kernighan and Dennis Ritchie,**
  - *The C Programming Language*, Second Edition, Prentice Hall, 1988
  - Still the best book about C, from the originators

Welcome
and Enjoy!

A Tour of Systems

C Hello Program

code/intro/hello.c

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

code/intro/hello.c

Information Is Bits + Context

The representation of `hello.c` illustrates a fundamental idea: All information in a system—including disk files, programs stored in memory, user data stored in memory, and data transferred across a network—is represented as a bunch of bits.

The only thing that distinguishes different data objects is the context in which we view them.

For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

#	i	n	c	l	u	d	e	*SP*	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	*SP*	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	*SP*	*SP*	*SP*	*SP*	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	*SP*	w	o	r	l	d	\	n	"	)	;	\n	*SP*
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	*SP*	*SP*	r	e	t	u	r	n	*SP*	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

Figure 1.2 **The ASCII text representation of** `hello.c`.

Programs Are Translated by Other Programs into Different Forms

linux> *gcc -o hello hello.c*

gcc -E hello.c

gcc -S hello.c

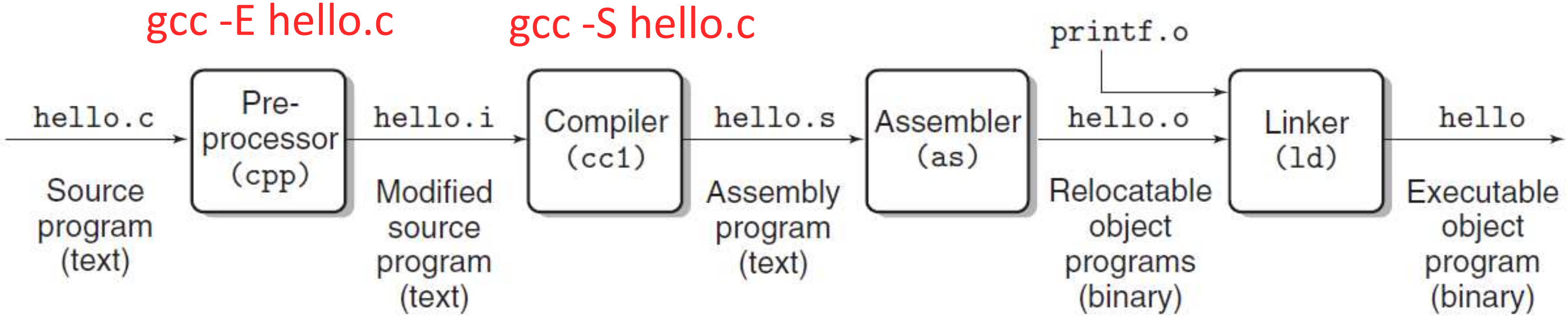


Figure 1.3 **The compilation system.**

gcc –o hello.o hello.c

- The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

- *Preprocessing phase.* The preprocessor (cpp) modifies the original C program according to directives that begin with the '#' character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.
- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. This program includes the following definition of function `main`:

```
1  main:
2    subq    $8, %rsp
3    movl    $.LC0, %edi
4    call    puts
5    movl    $0, %eax
6    addq    $8, %rsp
7    ret
```

Each of lines 2–7 in this definition describes one low-level machine-language instruction in a textual form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. This file is a binary file containing 17 bytes to encode the instructions for function `main`. If we were to view `hello.o` with a text editor, it would appear to be gibberish.

- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an executable object file (or simply *executable*) that is ready to be loaded into memory and executed by the system.

