

 master ▾

...

[bestJavaer / computersystem / csapp-basic.md](#)



cxuan 增加部分目录 

 History

 0 contributors

# 深入理解计算机系统

- 深入理解计算机系统
  - 什么是计算机系统
    - 你想要什么
    - 一段简单的程序
    - 为什么是 C
    - 程序被其他程序翻译成不同的形式
    - 你需要理解编译系统做了什么
    - 处理器读取、解释内存中的指令
      - 系统硬件组成
    - 剖析 hello 程序的执行过程
    - 高速缓存是关键

 325 lines (185 sloc) | 26.7 KB

...

- 进程
- 线程
- 虚拟内存
- 文件

# 什么是计算机系统

计算机系统(A computer system) 是由硬件和软件组成的，它们协同工作运行程序。不同的系统可能会有不同实现，但是核心概念是一样的，通用的。

不同的系统有 Microsoft Windows、Apple Mac OS X、Linux 等。

所有的计算机系统都有相似的软件和硬件组成，它们执行相似的功能。

## 你想要什么

首先，问你一个问题，你想成为哪种程序员？

### 为什么要学习计算机科学？

软件工程师分为两种：一种充分理解了计算机科学，从而有能力应对充满挑战的创造性工作；另一种仅仅凭着对一些高级工具的熟悉而勉强应付。

这两种人都自称软件工程师，都能在职业生涯早期挣到差不多的工资。然而，随着时间流逝，第一种工程师不断成长，所做的事情将会越来越有意义且更为高薪，不论是有价值的商业工作、突破性的开源项目、技术上的领导力或者高质量的个人贡献。

全球短信系统每日收发约200亿条信息，而仅仅靠57名工程师，现在的 WhatsApp 每日收发420亿条。

— Benedict Evans (@BenedictEvans) 2016年2月2日

第一种工程师总是寻求深入学习计算机科学的方法，或是通过传统的方法学习，或是在职业生涯中永无止息地学习；第二种工程师通常浮于表面，只学习某些特定的工具和技术，而不研究其底层的基本原理，仅仅在技术潮流的风向改变时学习新的技能。

如今，涌入计算机行业的人数激增，然而计算机专业的毕业生数量基本上未曾改变。第二种工程师的供过于求正在开始减少他们的工作机会，使他们无法涉足行业内更加有意义的工作。对你而言，不论正在努力成为第一种工程师，还是只想让自己的职业生涯更加安全，学习计算机科学是唯一可靠的途径。

23333 然而他们... [pic.twitter.com/XVNYIXAHar](https://pic.twitter.com/XVNYIXAHar)

— Jenna Bilotta (@jenna) 2017年3月4日

这是我最近搜索到的一个很好的开源项目，它的路径是

<https://github.com/keithnull/TeachYourselfCS-CN/blob/master/TeachYourselfCS-CN.md>

## 简而言之

大致按照列出的顺序，借助我们所建议的教材或者视频课程（但是最好二者兼用），学习如下的九门科目。目标是先花100到200个小时学习完每一个科目，然后在你职业生涯中，不时温习其中的精髓。

| 科目       | 为何要学？                                       | 最佳书籍   | 最佳视频                                  |
|----------|---|--|---------------------------------------|
| 编程       | 不要做一个“永远没彻底搞懂”诸如递归等概念的程序员。                  | <a href="#">《计算机程序的构造和解释》</a>                          | Brian Harvey's<br>Berkeley CS 61A     |
| 计算机架构    | 如果你对于计算机如何工作没有具体的概念，那么你所做出的所有高级抽象都是空中楼阁。    | <a href="#">《计算机组成与设计》</a>                             | Berkeley CS 61C                       |
| 算法与数据结构  | 如果你不懂得如何使用栈、队列、树、图等常见数据结构，遇到有难度的问题时，你将束手无策。 | <a href="#">《算法设计手册》</a>                               | Steven Skiena's<br>lectures           |
| 数学知识     | 计算机科学基本上是应用数学的一个“失控的”分支，因此学习数学将会给你带来竞争优势。   | <a href="#">《计算机科学中的数学》</a>                            | Tom Leighton's<br>MIT 6.042J          |
| 操作系统     | 你所写的代码，基本上都由操作系统来运行，因此你应当了解其运作的原理。          | <a href="#">《操作系统导论》</a>                               | Berkeley CS 162                       |
| 计算机网络    | 互联网已然势不可挡：理解工作原理才能解锁全部潜力。                   | <a href="#">《计算机网络：自顶向下方法》</a>                         | Stanford CS 144                       |
| 数据库      | 对于多数重要程序，数据是其核心，然而很少人理解数据库系统的工作原理。          | <a href="#">《Readings in Database Systems》 (暂无中译本)</a> | Joe Hellerstein's<br>Berkeley CS 186  |
| 编程语言与编译器 | 若你懂得编程语言和编译器如何工作，你就能写出更好的代码，更轻松地学习新的编程语言。   | <a href="#">《编译原理》</a>                                 | Alex Aiken's<br>course on<br>Lagunita |
| 分布式系统    | 如今，多数系统都是分布式的。                              | <a href="#">《分布式系统原理与范型》<br/>(中文第二版, 英文第三版)</a>        | MIT 6.824                             |

也就是



## 年轻人，你渴望力量吗？

我也把它里面涉及的中文/英文书籍都下载下来了，公众号回复 计算机基础，即可领取。  
(图中是冯·诺伊曼)

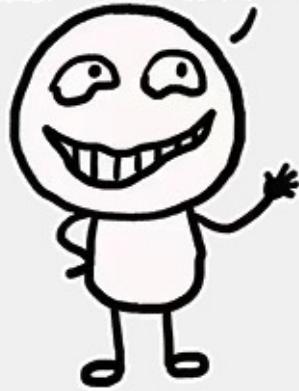
我一直想成为第一种工程师，即使我永远成为不了，我也要越来越靠近它。

回到正题

You are poised for an exciting journey. If you dedicate yourself to learning the concepts in this book, then you will be on your way to becoming a rare “power programmer,” enlightened by an understanding of the underlying computer system and its impact on your application programs.

