

Intro to Computer Systems



Introduction to Computer Systems
By
Bhimashankar Takalki

Course Objectives

- **The better knowledge of computer systems, the better programming.**

Computer System	C Programming Language
Computer architecture CPU (Central Processing Unit) IA32 assembly language	Introduction to C language
	Compiling, linking, loading, executing
Physical main memory MMU (Memory Management Unit)	Virtual memory space
Memory hierarchy Cache	Dynamic memory management
	Better coding – locality
Reliable and efficient programming for power programmers (to avoid strange errors, to optimize codes, to avoid security holes, ...)	

Unit Learning Objectives

- List the four phases in the compilation system.
- List the four major hardware components.
- List the three major components in CPU.
- Explain the logical concepts of main memory.
- List the two steps in an instruction cycle.
- Explain briefly for what and how PC is used.
- Explain briefly what locality means.
- Give an example code that can be executed fast by using cache memory.
- Define what a process is.
- Explain how process and thread are different.
- Define what multi-processing is.
- Distinguish virtual memory and physical memory.

- Explain what heap in the virtual memory space is.
- Explain what stack in the virtual memory space is.
- Name the hardware component that translates virtual addresses to physical addresses.
- ...

Unit Contents

- What is information?
- Programs and compilation systems
- CPU
- Cache
- Memory hierarchy
- Operating systems
- Communications

Introduction

- A computer system consists of

1. H...
2. S...

that work together to run ...

- Hardware affects the CORRECTNESS and PERFORMANCE of programs.
- Software helps hardware to perform its tasks in a systematic way.

1. What is Information?

- A source program (or source file) – a sequence of bits?

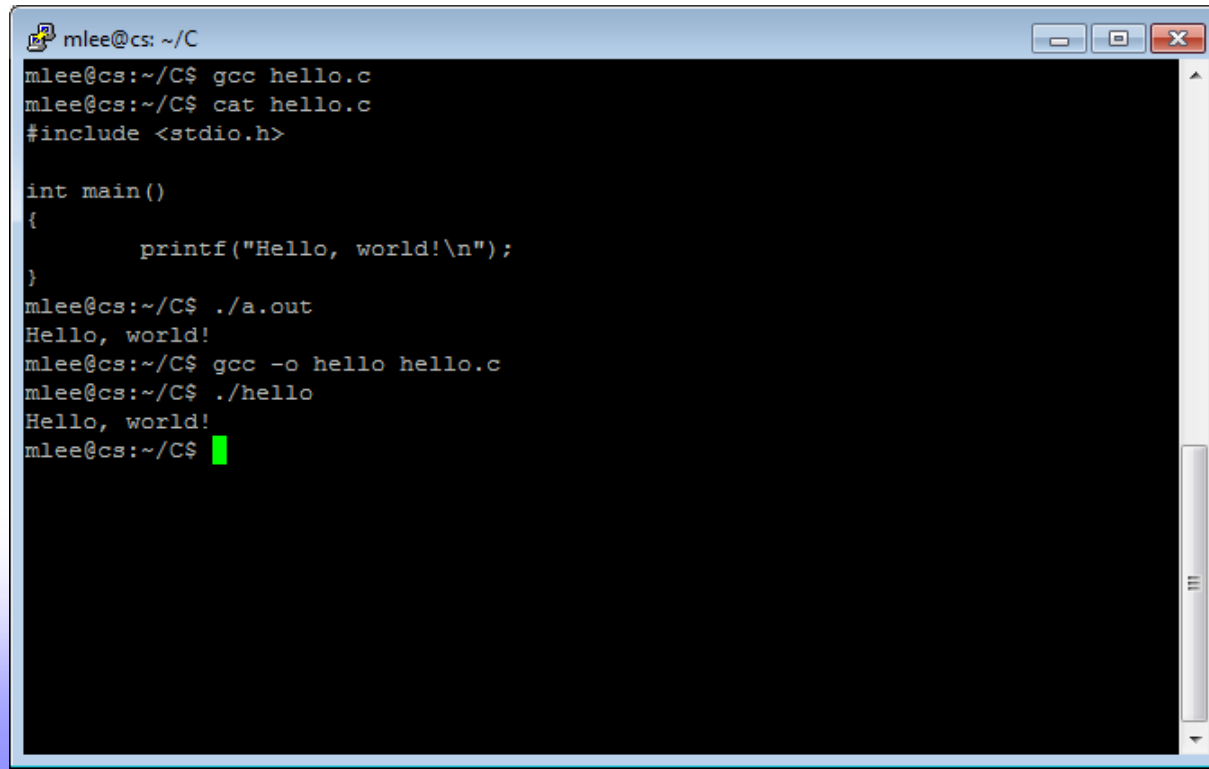
```
#include <stdio.h>
int main()
{
    printf("Hello, world\n");
}
```

- Bits ???
- Bytes ???
- ASCII code ???
 - For each character, a unique byte-sized integer is assigned.
- Text file ???
- Binary file ???
- But files are files for operating systems.

- Where are files (or data) stored in a computer system?
- How is information different from data?
- The only thing that distinguishes different data objects is the **context** in which we view. A data object can be interpreted in different ways.
 - E.g., the same sequence of bytes might represent
 - integer,
 - floating-point number (???),
 - character string,
 - machine instruction, ...
 - As programmers, we need to understand **machine representations of numbers**. Details in B&O 2.

2. Programs and Compilation Systems

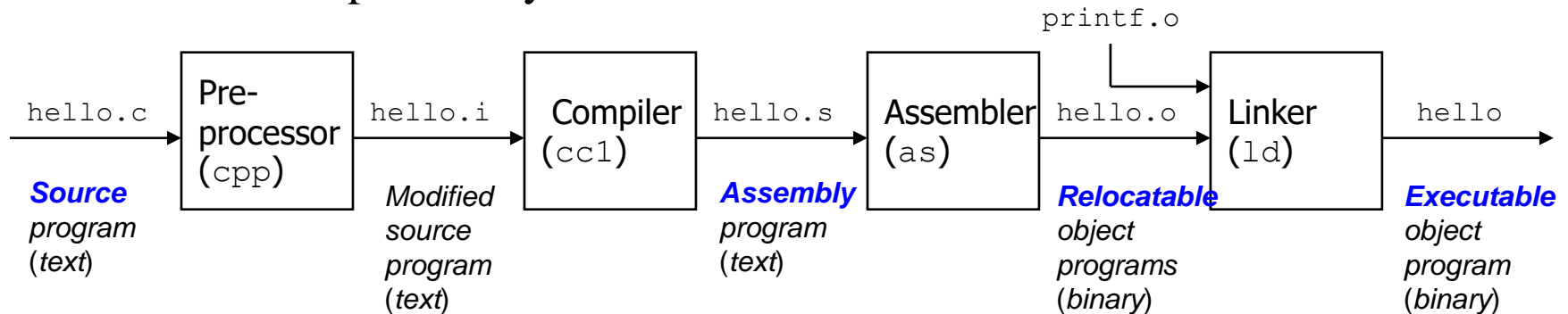
- Can a computer system understand a high-level C program?
- A high-level C program → translated to → low-level *machine-language instructions* packaged in a form called *executable object program* and stored as a binary file. Object programs are also referred as *object files*.
- How to translate?
- Compilers
 - gcc (or cc)