It Pays to Understand How Compilation Systems Work

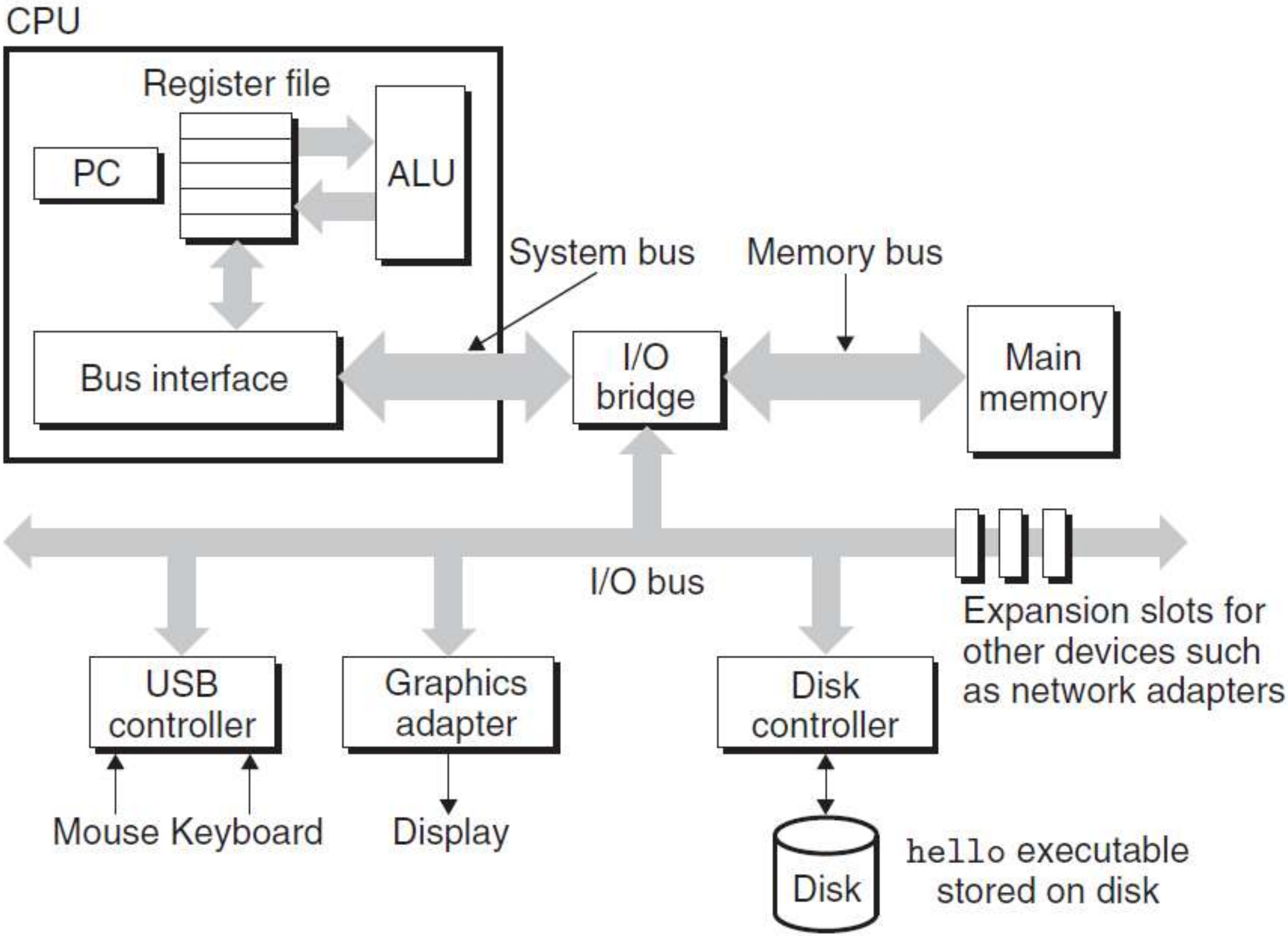
- *Optimizing program performance*
- *Understanding link-time errors*
- *Avoiding security holes*

Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell:*

```
linux> ./hello
hello, world
linux>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello` program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.



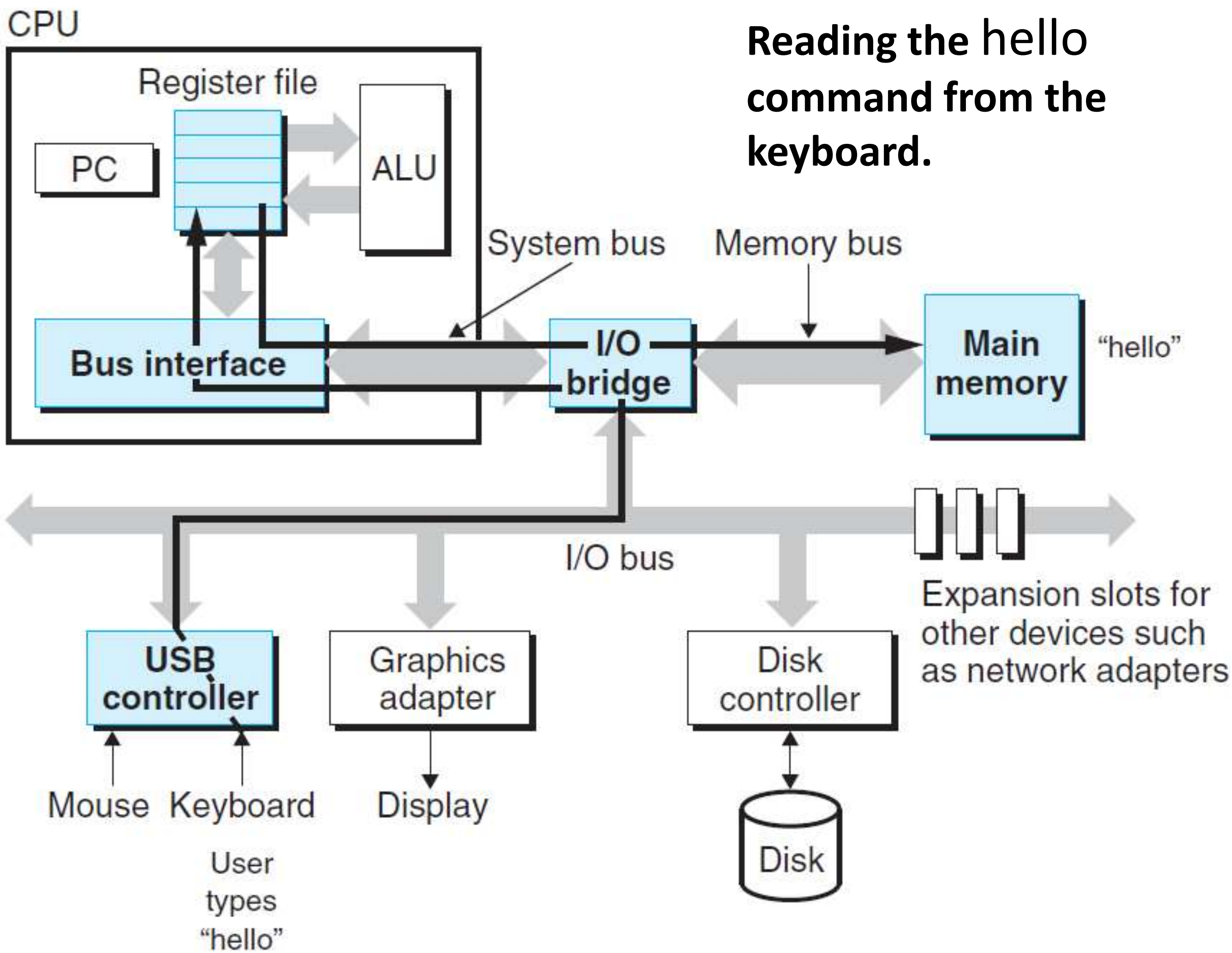
Hardware Organization of a System

Inside your computer

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-size chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). In this book, we do not assume any fixed definition of word size. Instead, we will specify what we mean by a "word" in any context that requires this to be defined.

Running the hello Program



Reading the hello **command from the keyboard.**

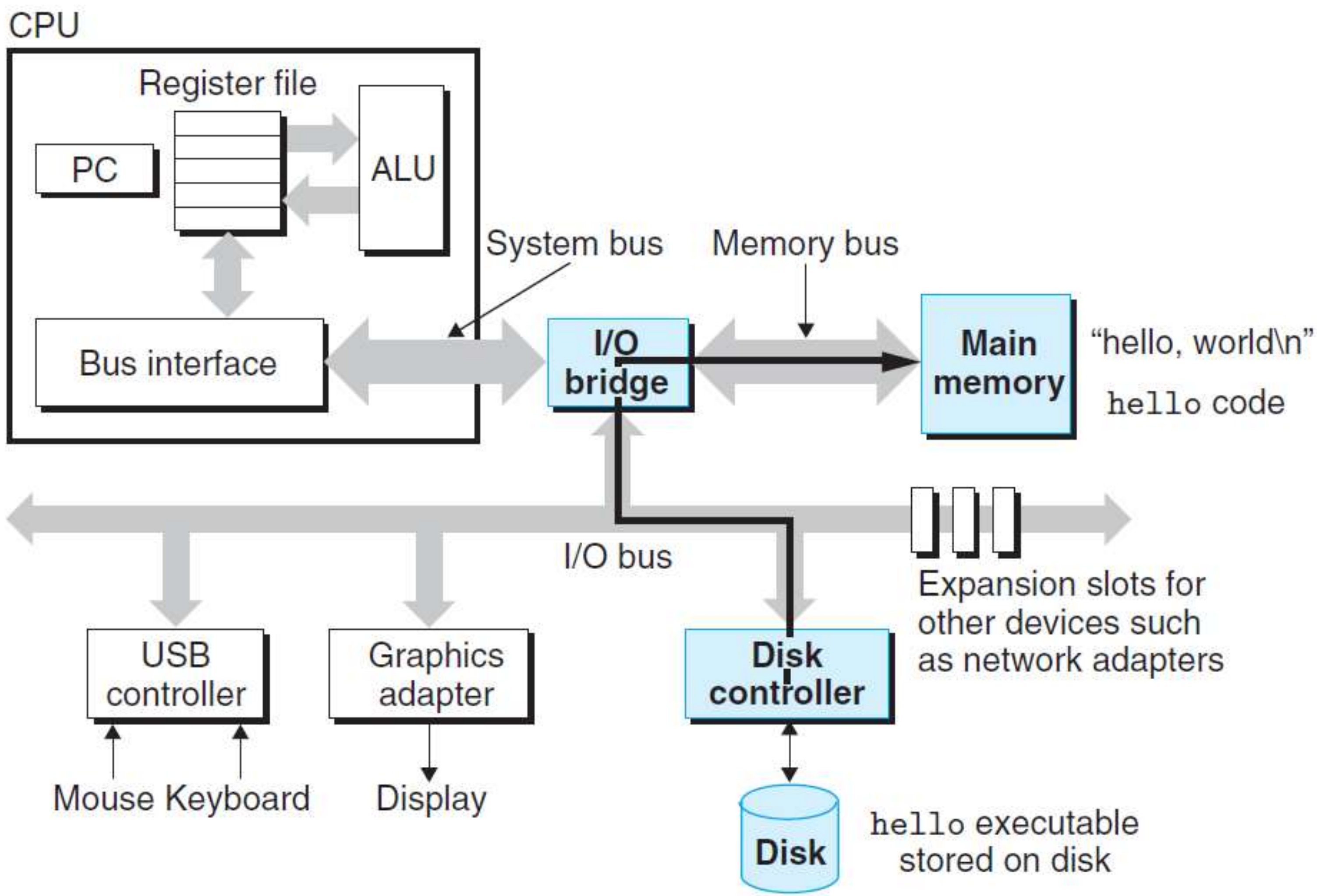