没错，我就想成为一种 电源程序员

灿烂而又不失诡异的笑



## 一段简单的程序

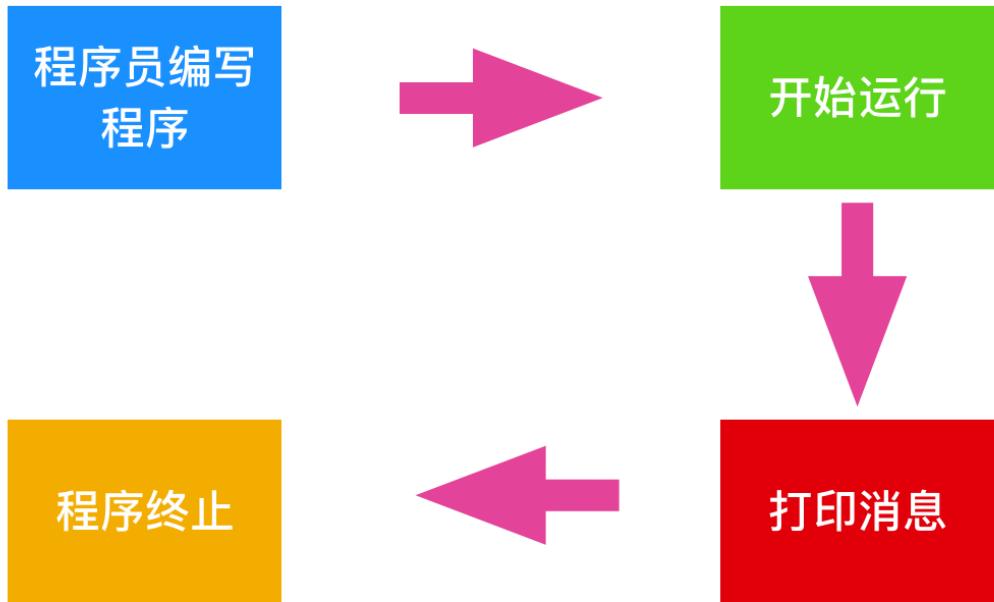
这次真的言归正传了，下面是一道很简单的 C 程序（不要管我的名字是 Java 建设者还是什么，Java 建设者就不能学习 C 了吗？虽然饭碗是 Java，但是 C 才是爸爸啊。）

```
#include <stdio.h>

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

这是用 C 语言输出的一个 Hello,world 程序，尽管它是一个非常简单的程序，但系统的每个部分都必须协同工作才能运行。

这段程序的生命周期就是程序员创建程序、在系统中运行这段程序、打印出一个简单的消息然后终止。



程序员首先在文本中创建这段代码，这个文本又被称为 源文件 或者 源程序，然后保存为 hello.c 文件，源程序实际上就是一个由 0 和 1 组成的位（又称为 比特，即 bit）。8 个 bit 成为一组，称做 字节。每个字节又表示着一个文本字符，这些文本字符通常是由 ASCII 码组成的，下面是 hello.c 程序的 ASCII 码

|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| #   | 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  | p   | r   | i   | n   | t   | f   | (   | "   | h   | e   | 1   |     |
| 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  |     |

### hello.c 文件的 ASCII 码

hello.c 程序以字节顺序存储在文件中，每个字节都对应一个整数值，也就是 8 位表示一个整数。比如第一个字符是 35，那这个 35 是从哪来的呢？这其实是有个 ASCII 码的对照表（因为 ASCII 非常多，可以去 <http://ascii.911cha.com/?year=%23> 官网查询，这里只选取几个作为参考哦）

| 二进制       | 十进制 | 十六进制 | 图形 |
|-----------|-----|------|----|
| 0010 0011 | 35  | 23   | #  |
| 0110 1001 | 105 | 69   | i  |
| 0110 1110 | 110 | 6E   | n  |
| 0110 0011 | 99  | 63   | c  |
| 0110 1100 | 108 | 6C   | l  |
| 0111 0101 | 117 | 75   | u  |
| 0110 0100 | 100 | 64   | d  |
| 0110 0101 | 101 | 65   | e  |

每行都以不可见的 \n 来结尾，它的 ASCII 码值是 10。

注意：只由 ASCII 字符组成的诸如 hello.c 之类的文件称为文本文件。所有其他文件称为二进制文件。

hello.c 的表示方法说明了一个基本思想：系统中所有的信息 --- 包括磁盘文件、内存中的程序、内存中存放的数据以及网络上传输的数据，都是由一串比特表示的。区分不同数据对象的唯一方法是我们读取对象时的上下文，比如，在不同的上下文中，一个同样的字节序列可能表示一个整数、浮点数、字符串或者机器指令。

## 为什么是 C

这里插播一则新闻，为什么我们要学 C 语言？学 Java 用不用懂 C 语言？这里需要聊聊 C 语言的发家史了

C 语言起源于贝尔实验室。美国国家标准学会 ANSI 在 1981 年颁布了 ANSI C 的标准，后来 C 就被标准化了，这些标准定义了 C 语言和一系列函数库，即所谓的 c 语言标准库，那么 C 语言有什么特点呢？

- C 语言与 Unix 操作系统密切关联。C 从一开始就被开发为 UNIX 系统的编程语言，大部分 UNIX 内核（操作系统和核心部分）和工具，动态库都是使用 C 编写

的。UNIX 成为 1970 - 1980 年代最火的操作系统，而 C 成为最火的编程语言

- C 是一种非常小巧，简单的语言。并且 C 语言的简单使他移植性比较强。
- C 语言是为实践目的设计的。

我们上面提到了 C 语言的各种优势，但是 C 语言也并非所有程序员都能熟练掌握并运用的，C 语言的指针经常让很多程序员头疼，C 语言还缺乏对抽象的良好支持，例如类、对象，但是 C++ 和 Java 都解决了这些问题。

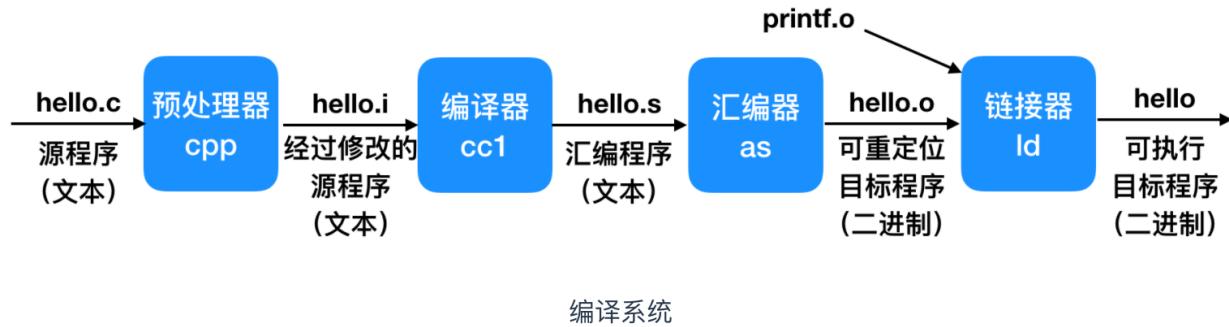
## 程序被其他程序翻译成不同的形式

C 语言程序成为高级语言的原因是它能够读取并理解人们的思想。然而，为了能够在系统中运行 `hello.c` 程序，则各个 C 语句必须由其他程序转换为一系列低级机器语言指令。这些指令被打包作为 可执行对象程序，存储在二进制磁盘文件中。目标程序也称为可执行目标文件。

在 UNIX 系统中，从源文件到对象文件的转换是由 编译器 执行完成的。

```
gcc -o hello hello.c
```

gcc 编译器驱动从源文件读取 `hello.c`，并把它翻译成一个可执行文件 `hello`。这个翻译过程可用如下图来表示



这就是一个完整的 `Hello World` 程序执行过程，会涉及几个核心组件：**预处理器**、**编译器**、**汇编器**、**连接器**，下面我们逐个击破。

- 预处理阶段(Preprocessing phase)，预处理器会根据开始的 `#` 字符，修改源 C 程序。`#include <stdio.h>` 命令就会告诉预处理器去读系统头文件 `stdio.h` 中的内容，并把它插入到程序作为文本。然后就得到了另外一个 C 程序 `hello.i`，这个程序通常是以 `.i` 为结尾。
- 然后是 编译阶段(Compilation phase)，编译器会把文本文件 `hello.i` 翻译成文本 `hello.s`，它包括一段 汇编语言程序(assembler-language program)。这个函数包含 `main` 函数的定义，如下