A terminal window titled 'mlee@cs: ~/C' with standard window controls. It shows the process of compiling a C program 'hello.c' using 'gcc', viewing the source code with 'cat', and running the resulting executable 'a.out' and 'hello'. The output of both runs is 'Hello, world!'.

```
mlee@cs: ~/C
mlee@cs:~/C$ gcc hello.c
mlee@cs:~/C$ cat hello.c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}
mlee@cs:~/C$ ./a.out
Hello, world!
mlee@cs:~/C$ gcc -o hello hello.c
mlee@cs:~/C$ ./hello
Hello, world!
mlee@cs:~/C$
```

■ The compilation system



■ Preprocessing phase

- `#include <stdio.h>`
- `#define PI 3.141593`

■ Compilation phase

■ Assembly phase

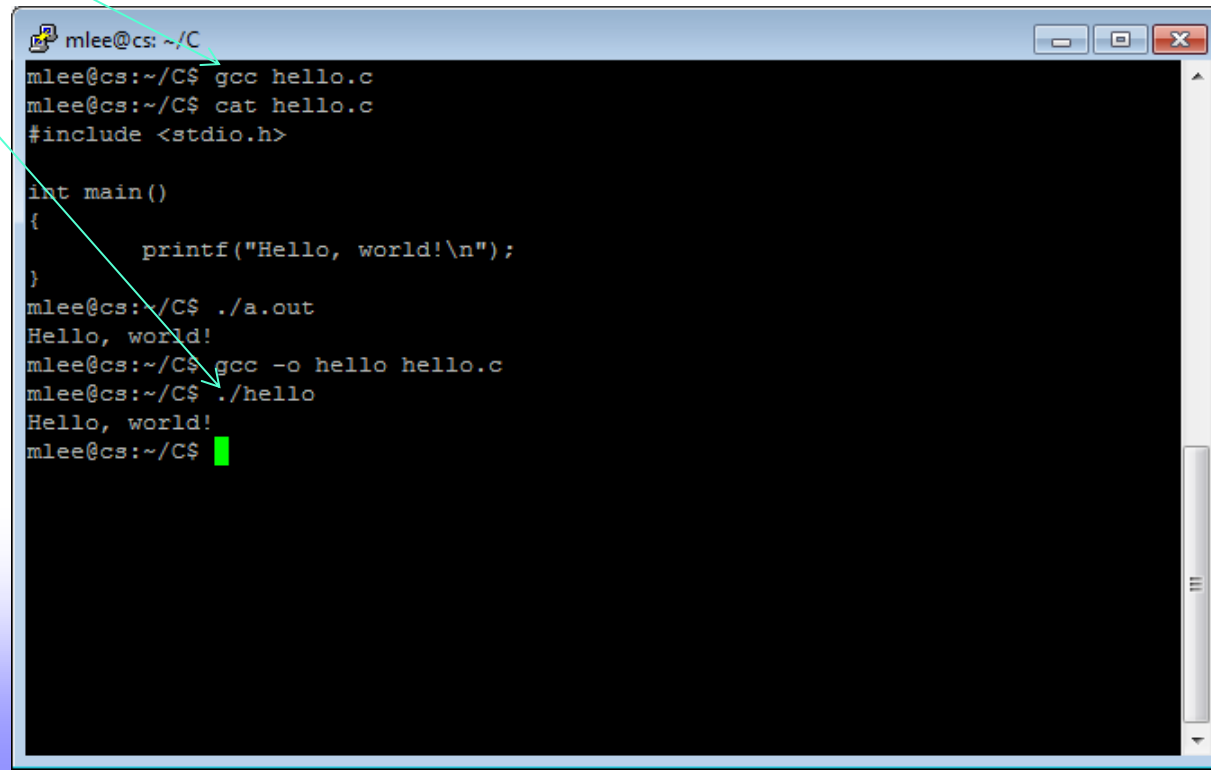
■ Linking phase

- `hello` is ready to be **loaded** into **main memory** and **executed**.
 - Being loaded into main memory
 - Being executed

- Compilation systems try to produce correct and efficient machine codes, but lists the errors which it could not understand
- Reasons to understand how the compilation systems work
 - Optimizing program performance
 - Not easy for them to optimize source codes
 - Understanding link-time errors
 - Especially in the development of a large software system
 - Avoiding security holes
 - Run-time errors

3. CPU

- CPU stands for Central Processing Unit
- Let's discuss how CPU is used to run programs.
- **Shell**, a special application program called a **command-line interpreter**, accepts commands from the user and execute them.
- What happens to hello program when we run?



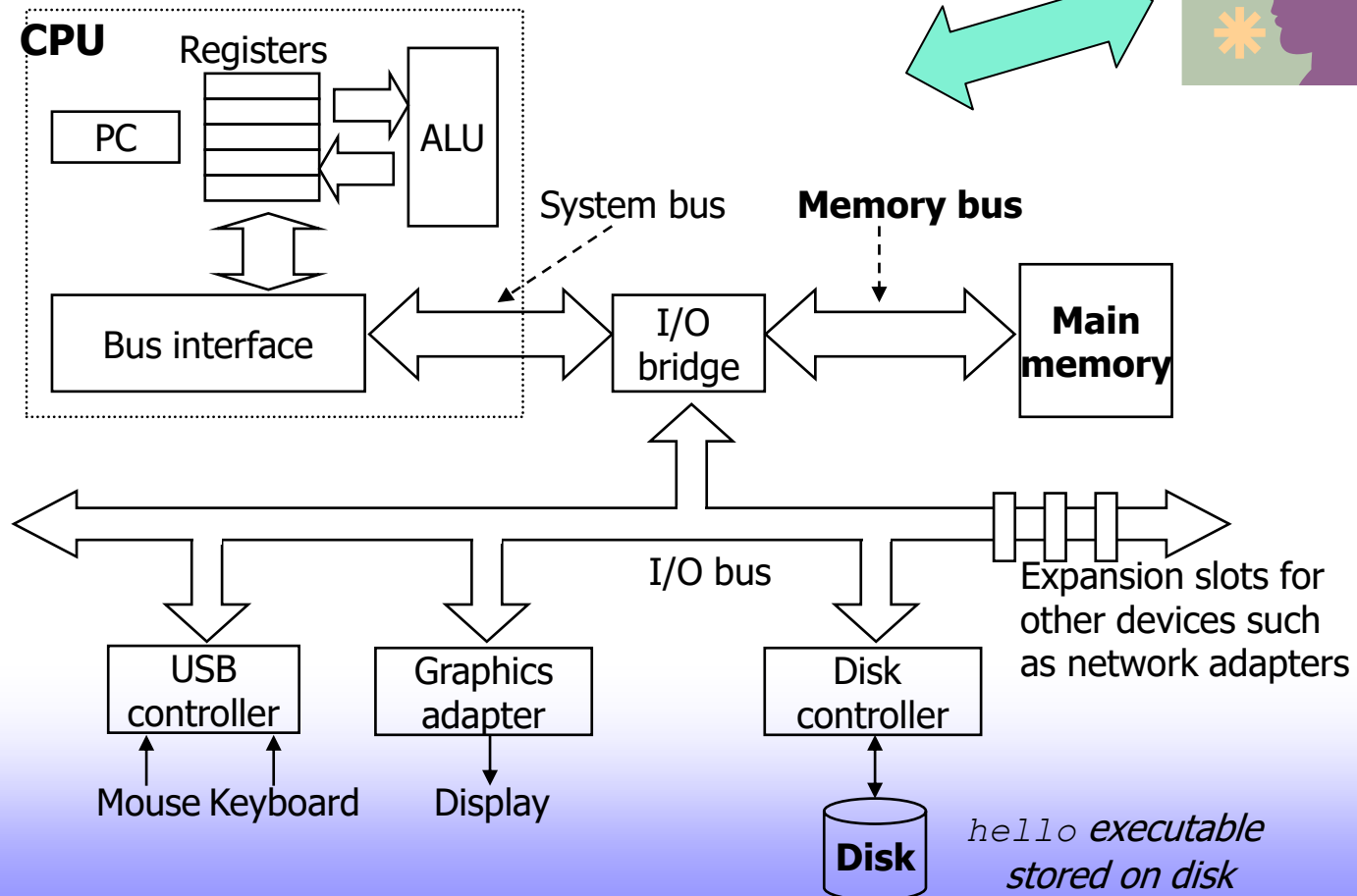
```
mlee@cs: ~/C
mlee@cs:~/C$ gcc hello.c
mlee@cs:~/C$ cat hello.c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}
mlee@cs:~/C$ ./a.out
Hello, world!
mlee@cs:~/C$ gcc -o hello hello.c
mlee@cs:~/C$ ./hello
Hello, world!
mlee@cs:~/C$
```