Figure 1.6 **Loading the executable from disk into main memory.**

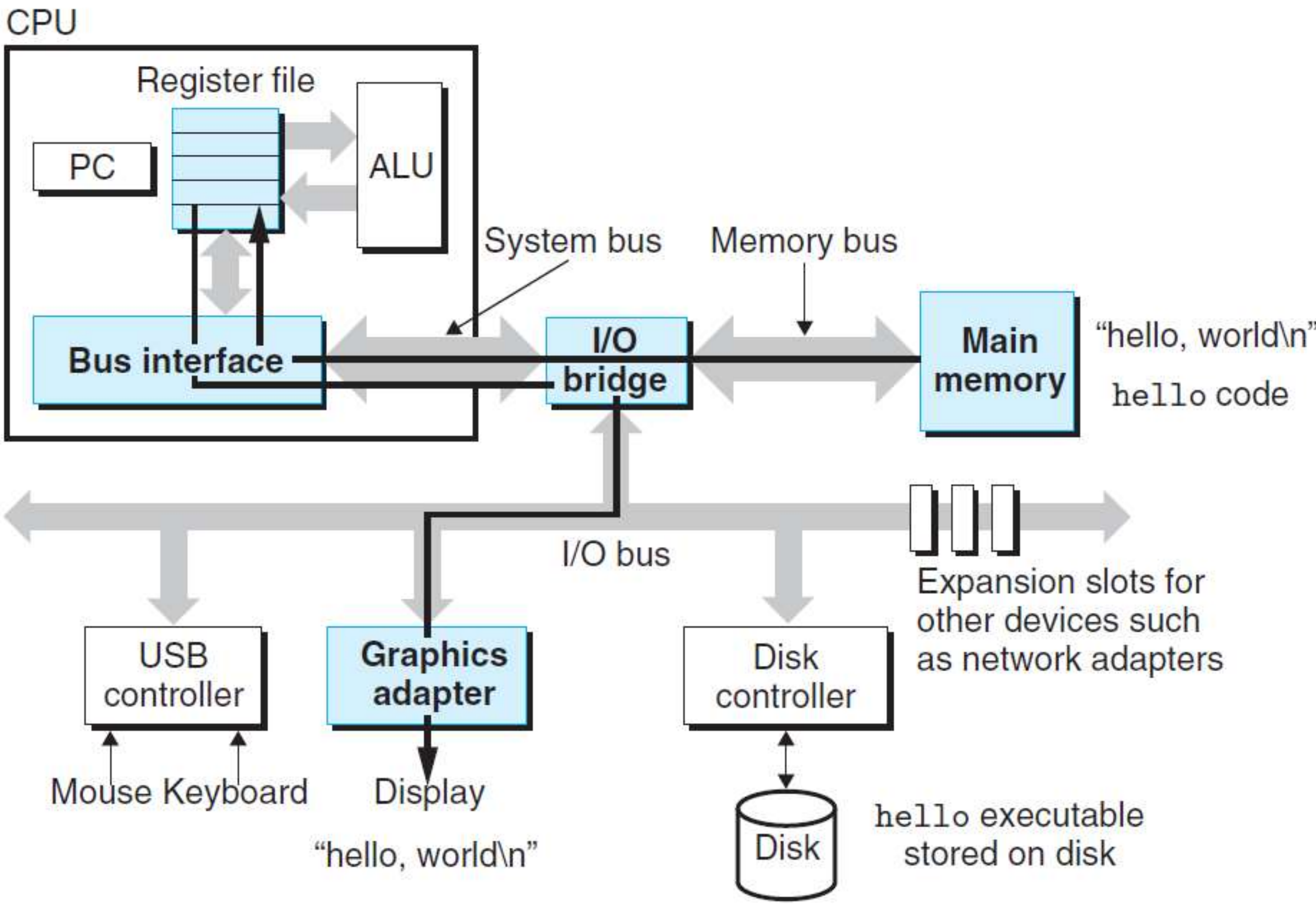


Figure 1.7 **Writing the output string from memory to the display.**

Processor Memory Gap

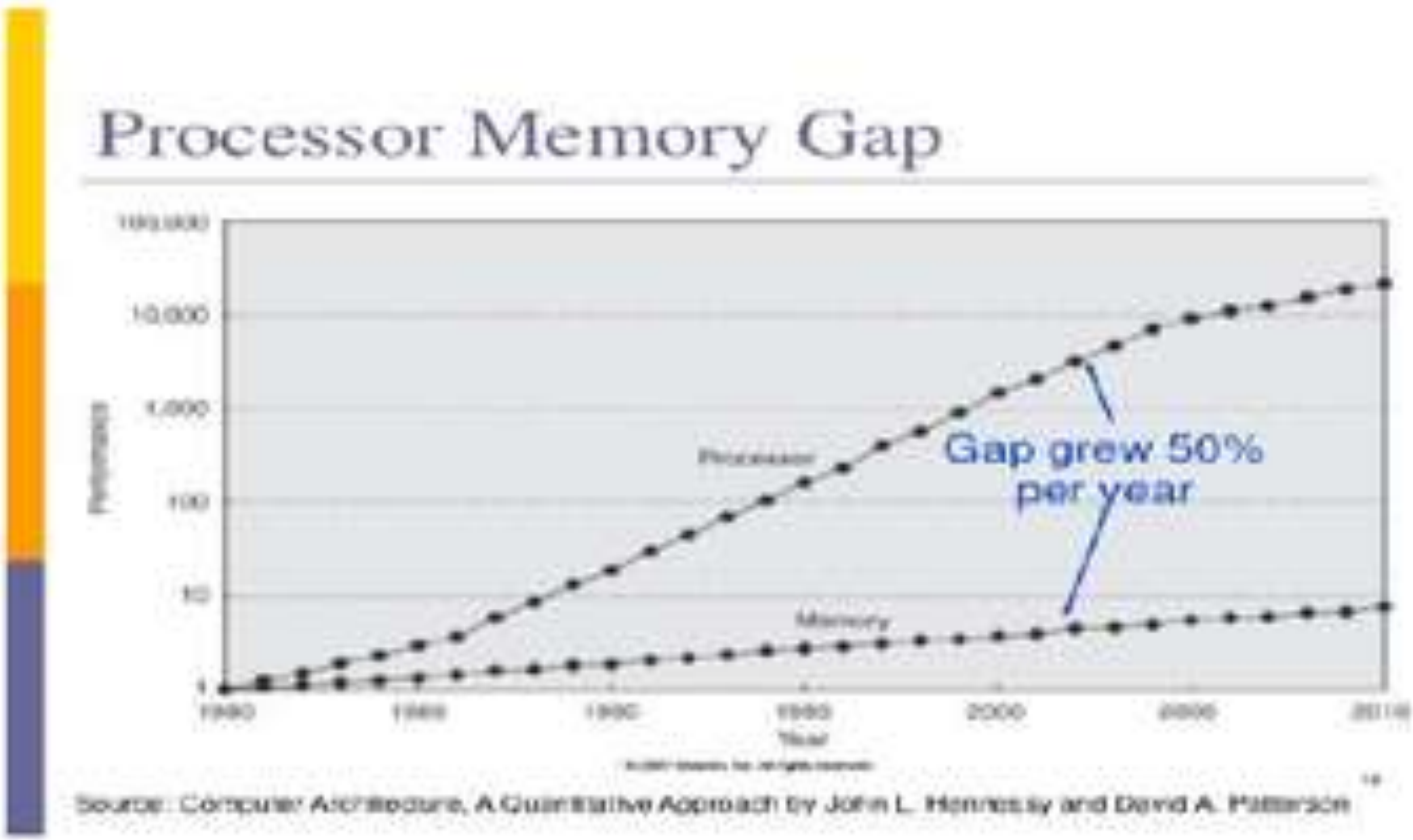
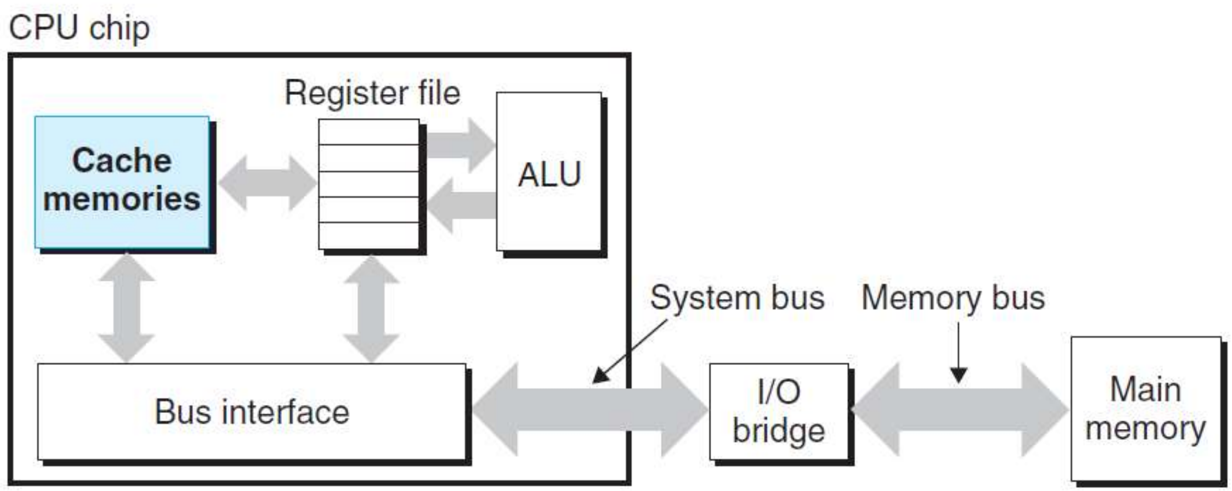


Fig. 1

Cache Matters

Figure 1.8
Cache memories.



Memory Hierarchy

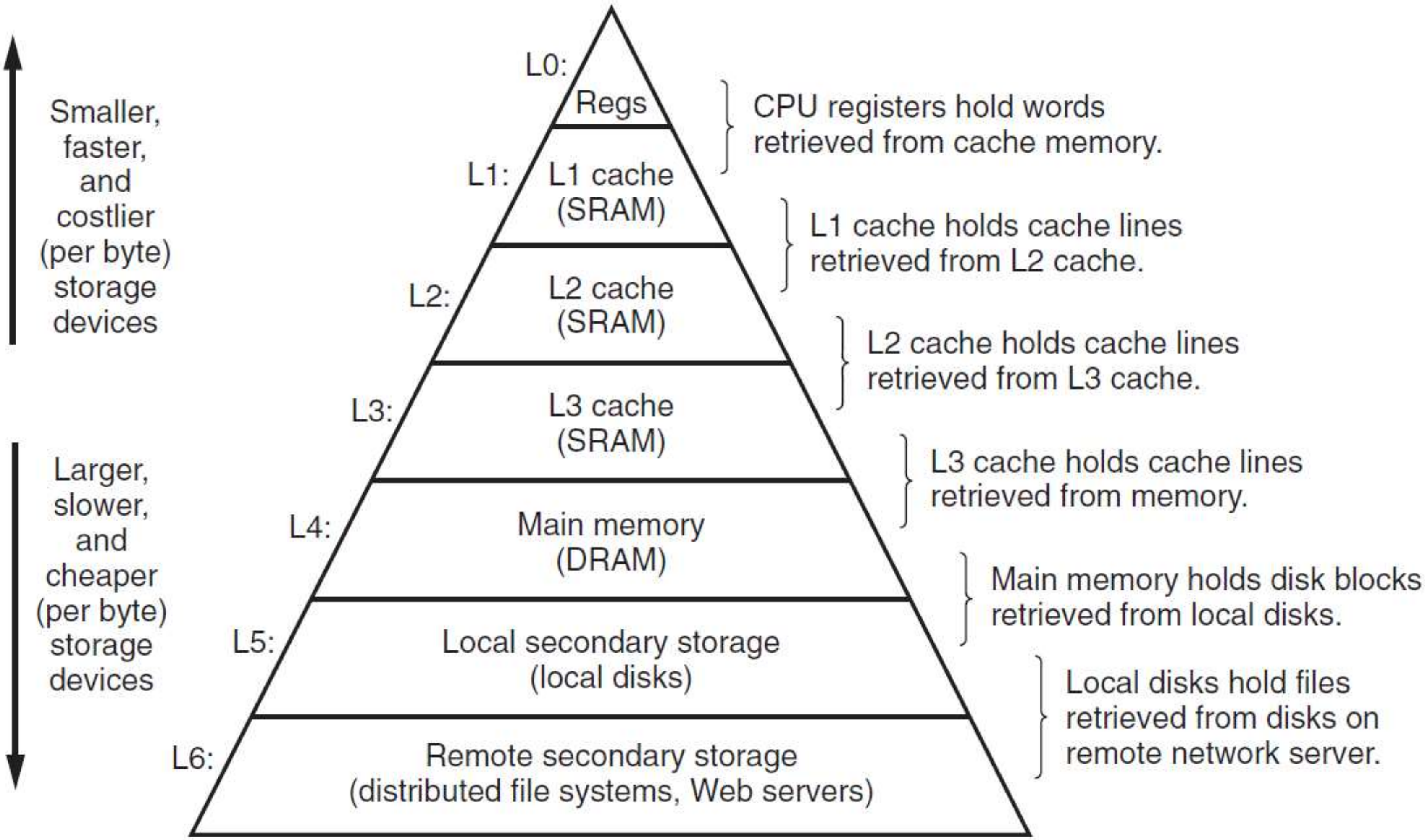


Figure 1.9 **An example of a memory hierarchy.**

The Operating System Manages the Hardware

Back to our hello example. When the shell loaded and ran the hello program,

and when the hello program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the

services provided by the *operating system*.

We can think of the operating system as