```
main:  
    subq    $8, %rsp  
    movl    $.LC0, %edi  
    call    puts  
    movl    &0, %eax  
    addq    $8, %rsp  
    ret
```

上面定义中的 2 - 7 描述了一种低级语言指令。汇编语言是非常有用的，因为它能够针对不同高级语言来提供自己的一套标准输出语言。

- 编译完成之后是 汇编阶段(Assembly phase) , 这一步, 汇编器 as 会把 hello.s 翻译成机器指令, 把这些指令打包成 可重定位的二进制程序(relocatable object program) 放在 hello.o 文件中。它包含的 17 个字节是函数 main 的指令编码, 如果我们在文本编辑器中打开 hello.o 将会看到一堆乱码。
- 最后一个是 链接阶段(Linking phase) , 我们的 hello 程序会调用 printf 函数, 它是 C 编译器提供的 C 标准库中的一部分。printf 函数位于一个叫做 printf.o 文件中, 它是一个单独的预编译好的目标文件, 而这个文件必须要和我们的 hello.o 进行链接, 连接器(lld) 会处理这个合并操作。结果是, hello 文件, 它是一个可执行的目标文件(或称为可执行文件) , 已准备好加载到内存中并由系统执行。

## 你需要理解编译系统做了什么

对于上面这种简单的 hello 程序来说, 我们可以依赖 编译系统(compilation system) 来提供一个正确和有效的机器代码。然而, 对于我们上面讲的程序员来说, 编译器有几大特征你需要知道

- 优化程序性能(Optimizing program performance) , 现代编译器是一种高效的用来生成良好代码的工具。对于程序员来说, 你无需为了编写高质量的代码而去理解编译器内部做了什么工作。然而, 为了编写出高效的 C 语言程序, 我们需要了解一些基本的机器码以及编译器将不同的 C 语句转化为机器代码的过程。
- 理解链接时出现的错误(Understanding link-time errors) , 在我们的经验中, 一些非常复杂的错误大多是由链接阶段引起的, 特别是当你想要构建大型软件项目时。
- 避免安全漏洞(Avoiding security holes) , 近些年来, 缓冲区溢出(buffer overflow vulnerabilities) 是造成网络和 Internet 服务的罪魁祸首, 所以我们有必要去规避这种问题

## 处理器读取、解释内存中的指令

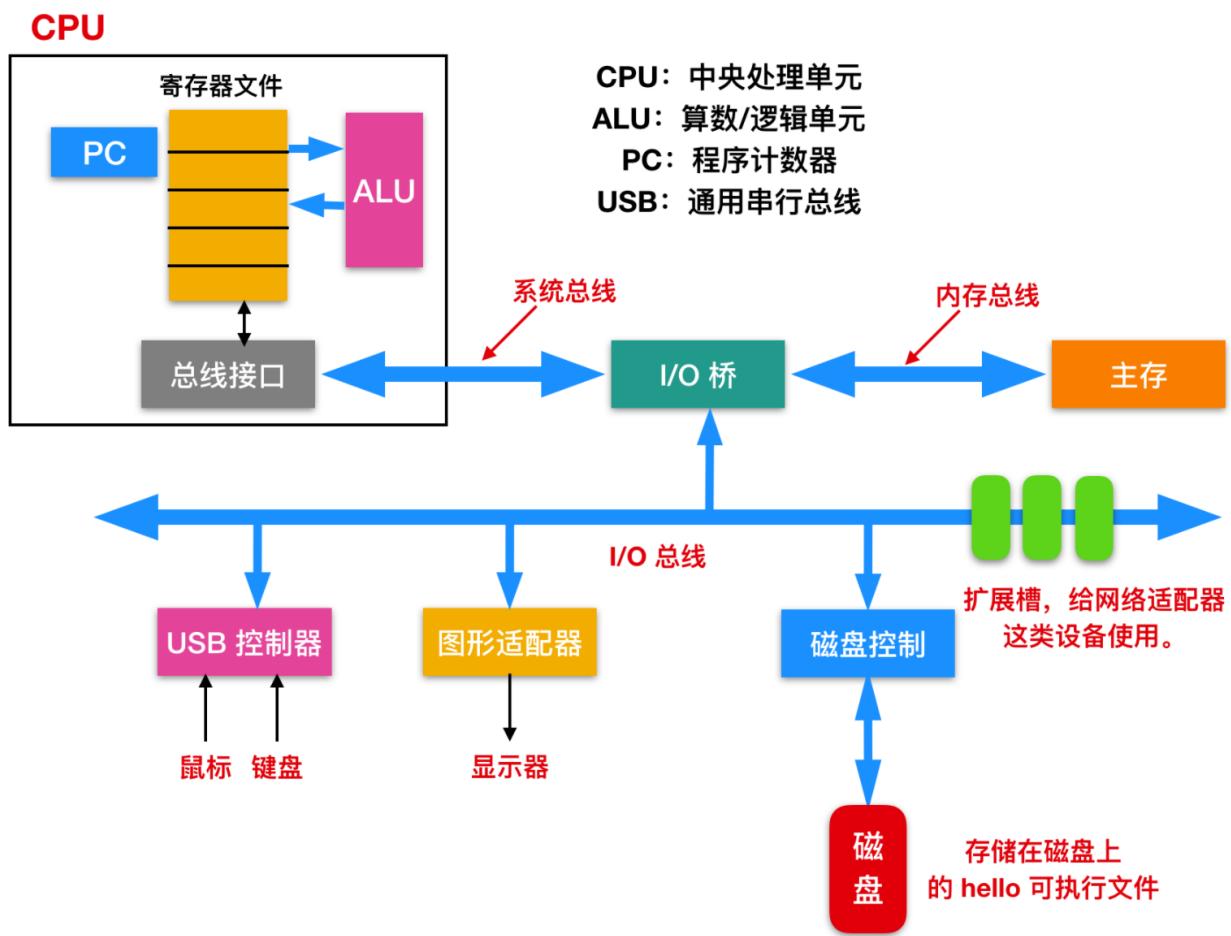
现在, 我们的 hello.c 源程序已经被解释成为了可执行的 hello 目标程序, 它存储在磁盘上。如果想要在 UNIX 操作系统中运行这个程序, 我们需要在 shell 应用程序中输入

