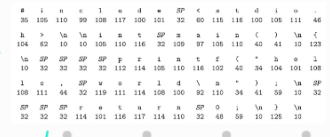
Chapter 1: Introduction

```c
#include <stdio.h>

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

↳ Source program



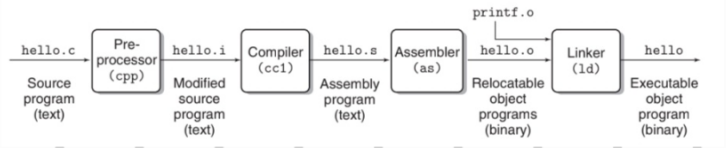
↳ ASCII representation.

↳ All information in a System is represented using bunch of bits the only things that distinguishes different data objects is the context in which we view them.

for ex:

↳ In different context same sequence of bytes can represent integer, floating point, character, string etc.

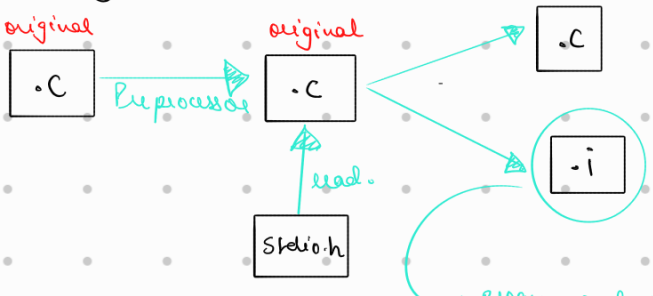
* Different forms of program.



Preprocessing Phase :

↳ modifies the original C program according to directives that begin with the # character.

↳ ex: #include<stdio.h> tells the preprocessor to read the contents of System header file stdio.h and it directly into the program text.



↳ program code with all the preprocessor directives handled / expanded.

Specifically

- Macros (`#define`) → expanded into actual text replacements.
- Header includes (`#include`) → replaced with the entire contents of the included header files.
- Conditional compilation (`#if`, `#ifdef`, `#ifndef`, etc.) → evaluated, and only the active branches remain.
- Comments → removed (since they are for humans, not for the compiler).

2) Compilation phase :

↳ Compiler translates the text file hello.i into the text file hello.s which contains assembly-language Program.

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

3) Assembly phase :

↳ assembler (as) translate hello.s into machine language instructions. [relocatable object program] i.e hello.o

↳ this file is binary.

4) Linking phase :

↳

> • *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.

Q. Where does the actual code for printf resides and how it is executed?

↳ printf is declared in stdio.h its declaration is of the form

↳ `int printf(const char* format, ...);`

↳ but this is just the declaration the actual code code for printf is neither present in our code or the stdio.h

↳ It lives in the [standard library implementation Provided by the System. for ex:

1. Linux: inside glibc (GNU C library)
2. Windows: MSVC runtime library
3. mac: darwin libc

↳ During assembly for printf we might have call printf but here this is just symbol

↳ During linking the linker looks up printf in [libc.a or libc.so] or [msvcrt.lib or msvcrt.dll]

Linux — Windows

4. How is `printf` actually implemented?

`printf` is a very large and complex function in libc. It has to:

1. Parse the format string (`"%d %f %s"` etc.).
2. Handle variable arguments (using `va_list` and macros like `va_arg`).
3. Convert values (integers, floats, strings) into text.
4. Send the text to the correct output stream (`stdout`), which ultimately involves a system call (`write` on Unix-like systems, `WriteFile` on Windows).