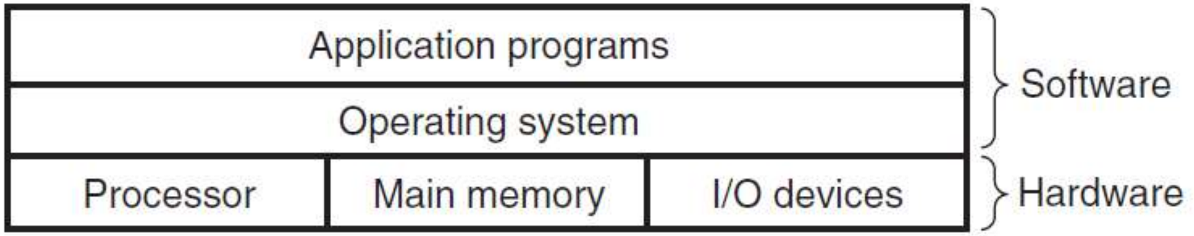
a layer of software interposed between the application program and the hardware,

as shown in Figure 1.10.

All attempts by an application program to manipulate the

hardware must go through the operating system.

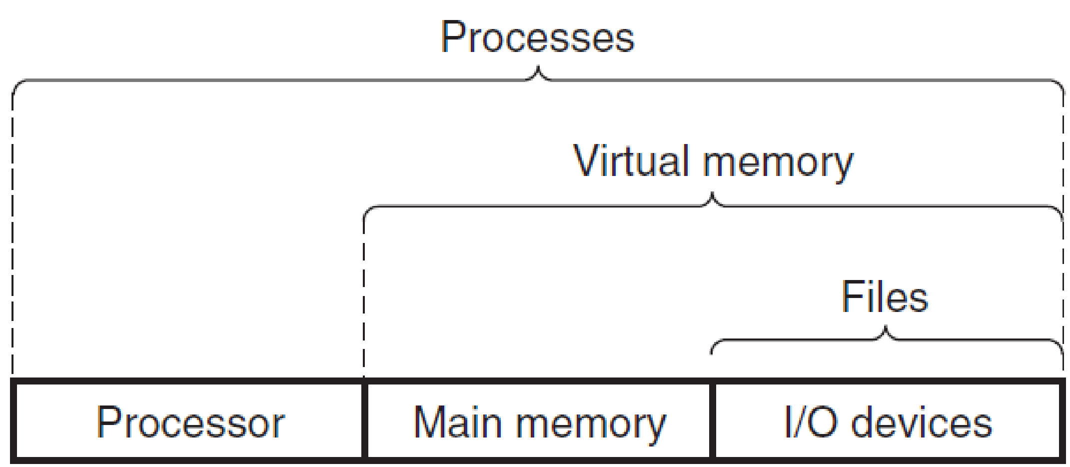
Figure 1.10
Layered view of a computer system.



The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices.

The operating system achieves both goals via the fundamental abstractions shown in Figure 1.11: *processes*, *virtual memory*, and *files*. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices.

Figure 1.11
Abstractions provided by an operating system.



Processes