```
cxuan $ ./hello  
hello, world  
cxuan $
```

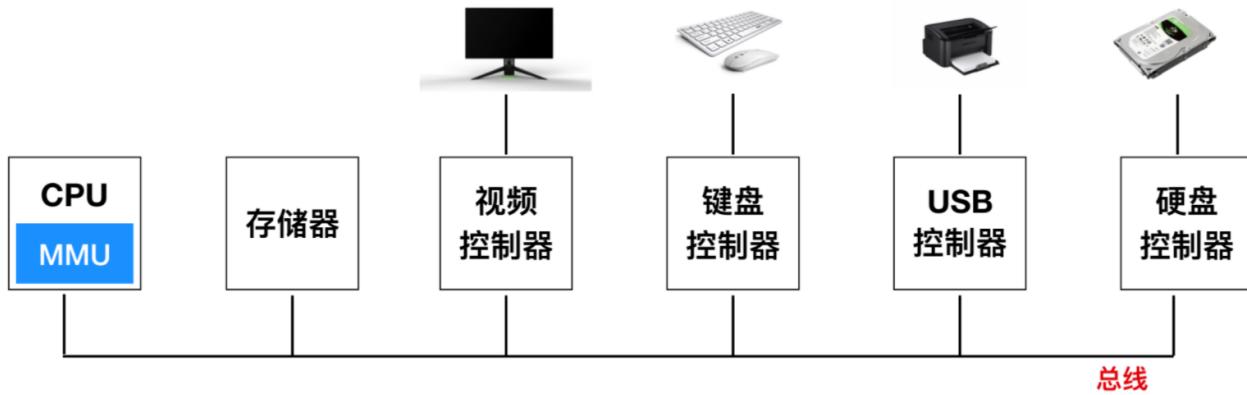
这里解释下什么是 shell，shell 其实就是一个命令解释器，它输出一个字符，等待用户输入一条命令，然后执行这个命令。如果命令行的第一个词不是 shell 内置的命令，那么 shell 就会假设这是一个可执行文件，它会加载并运行这个可执行文件。

## 系统硬件组成

为了理解 hello 程序在运行时发生了什么，我们需要首先对系统的硬件有一个认识。下面这是一张 Intel 系统产品的模型，我们来对其进行解释



- **总线(Buses)**：在整个系统中运行的是称为总线的电气管道的集合，这些总线在组件之间来回传输字节信息。通常总线被设计成传送定长的字节块，也就是 **字(word)**。字中的字节数（字长）是一个基本的系统参数，各个系统中都不尽相同。现在大部分的字都是 4 个字节（32 位）或者 8 个字节（64 位）。



简单个人计算机的组件

- I/O 设备(I/O Devices) : Input/Output 设备是系统和外部世界的连接。上图中有四类 I/O 设备：用于用户输入的键盘和鼠标，用于用户输出的显示器，一个磁盘驱动用来长时间的保存数据和程序。刚开始的时候，可执行程序就保存在磁盘上。

每个I/O 设备连接 I/O 总线都被称为 控制器(controller) 或者是 适配器(Adapter)。控制器和适配器之间的主要区别在于封装方式。控制器是 I/O 设备本身或者系统的主印制板电路（通常称作主板）上的芯片组。而适配器则是一块插在主板插槽上的卡。无论组织形式如何，它们的最终目的都是彼此交换信息。

- 主存(Main Memory) , 主存是一个 临时存储设备 , 而不是永久性存储, 磁盘是 永久性存储 的设备。主存既保存程序，又保存处理器执行流程所处理的数据。从物理组成上说，主存是由一系列 DRAM(dynamic random access memory) 动态随机存储构成的集合。