A terminal window titled 'mlee@cs: ~/C' showing the process of compiling and running a C program. The user enters 'gcc hello.c' to compile, 'cat hello.c' to view the source code, and './a.out' to run the program. The source code is a simple C program that prints 'Hello, world!'. The user then compiles with 'gcc -o hello hello.c' and runs the executable 'hello', which also prints 'Hello, world!'. The prompt 'mlee@cs:~/C\$' is shown at the end of the last line.

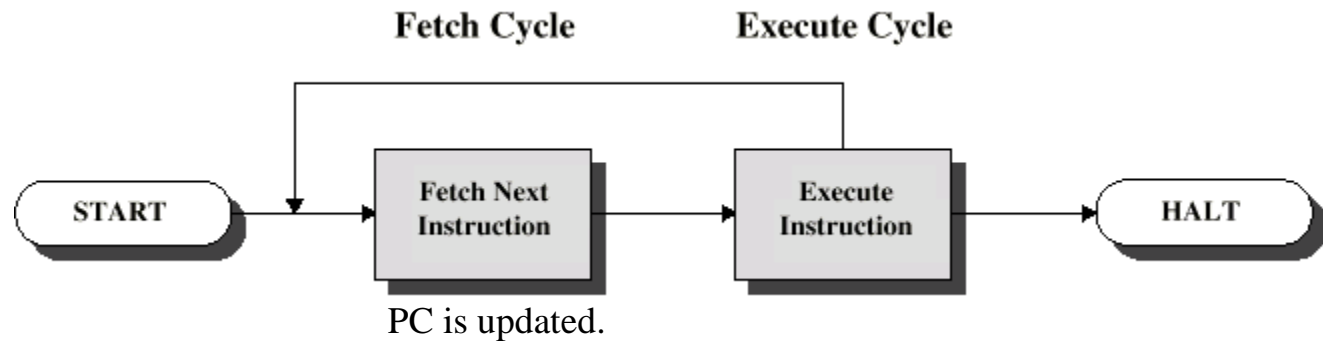
Hardware Organization of a System

- What happens to `hello` program when we run?
 - Need to understand the hardware organization that consists of
 - 4 major components: CPU, main memory, buses, i/o devices



- Buses
 - Carry fixed-size bytes known as **words**
 - 4 bytes (32bits) or 8 bytes (64bits)
- I/O devices
- Main memory
 - DRAM
 - Logically a **linear array of addressable bytes**, each with its own unique **address** starting at zero
- CPU (...), or simply processor
 - **Register** files, or simply registers
 - A register is a word-sized storage device.
 - **PC** (Program Counter)
 - A special register pointing at (contains the address of) some machine-language instruction stored in main memory; increased after the fetch cycle
 - **ALU** (???)
 - Compute new data and address values

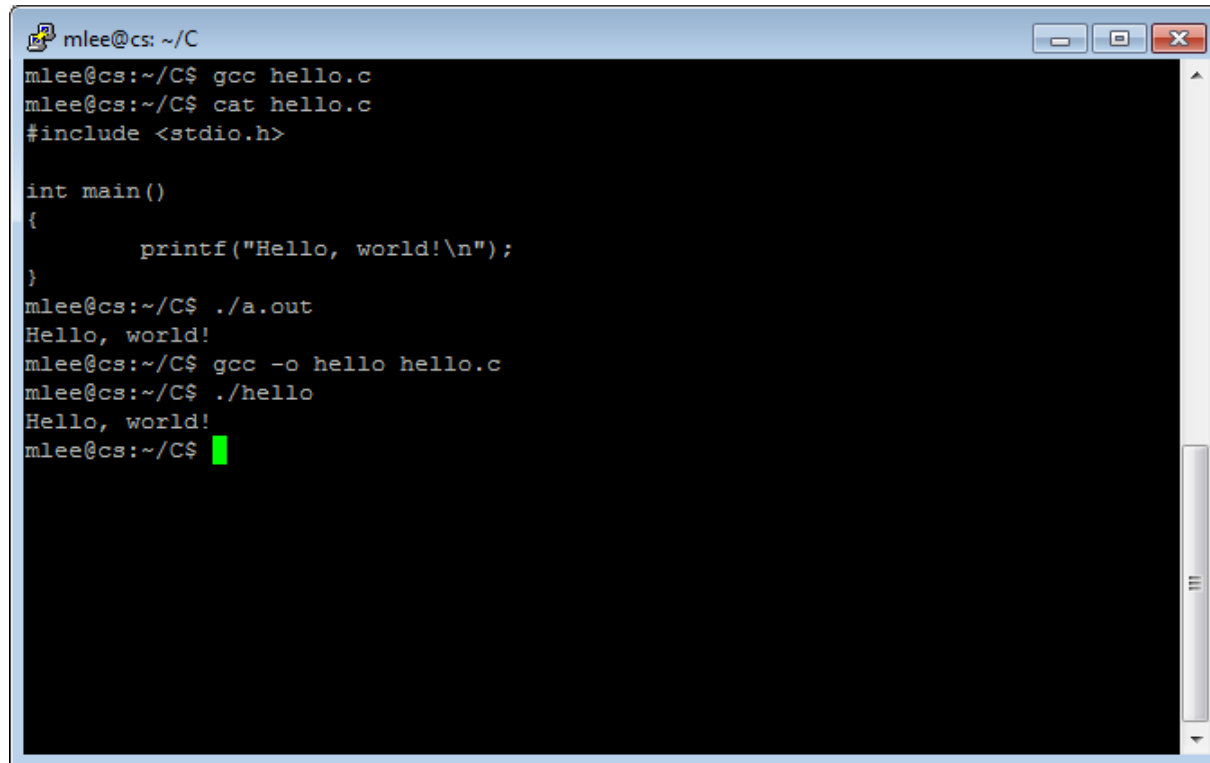
- Only one instruction in main memory, which is pointed by PC, is read (called **fetch**) by CPU and **executed**, not multiple instructions. This is called **instruction cycle**.
- PC is increased after the instruction fetch so that the next instruction is pointed by PC and read by CPU.