When a program such as hello runs on a modern system, the operating system provides the illusion that the program is the only one running on the system.

A *process* is the operating system's abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware.

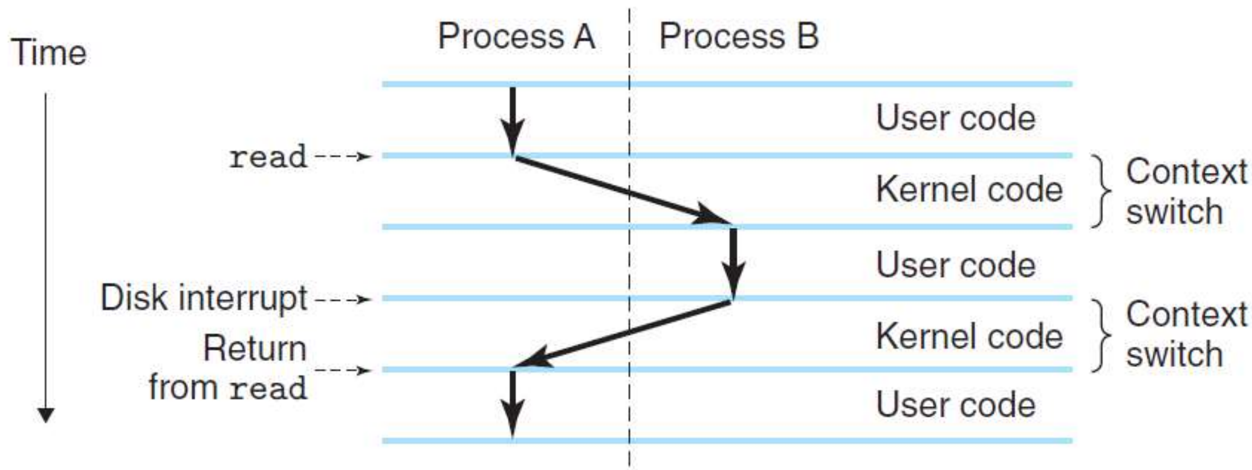
The operating system performs this interleaving
with a mechanism known as *context switching*. To simplify the rest of this discussion, we consider only a *uniprocessor system* containing a single CPU

The operating system keeps track of all the state information that the process needs in order to run. This state, which is known as the *context*, includes information such as the current values of the PC, the register file, and the contents of main memory.