逻辑上说，内存就是一个线性的字节数组，有它唯一的地址编号，从 0 开始。一般来说，组成程序的每条机器指令都由不同数量的字节构成，C 程序变量相对应的数据项的大小根据类型进行变化。比如，在 Linux 的 x86-64 机器上，short 类型的数据需要 2 个字节，int 和 float 需要 4 个字节，而 long 和 double 需要 8 个字节。
- 处理器(Processor) , CPU(central processing unit) 或者简单的处理器，是解释(并执行) 存储在主存储器中的指令的引擎。处理器的核心大小为一个字的存储设备(或寄存器) , 称为 程序计数器(PC) 。在任何时刻，PC 都指向主存中的某条机器语言指令 (即含有该条指令的地址) 。

从系统通电开始，直到系统断电，处理器一直在不断地执行程序计数器指向的指令，再更新程序计数器，使其指向下一条指令。处理器根据其指令集体体系结构定义的指令模型进行操作。在这个模型中，指令按照严格的顺序执行，执行一条指令涉及执行一系列的步骤。处理器从程序计数器指向的内存中读取指令，解释指令中的位，执行该指令指示的一些简单操作，然后更新程序计数器以指向下一条指令。指令与指令之间可能连续，可能不连续（比如 jmp 指令就不会顺序读取）

下面是 CPU 可能执行简单操作的几个步骤

- 加载(Load)：从主存中拷贝一个字节或者一个字到内存中，覆盖寄存器先前的内容
- 存储(Store)：将寄存器中的字节或字复制到主存储器中的某个位置，从而覆盖该位置的先前内容
- 操作(Operate)：把两个寄存器的内容复制到 ALU(Arithmetic logic unit)。把两个字进行算术运算，并把结果存储在寄存器中，重写寄存器先前的内容。

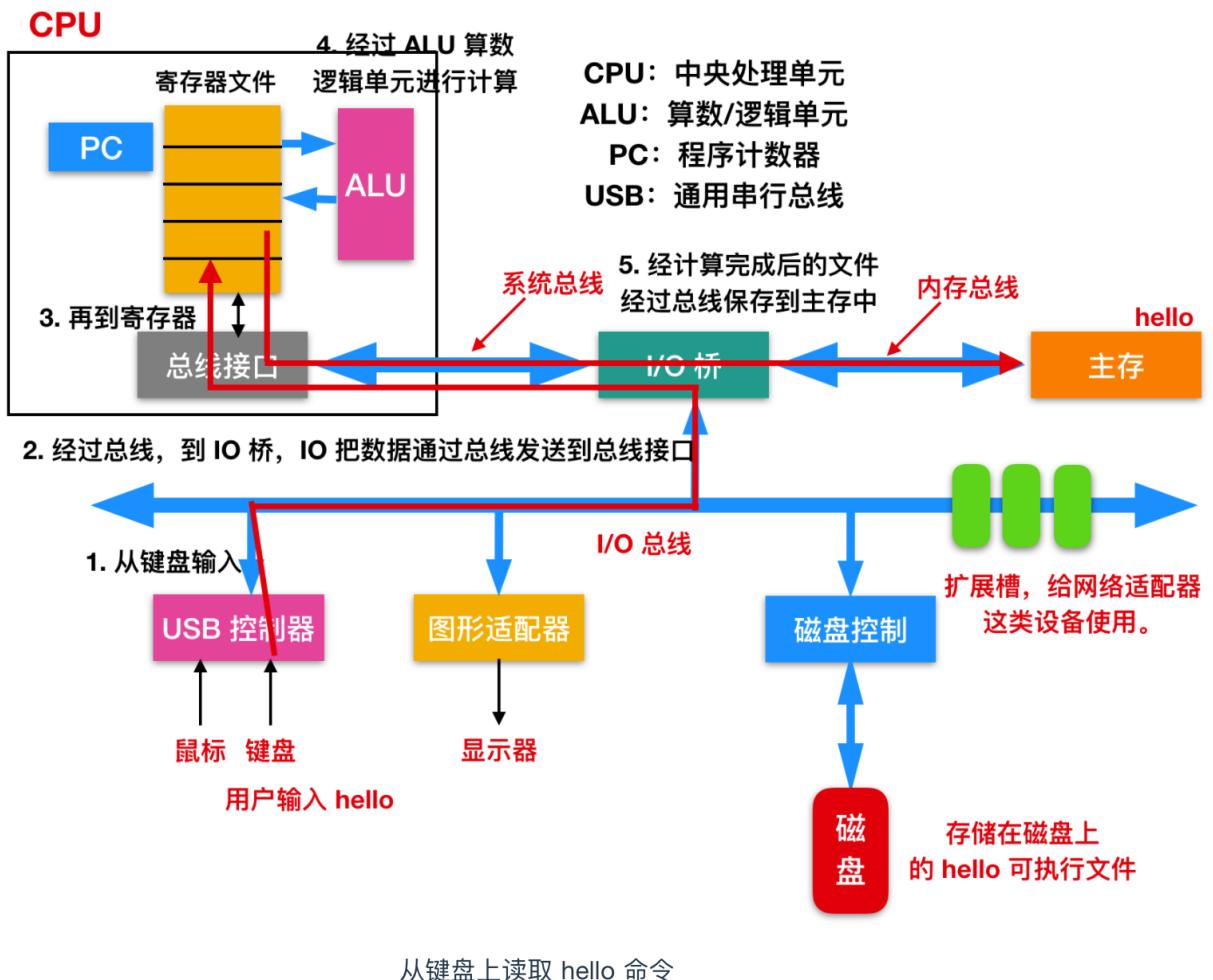
算术逻辑单元 (ALU) 是对数字二进制数执行算术和按位运算的组合数字电子电路。

- 跳转(jump)：从指令中抽取一个字，把这个字复制到 程序计数器(PC) 中，覆盖原来的值