So the flow looks like this:

bash — Copy code

```
your code → printf(...) → libc implementation → system call → OS kernel → screen/terminal
```

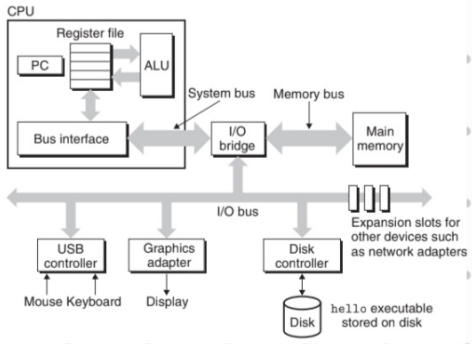
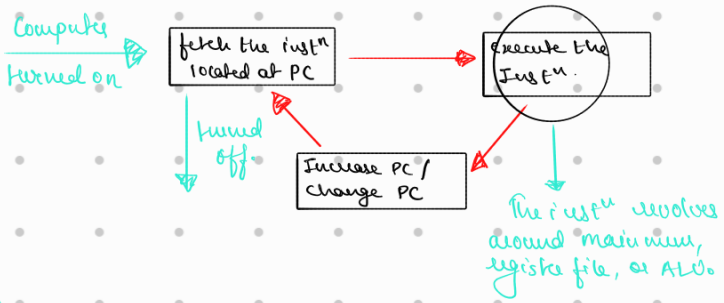


Figure 1.4 **Hardware organization of a typical system.** CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

① Buses : Carry bytes of information between components these are fixed-size chunks of bytes known as words

↕ all other components in buf.

② processor : flow of processor.



Some simple Instⁿ that the processor may carry out.

① load → copy a byte or a word from mm into reg

② store → " " " " " " reg to mm.

③ operate → copy contents of 2 reg to ALU and perform some arithmatic opⁿ

④ Jump → update PC and go to that address to get new Instⁿ.

* Running of a program

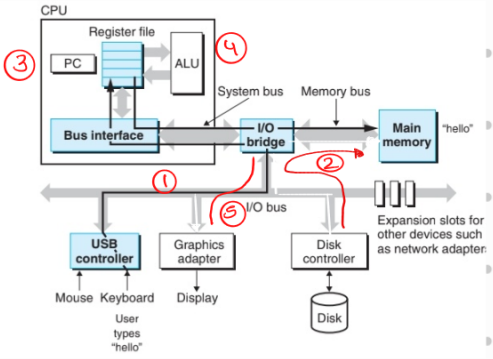


Figure 1.5 **Reading the `hello` command from the keyboard.**

Steps

① user input to run the program

② loading the code into main memory

③ updating PC + applying opⁿ

④ repeating step ③ till completion

⑤ display to user.

* Memory hierarchy

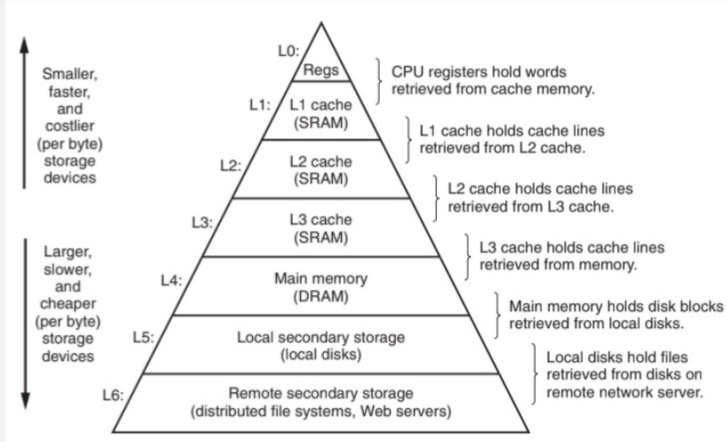


Figure 1.9 **An example of a memory hierarchy.**

* Operating System Manages the hardware.

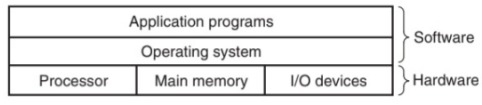


Figure 1.10 **Layered view of a computer system.**

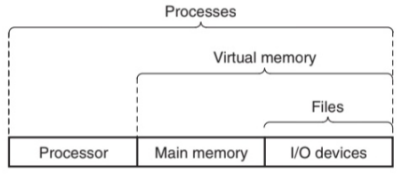


Figure 1.11 **Abstractions provided by an operating system.**

↳ When we run a program like hello world.c the the program doesn't access the keyboard, display, disk or main memory instead it uses the services provided by the OS

The OS has 2 primary purposes.