At any point in time, a uniprocessor system can only execute the code for a single process.

When the operating system decides to transfer control from the current process to some new process, it performs a *context switch* by saving the context of the current process, restoring the context of the new process, and then passing control to the new process. The new process picks up exactly where it left off. Figure 1.12 shows the basic idea for our example hello scenario.

Figure 1.12
Process context switching.



Threads

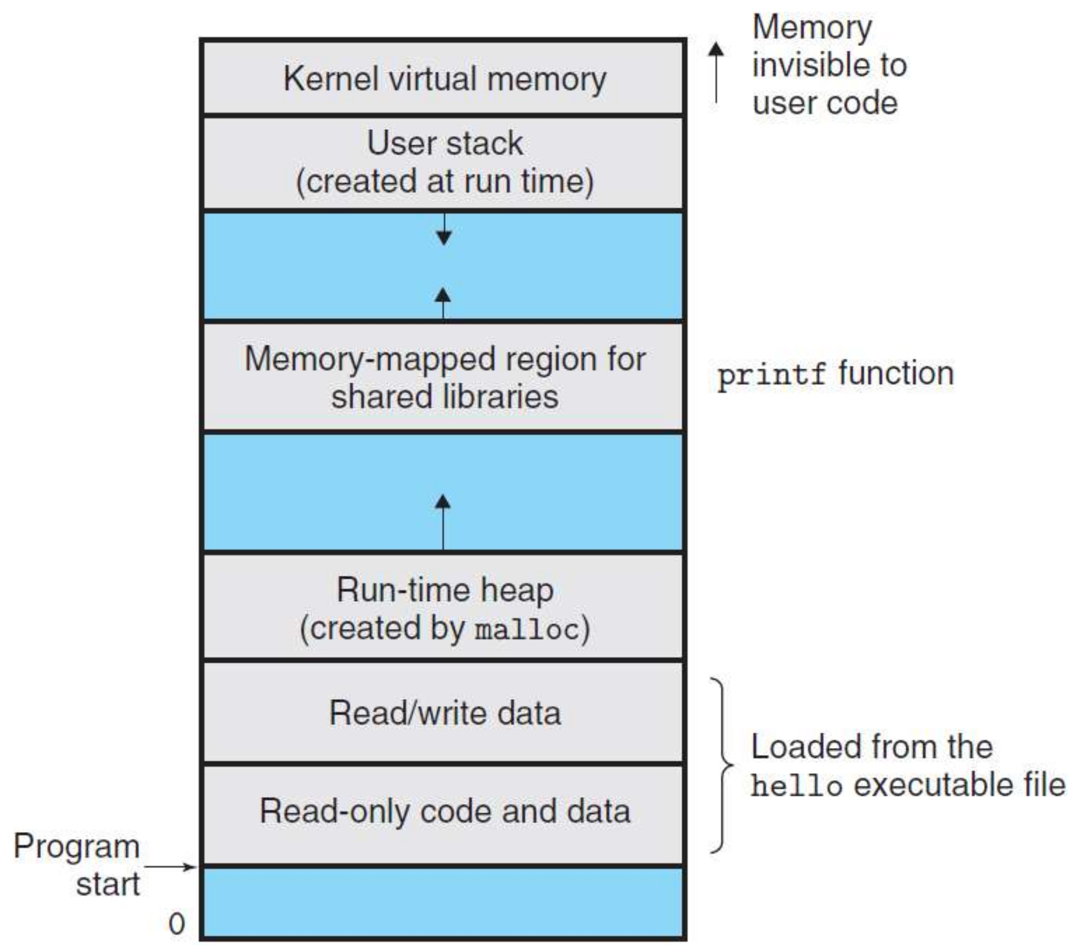
Although we normally think of a process as having a single control flow, in modern systems a process can actually consist of multiple execution units, called *threads*, each running in the context of the process and sharing the same code and global data.