## 剖析 hello 程序的执行过程

前面我们简单的介绍了一下计算机的硬件的组成和操作，现在我们正式介绍运行示例程序时发生了什么，我们会从宏观的角度进行描述，不会涉及到所有的技术细节

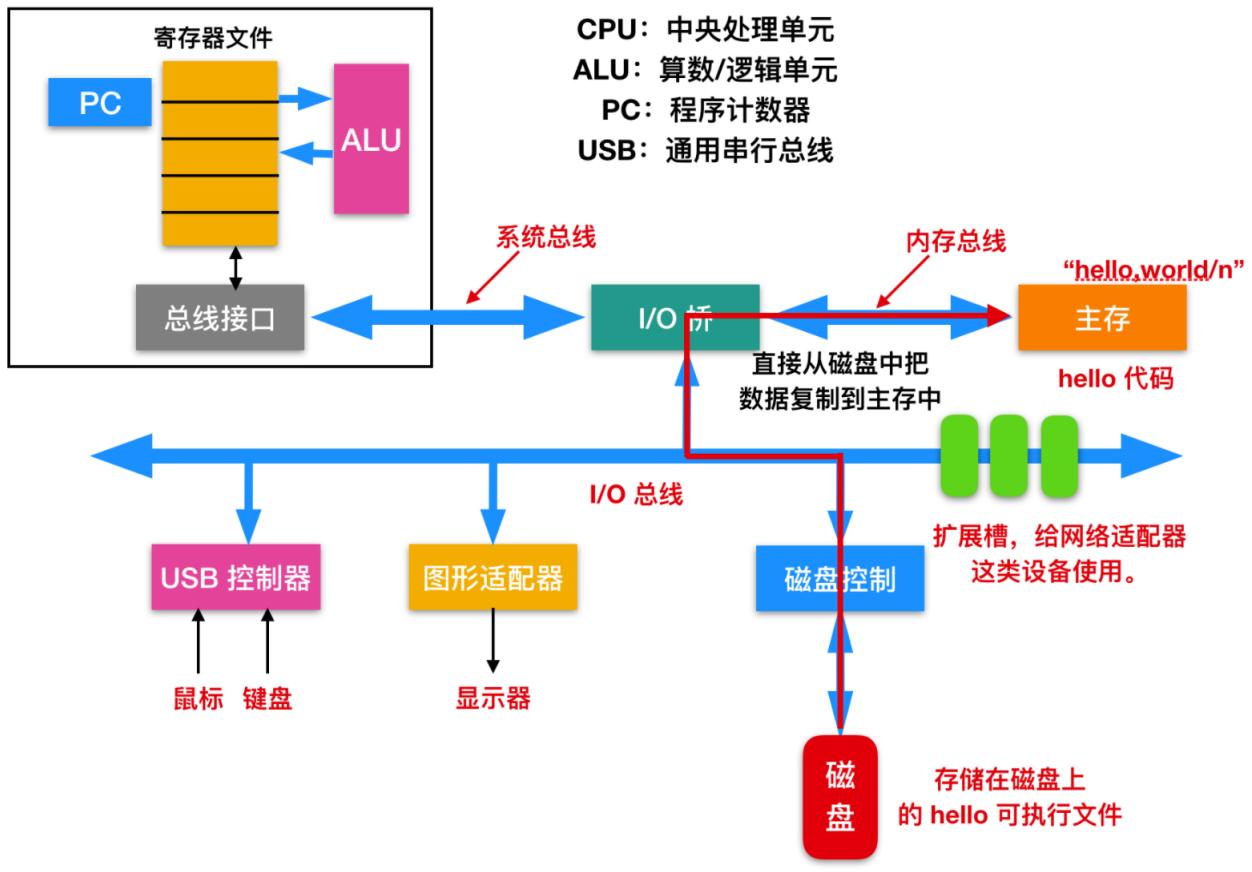
刚开始时，shell 程序执行它的指令，等待用户键入一个命令。当我们在键盘上输入了 `./hello` 这几个字符时，shell 程序将字符逐一读入寄存器，再把它放到内存中，如下图所示



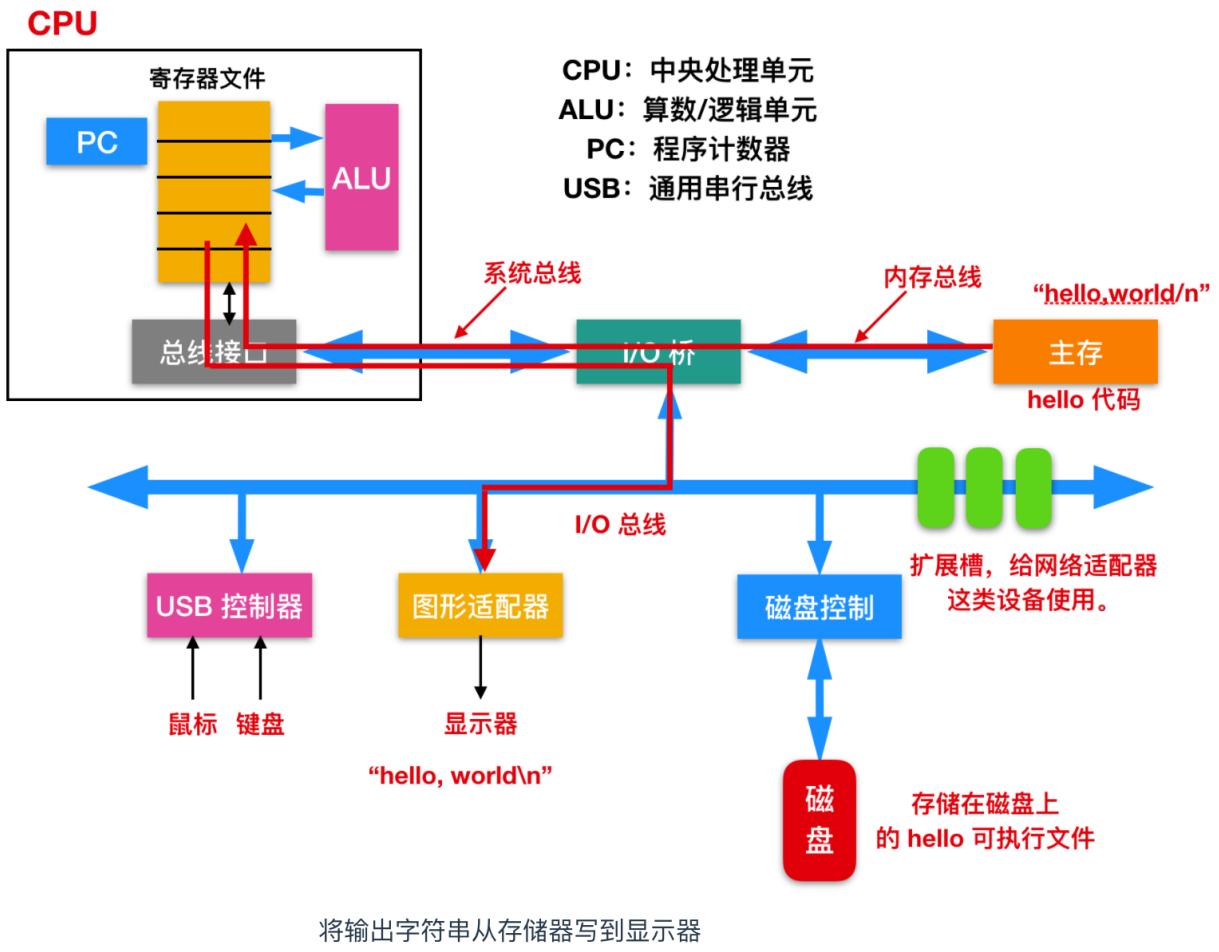
当我们在键盘上敲击 回车键 的时候, shell 程序就知道我们已经结束了命令的输入。然后 shell 执行一系列指令来加载可执行的 hello 文件, 这些指令将目标文件中的代码和数据从磁盘复制到主存。

利用 DMA(Direct Memory Access) 技术可以直接将磁盘中的数据复制到内存中, 如下

## CPU



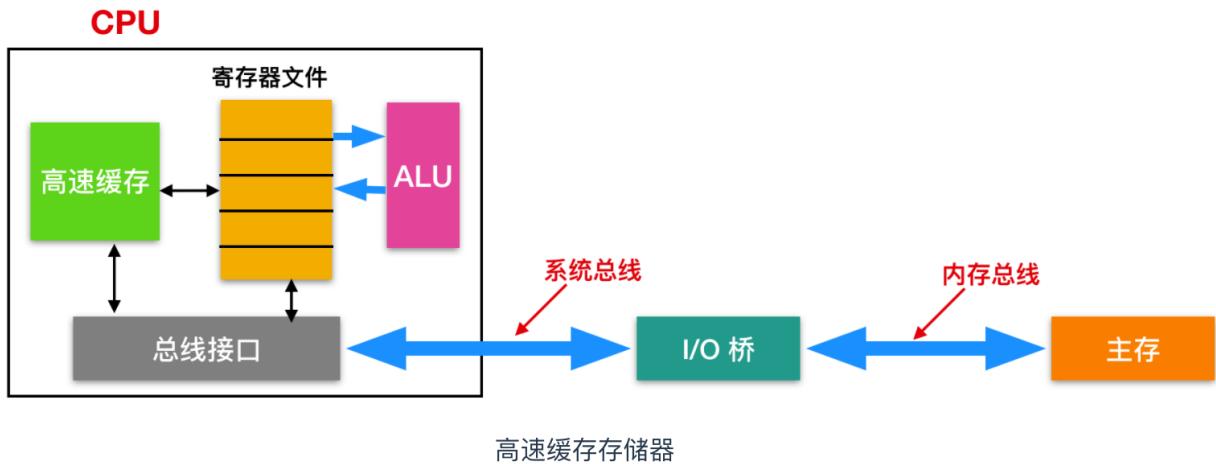
一旦目标文件中 hello 中的代码和数据被加载到主存，处理器就开始执行 hello 程序的 main 程序中的机器语言指令。这些指令将 hello,world\n 字符串中的字节从主存复制到寄存器文件，再从寄存器中复制到显示设备，最终显示在屏幕上。如下所示



## 高速缓存是关键

上面我们介绍完了一个 hello 程序的执行过程，系统花费了大量时间把信息从一个地方搬运到另外一个地方。hello 程序的机器指令最初存储在磁盘上。当程序加载后，它们会拷贝到主存中。当 CPU 开始运行时，指令又从内存复制到 CPU 中。同样的，字符串数据 hello,world \n 最初也是在磁盘上，它被复制到内存中，然后再写到显示器设备输出。从程序员的角度来看，这种复制大部分是开销，这减慢了程序的工作效率。因此，对于系统设计来说，最主要的一个工作是让程序运行的越来越快。