- Hence programs have to be loaded into main memory to be executed.
- How to start after the program is loaded into memory?
 - PC must be updated to the address of the first instruction in the program.

Running the hello Program

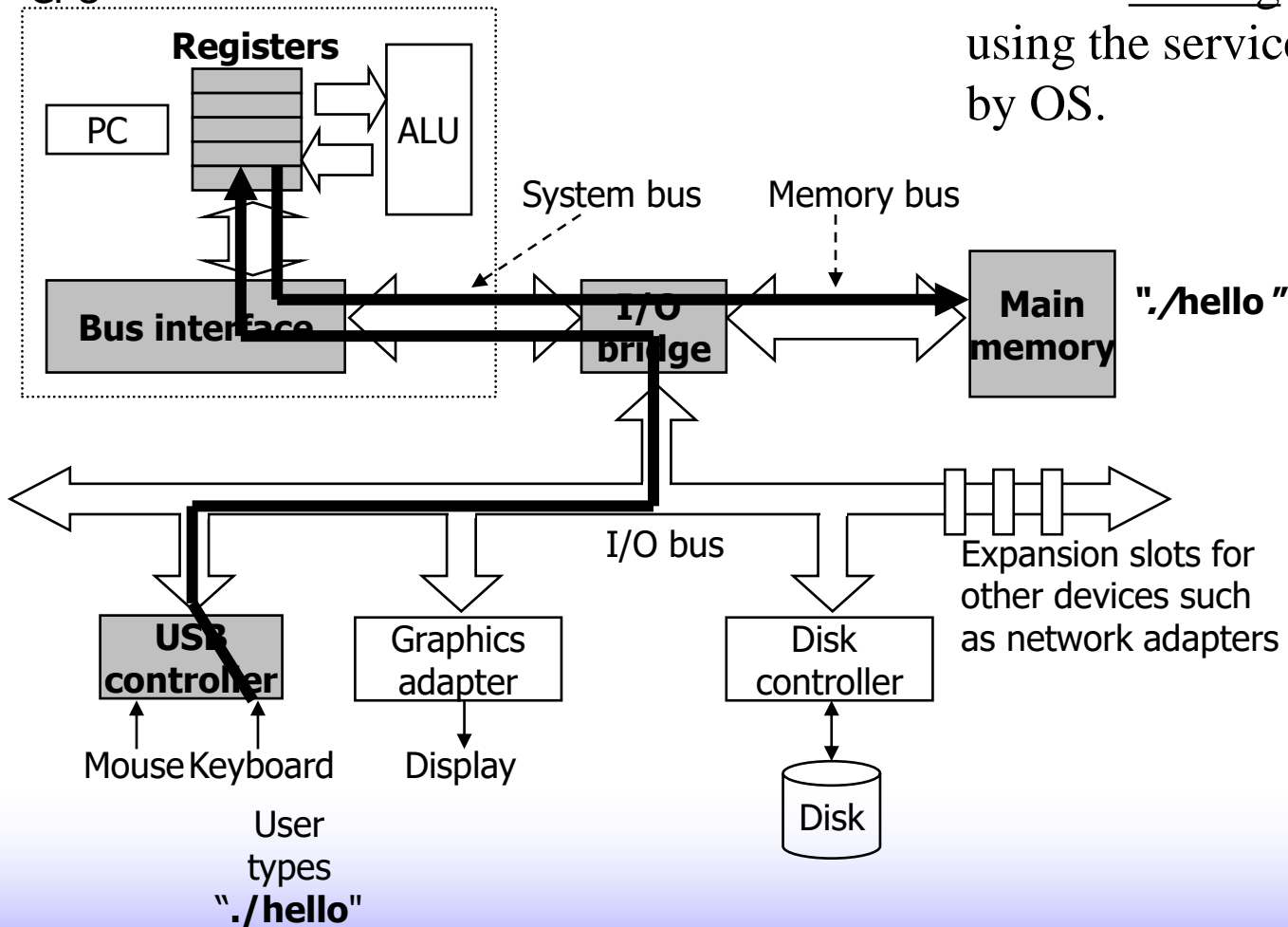
- What happens to hello program when we run?



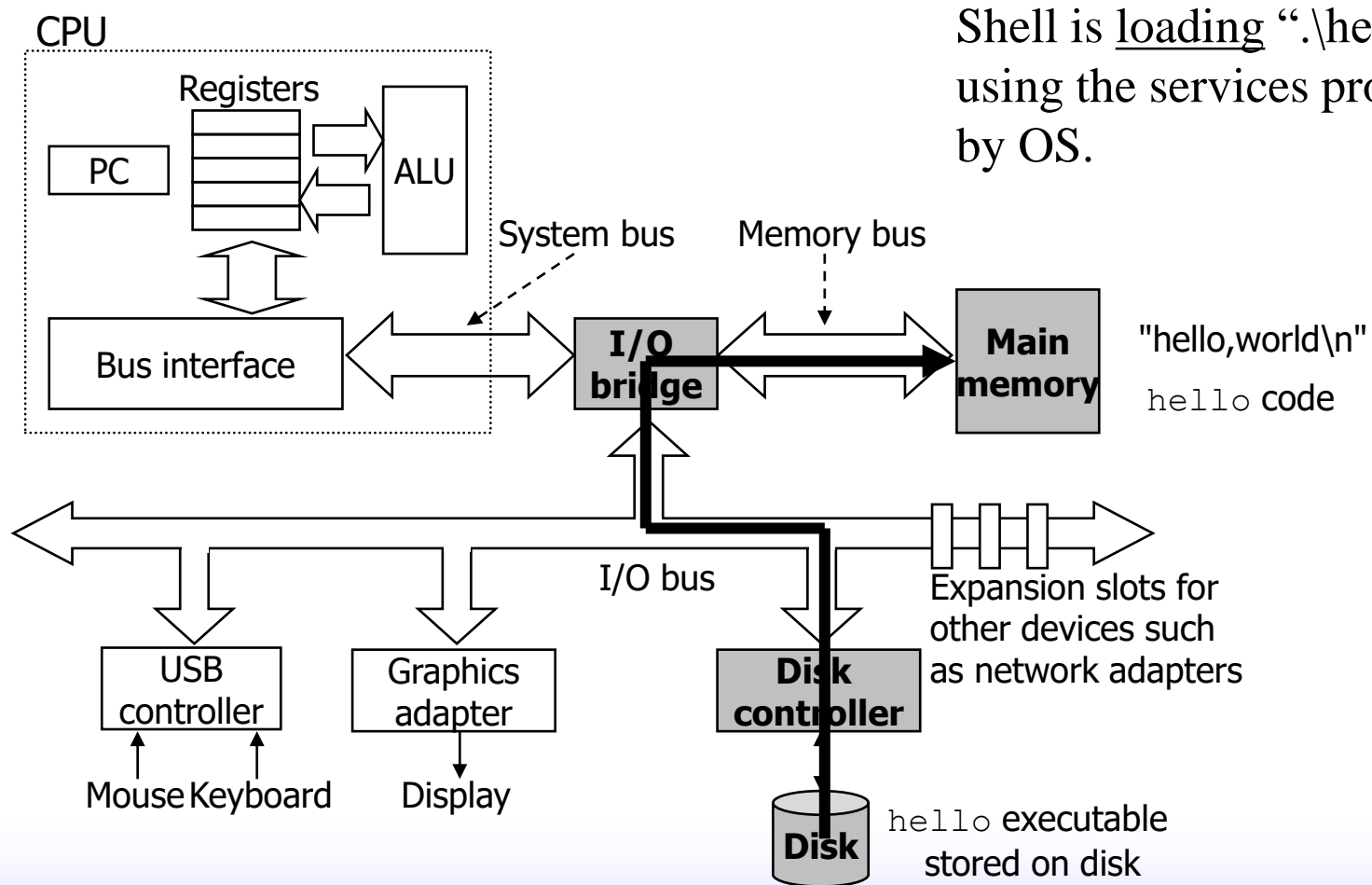
```
mlee@cs: ~/C
mlee@cs:~/C$ gcc hello.c
mlee@cs:~/C$ cat hello.c
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}
mlee@cs:~/C$ ./a.out
Hello, world!
mlee@cs:~/C$ gcc -o hello hello.c
mlee@cs:~/C$ ./hello
Hello, world!
mlee@cs:~/C$
```