① protect hardware from misuse by runaway applications

② to provide with a simple mechanism for application to operate on widely different hardware.

↳ OS achieves both the goals via the use of fundamental abstraction shown in fig 1.1.

* Process

↳ helps the application believe that it is the only program running on the system. i.e it believes it has exclusive use of

① processor ② main memory ③ I/O devices.

A process is OS's abstraction of a running prog.

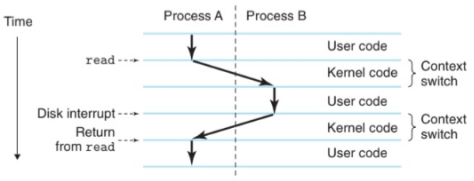
Concurrency allows multiple processes to run simultaneously.

↳ A single processor can perform these multiple processes by the use of Context switching

Context is all the state information required by the process in order to run. Such as PC, register file, contents of main memory.

↳ during Context Switching we save context of one process in main memory and either load an old process from main memory or get a new process (if available).

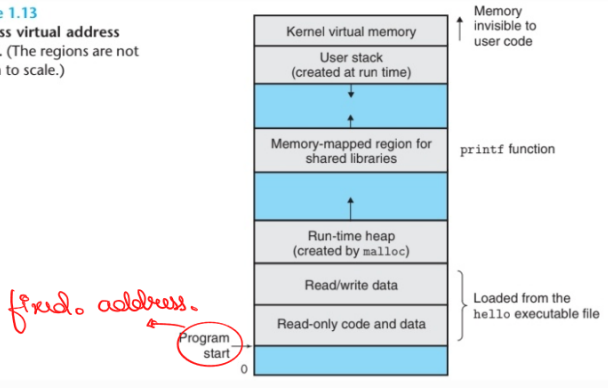
Figure 1.12
Process context switching.



As Figure 1.12 indicates, the transition from one process to another is managed by the operating system *kernel*. The kernel is the portion of the operating system code that is always resident in memory. When an application program requires some action by the operating system, such as to read or write a file, it executes a special *system call* instruction, transferring control to the kernel. The kernel then performs the requested operation and returns back to the application program. Note that the kernel is not a separate process. Instead, it is a collection of code and data structures that the system uses to manage all the processes.

A single process contains multiple execution units called threads.

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



fixed address.

Virtual memory

↳ abstraction of main memory so each process believes that it has the exclusive access to main memory.

↳ fig 1.13 Linux process address space

- *Program code and data.* Code begins at the same fixed address for all processes, followed by data locations that correspond to global C variables. The code and data areas are initialized directly from the contents of an executable object file—in our case, the `hello` executable. You will learn more about this part of the address space when we study linking and loading in Chapter 7.
- *Heap.* The code and data areas are followed immediately by the run-time *heap*. Unlike the code and data areas, which are fixed in size once the process begins running, the heap expands and contracts dynamically at run time as a result of calls to C standard library routines such as `malloc` and `free`. We will study heaps in detail when we learn about managing virtual memory in Chapter 9.
- *Shared libraries.* Near the middle of the address space is an area that holds the code and data for *shared libraries* such as the C standard library and the math library. The notion of a shared library is a powerful but somewhat difficult concept. You will learn how they work when we study dynamic linking in Chapter 7.
- *Stack.* At the top of the user's virtual address space is the *user stack* that the compiler uses to implement function calls. Like the heap, the user stack expands and contracts dynamically during the execution of the program. In particular, each time we call a function, the stack grows. Each time we return from a function, it contracts. You will learn how the compiler uses the stack in Chapter 3.
- *Kernel virtual memory.* The top region of the address space is reserved for the kernel. Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code. Instead, they must invoke the kernel to perform these operations.

Files.

↳ Sequence of bytes, Nothing More nothing less.

↳ Every I/O device

↳ disks ↳ keyboard ---- even networks are modelled as files.

↳ all input outputs are performed by reading or writing into some file.