由于物理定律，较大的存储设备要比较小的存储设备慢。而由于寄存器和内存的处理效率越来越大，所以针对这种差异，系统设计者采用了更小更快的存储设备，称为 高速缓存存储器(cache memory，简称为 cache 高速缓存)，作为暂时的集结区域，存放近期可能会需要的信息。如下图所示



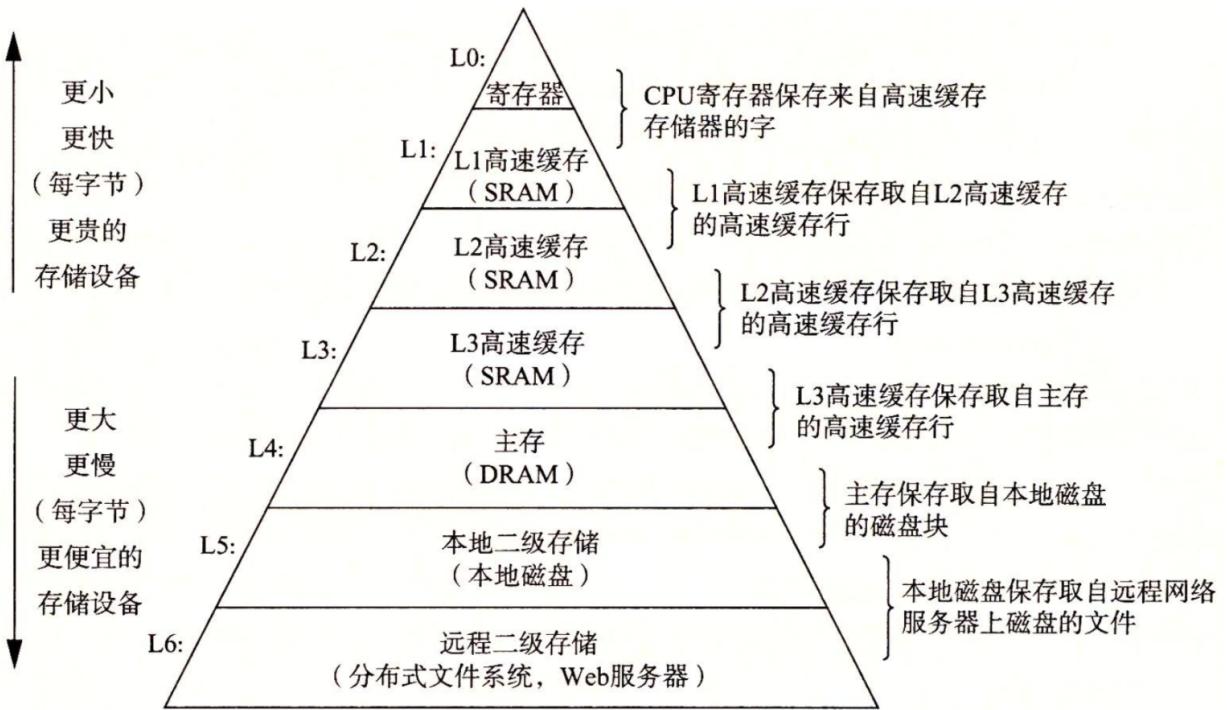
图中我们标出了高速缓存的位置，位于高速缓存中的 L1 高速缓存容量可以达到数万字节，访问速度几乎和访问寄存器文件一样快。容量更大的 L2 高速缓存通过一条特殊的总线链接 CPU，虽然 L2 缓存比 L1 缓存慢 5 倍，但是仍比内存要快 5 - 10 倍。L1 和 L2 是使用一种 静态随机访问存储器(SRAM) 的硬件技术实现的。最新的、处理器更强大的系统甚至有三级缓存：L1、L2 和 L3。系统可以获得一个很大的存储器，同时访问速度也更快，原因是利用了高速缓存的 局部性 原理。

**局部性原理：**在 cs 中，引用局部性，也称为局部性原理，是 CPU 倾向于在短时间内重复访问同一组内存的机制。

通过把经常访问的数据存放在高速缓存中，大部分对内存的操作直接在高速缓存中就能完成。

## 存储设备层次结构

上面我们提到了 L1、L2、L3 高速缓存还有内存，它们都是用于存储的目的，下面为你绘制了它们之间的层次结构

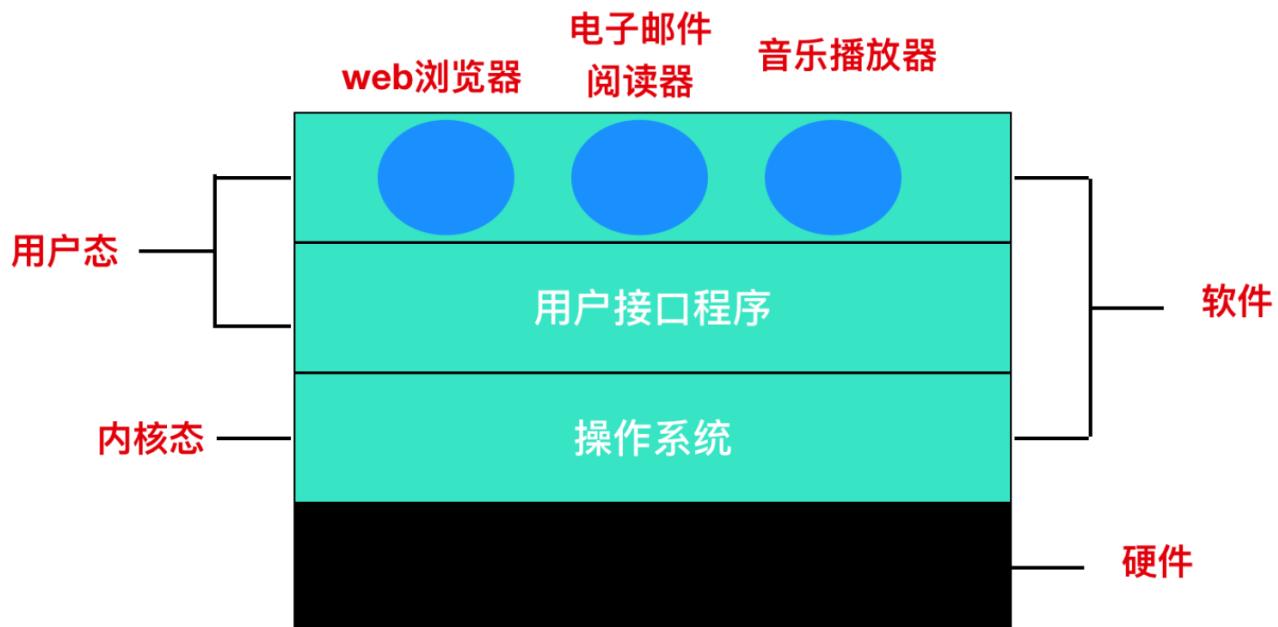


一个存储器层次结构的示例

存储器的主要思想就是上一层的存储器作为低一层存储器的高速缓存。因此，寄存器文件就是 L1 的高速缓存，L1 就是 L2 的高速缓存，L2 是 L3 的高速缓存，L3 是主存的高速缓存，而主存又是磁盘的高速缓存。这里简单介绍一下存储器设备层次结构，具体的会在后面介绍。

## 操作系统如何管理硬件

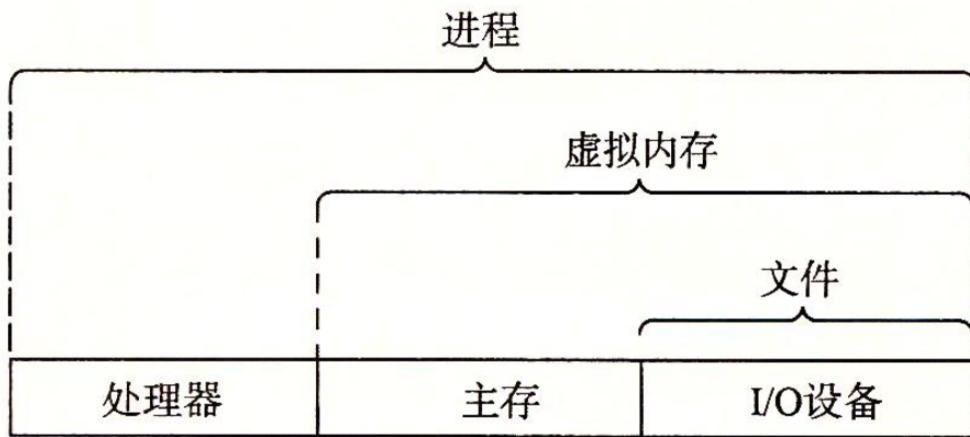
再回到我们这个 hello 程序中，当 shell 加载并运行 hello 程序，以及 hello 程序输出自己的消息时，shell 和 hello 程序都没有直接访问键盘、显示器、磁盘或者主存，相反，它们会依赖 操作系统(operating System) 做这项工作。操作系统是一种软件，我们可以将操作系统视为介于应用程序和硬件之间的软件层，所有想要直接对硬件的操作都会通过操作系统。