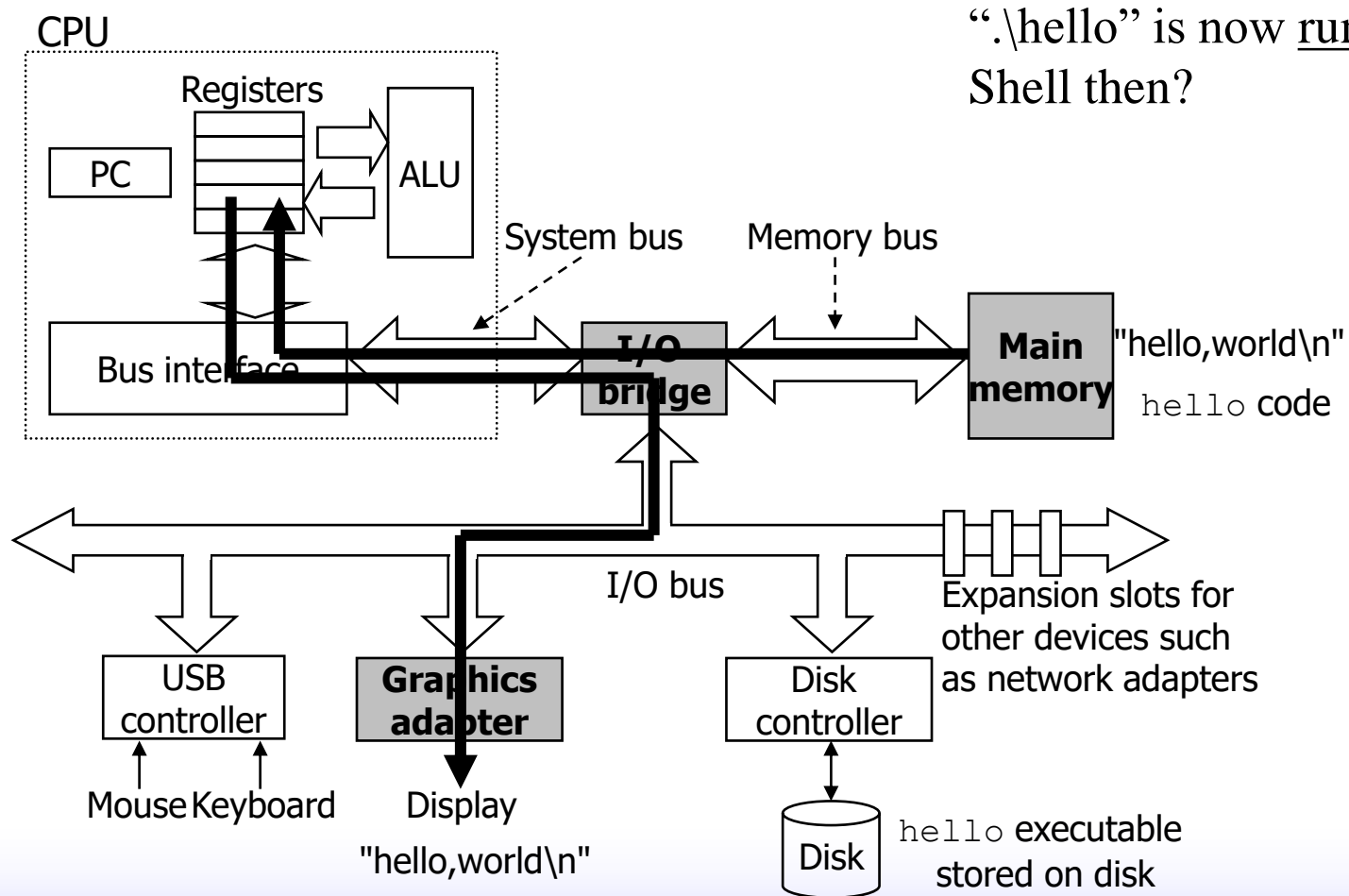

■ CPU



Shell is reading `"/hello"` using the services provided by OS.



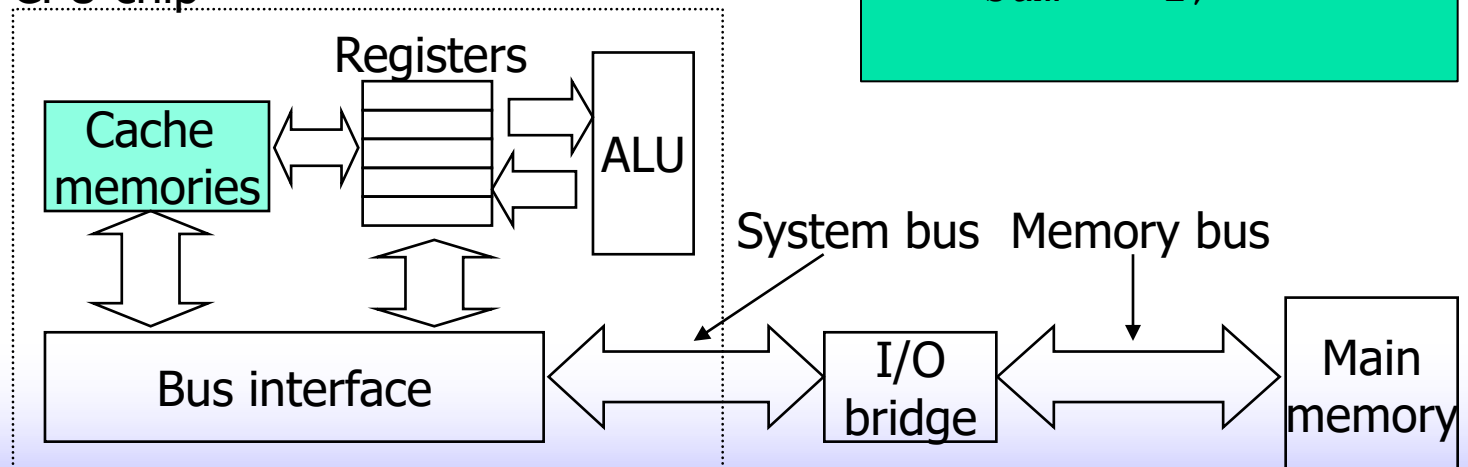
Shell is loading ".\hello" using the services provided by OS.