Multi-threading is also one way to make programs run faster when multiple processors are available

Virtual Memory

- *Virtual memory* is an abstraction that provides each process with the illusion that it has exclusive use of the main memory.
- Each process has the same uniform view of memory, which is known as its *virtual address space*

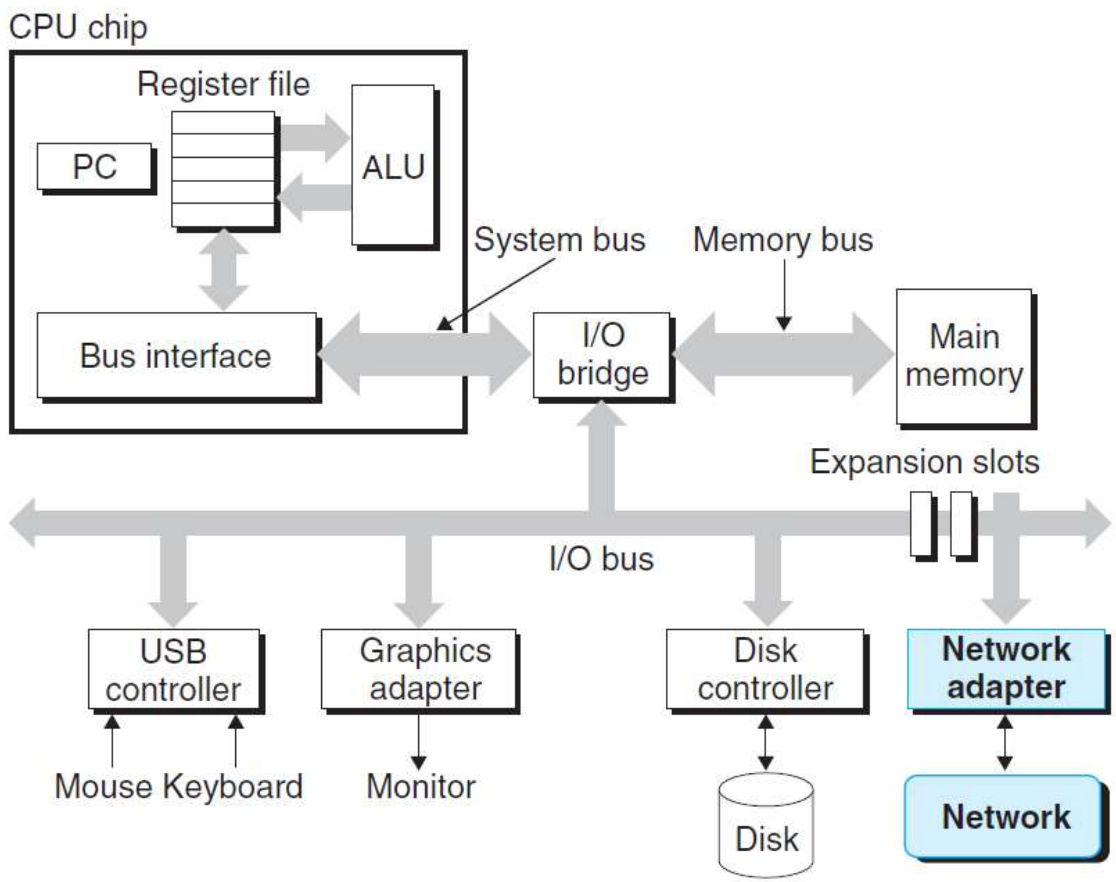
Figure 1.13
Process virtual address space. (The regions are not drawn to scale.)



Systems Communicate with Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks.

Figure 1.14
A network is another I/O device.



How is a CPU made?

https://youtu.be/qm67wbB5GmI