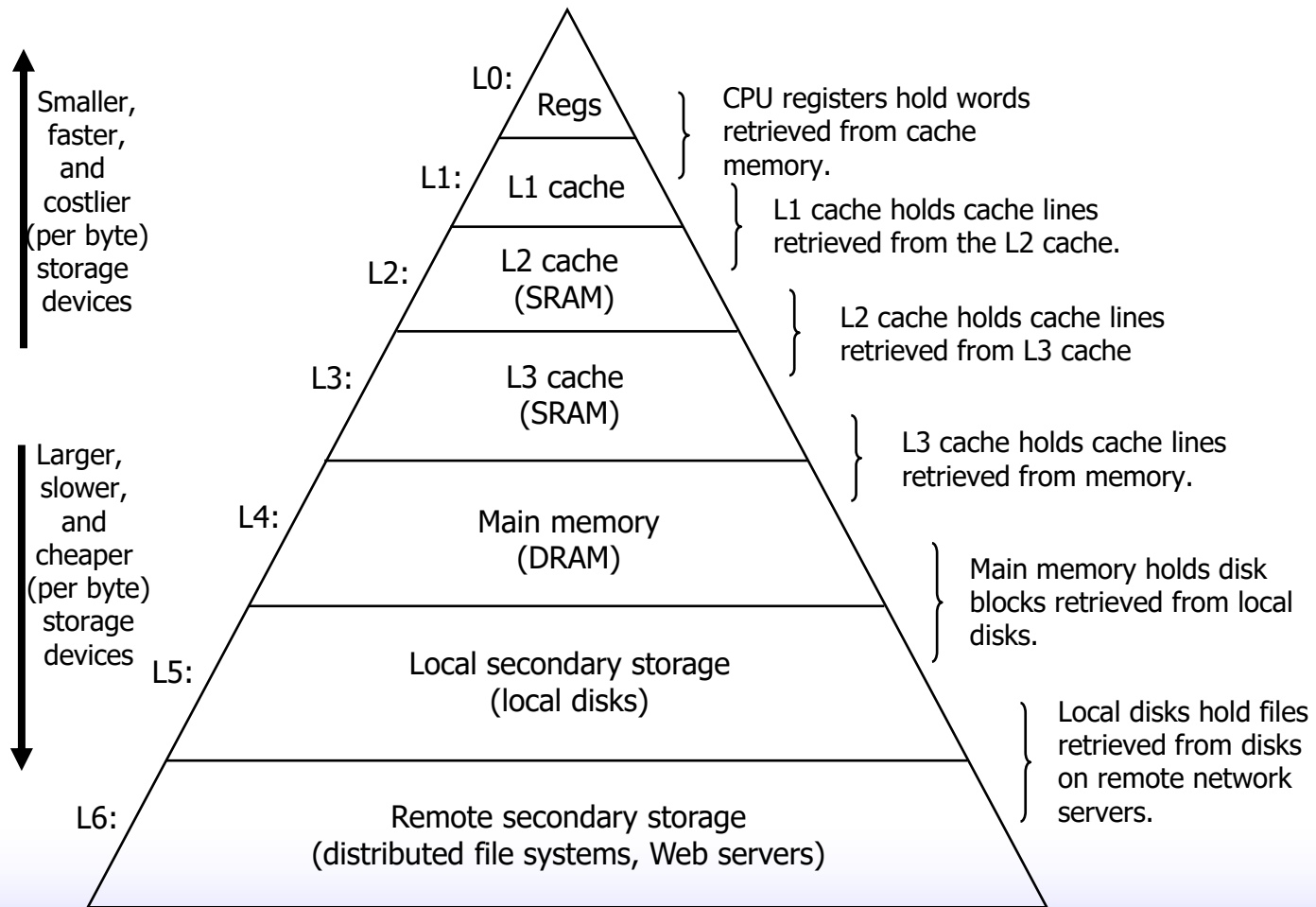
- Fetch time is much longer than execution time.
- Any good idea to solve this **processor-memory gap**?
 - Hint: The processor can read data from registers almost 100 times faster than from memory.

- Smaller faster and closely located storage device called **cache memory**, or simply **cache**
 - Multiple levels could be possible: E.g., L1, L2, L3
 - How is cache helpful? Programs will be loaded into cache, not memory?
 - Initially programs will be loaded into main memory.
 - **Locality** – the tendency for programs to access frequently data and code in localized regions; **any example code?**
 - Important for code optimization

```
for (i=0; i<100; i++)  
    sum += i;
```

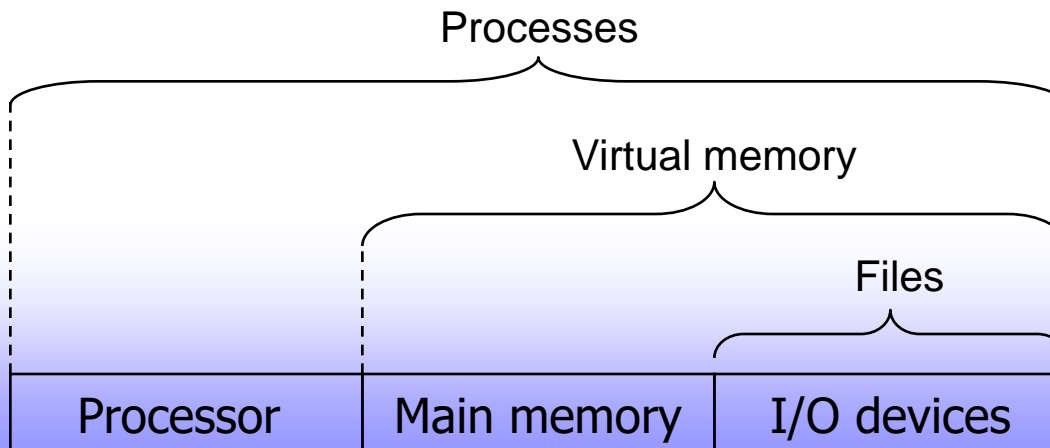
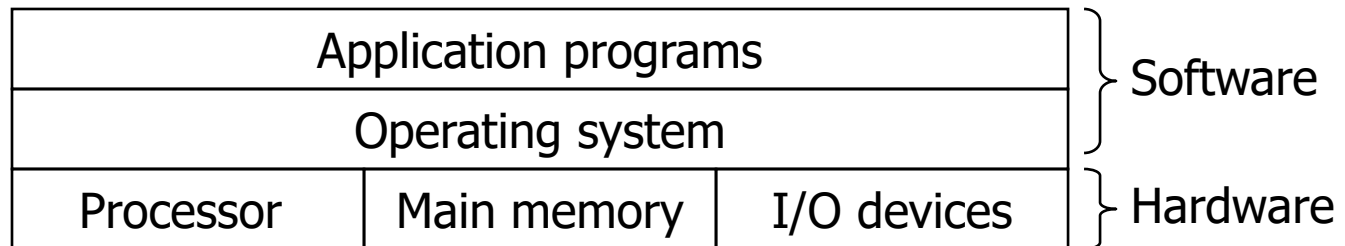


5. Memory Hierarchy



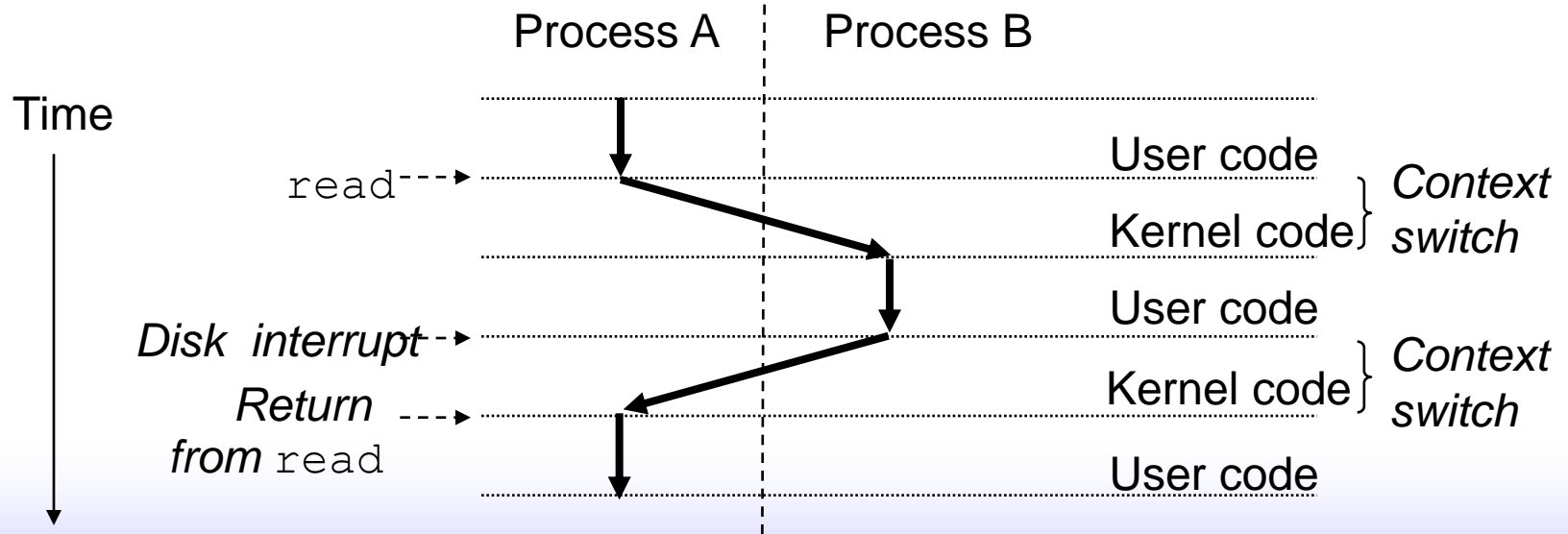
6. Operating Systems

- When the shell loaded and ran the hello program, and when the hello program printed its message, neither program actually accessed the keyboard, display, disk, or main memory directly.
- Then how?
 - Services including I/O are provided by the **operating system**.



Processes

- A **process** is the operating system's abstract for a running program.
- Multiple processes can run on the system? (We assume one CPU core.)
 - Logically yes. In the previous example, shell and hello programs.
 - But this is not real **parallel processing**. This is called **multi-processing**.
 - The instructions of one process are interleaved with the instructions of another process, using **context switching**. (all the register values + ...)



Threads

- A process can consist of multiple execution units, called **threads** or light-weight processes, each running in the context of the process and sharing the same code and global variables, but different local variables.
- Very important programming model. Any good example application?

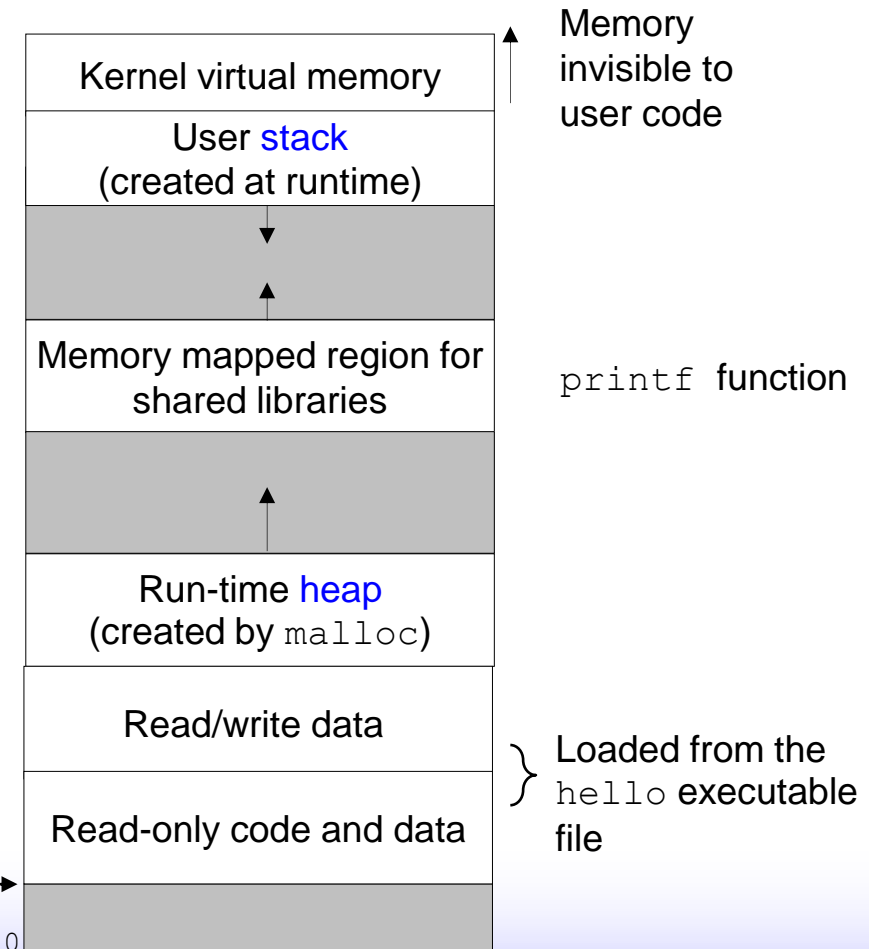


Virtual Memory

- A program includes instructions and data, and they will be loaded into memory. The addresses of instruction and data are used in the program. E.g., $x = y + 10$;
- Multiple processes should share the fixed-size memory. When a program is loaded next time, the location of the program in memory will be probably different.
- How to assign memory space to each process?
- How to let a process (i.e., machine codes) know its location in memory?
- If processes have to know their location in memory to access variables, programming would become a very difficult job.
- Do you think in which area in memory you would put variables?
- How to solve this problem?
 - Virtual (oac logical) address space and physical address space

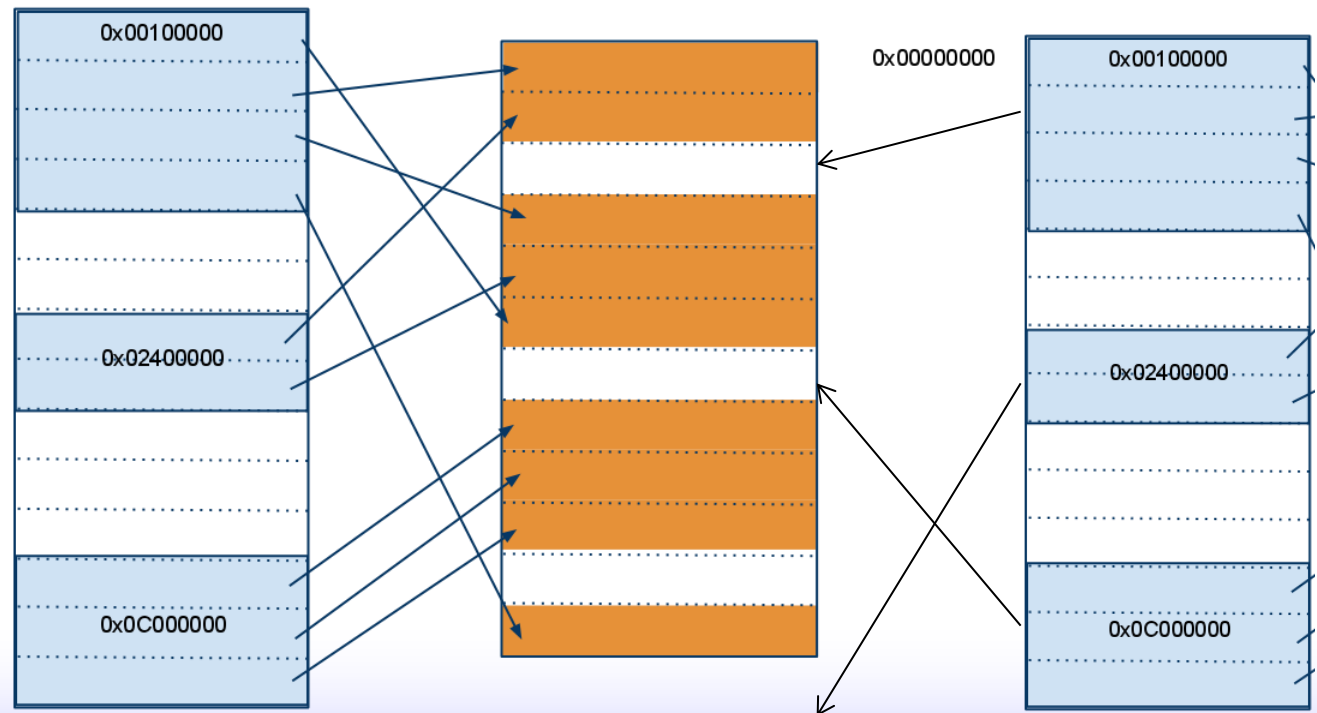
- **Virtual memory** is an abstraction that provides each process with the illusion that it has exclusive use of main memory.
- Each process has the same uniform view of memory, which is known as its **virtual address space**.
- Now compilers can compile programs to machine-level instructions.
(All instructions in programs use virtual addresses.)

0x00048000 (32)
0x00400000 (64)



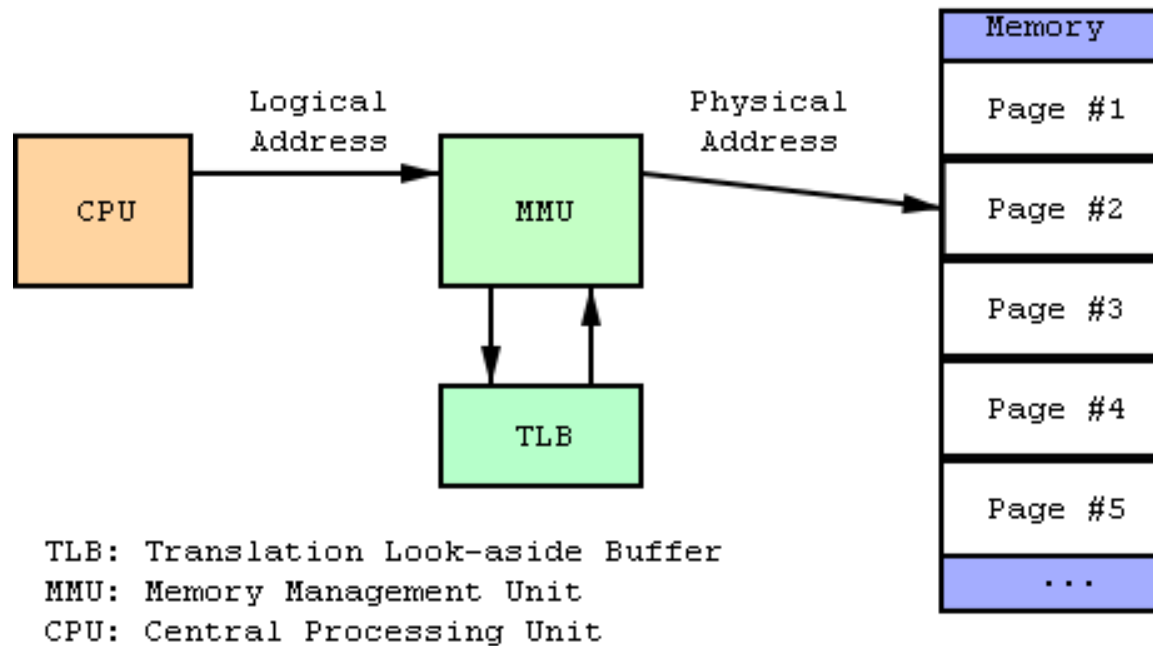
- Program code and data
 - Machine instructions
 - Global variables
- **Heap**
 - Dynamic memory allocation – malloc(), free(), ...
- **Shared libraries**
 - Shared by multiple process – e.g., C standard library, the math library, ...
- **Stack**
 - To allocate memory for parameters and local variables as functions are called
- Kernel virtual memory
 - The part of the operating system
 - Always resident in memory

- **Multiple processes** have program code and data within **the same virtual address space**.
- But they are loaded in different locations in the **physical memory**.



- **Multiple processes** have program code and data within **the same virtual address space**.
- But they are loaded in different locations in the **physical memory**.
- **Here is a very critical problem.** The program code, i.e., machine instructions, uses virtual addresses in the virtual address space, and the actual code and data are loaded in different physical addresses. When an instruction is fetched and executed by CPU, virtual addresses are used. E.g., pointing to a variable uses the virtual address of the variable, which is different from the physical address of the variable in memory. CPU would access wrong location in the physical memory.
- How to solve?
 - **MMU** (Memory Management Unit) – hardware component to translate virtual addresses to physical addresses

- Example



7. Communications