操作系统有两项基本的功能：

- **操作系统能够防止硬件被失控程序滥用**
- **向应用程序提供简单一致的机制来控制低级硬件设备。**

那么操作系统是通过什么实现对硬件的操作的呢？无非是通过 **进程**、**虚拟内存**、**文件** 来实现这两个功能。



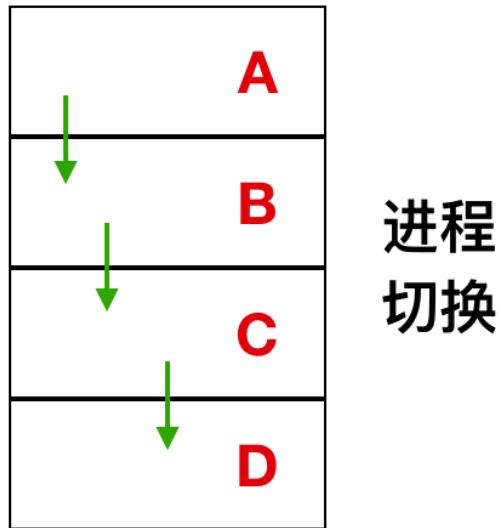
文件是对 I/O 设备的抽象表示，虚拟内存是对主存和磁盘 I/O 设备的抽象表示，进程则是对处理器、主存和 I/O 设备的抽象表示。下面我们依次来探讨一下

**进程**

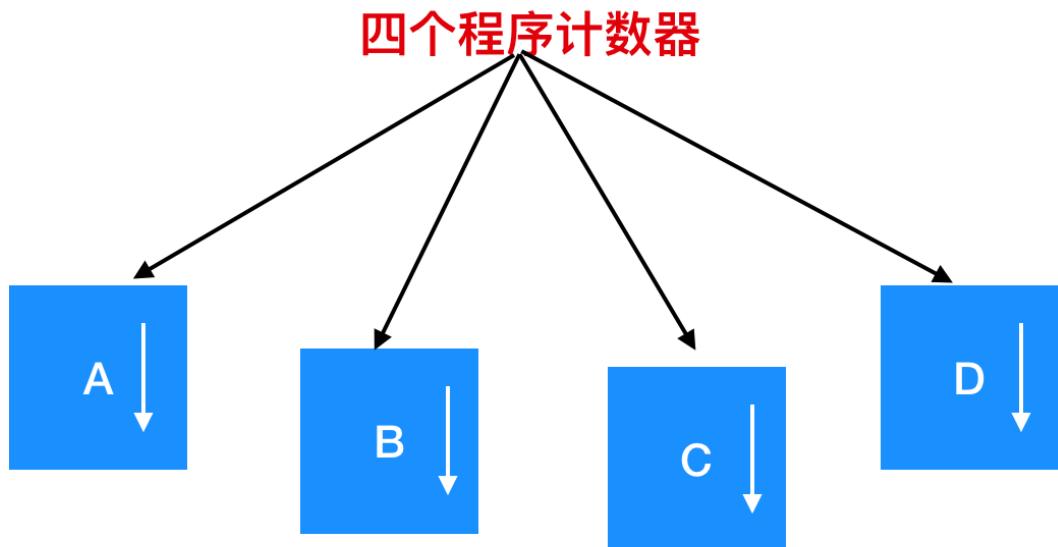
进程 是操作系统中的核心概念，进程是对正在运行中的程序的一个抽象。操作系统的其他所有内容都是围绕着进程展开的。即使只有一个 CPU，它们也支持（伪）并发操作。它们会将一个单独的 CPU 抽象为多个虚拟机的 CPU。我们可以把进程抽象为一种进程模型。

在进程模型中，一个进程就是一个正在执行的程序的实例，进程也包括程序计数器、寄存器和变量的当前值。从概念上来说，每个进程都有各自的虚拟 CPU，但是实际情况是 CPU 会在各个进程之间进行来回切换。

## 程序计数器

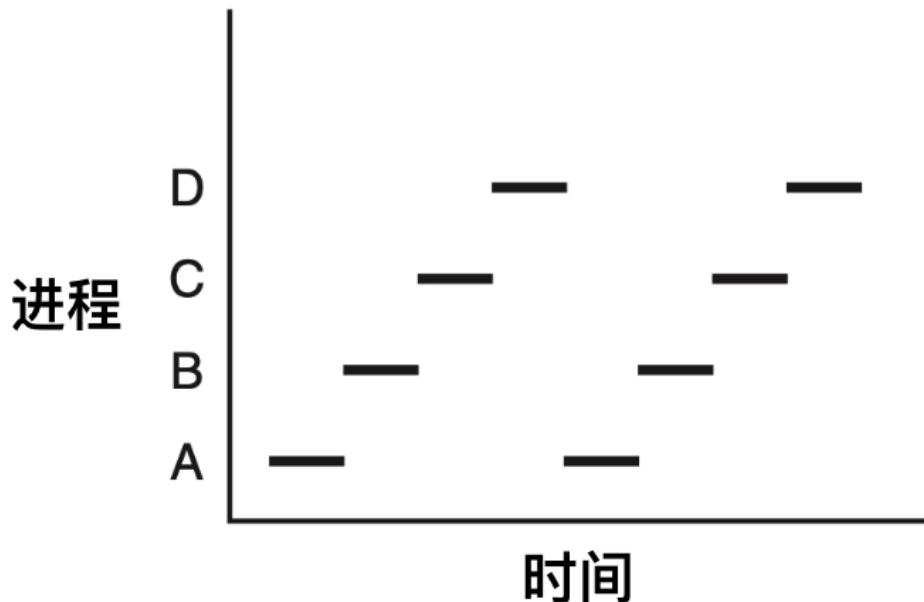


如上图所示，这是一个具有 4 个程序的多道处理程序，在进程不断切换的过程中，程序计数器也在不同的变化。



在上图中，这 4 道程序被抽象为 4 个拥有各自控制流程（即每个自己的程序计数器）的进程，并且每个程序都独立的运行。当然，实际上只有一个物理程序计数器，每个程序要运行时，其逻辑程序计数器会装载到物理程序计数器中。当程序运行结束后，其物理程序计数器就会是真正的程序计数器，然后再把它放回进程的逻辑计数器中。

从下图我们可以看到，在观察足够长的一段时间后，所有的进程都运行了，**但在任何一个给定的瞬间仅有一个进程真正运行**。

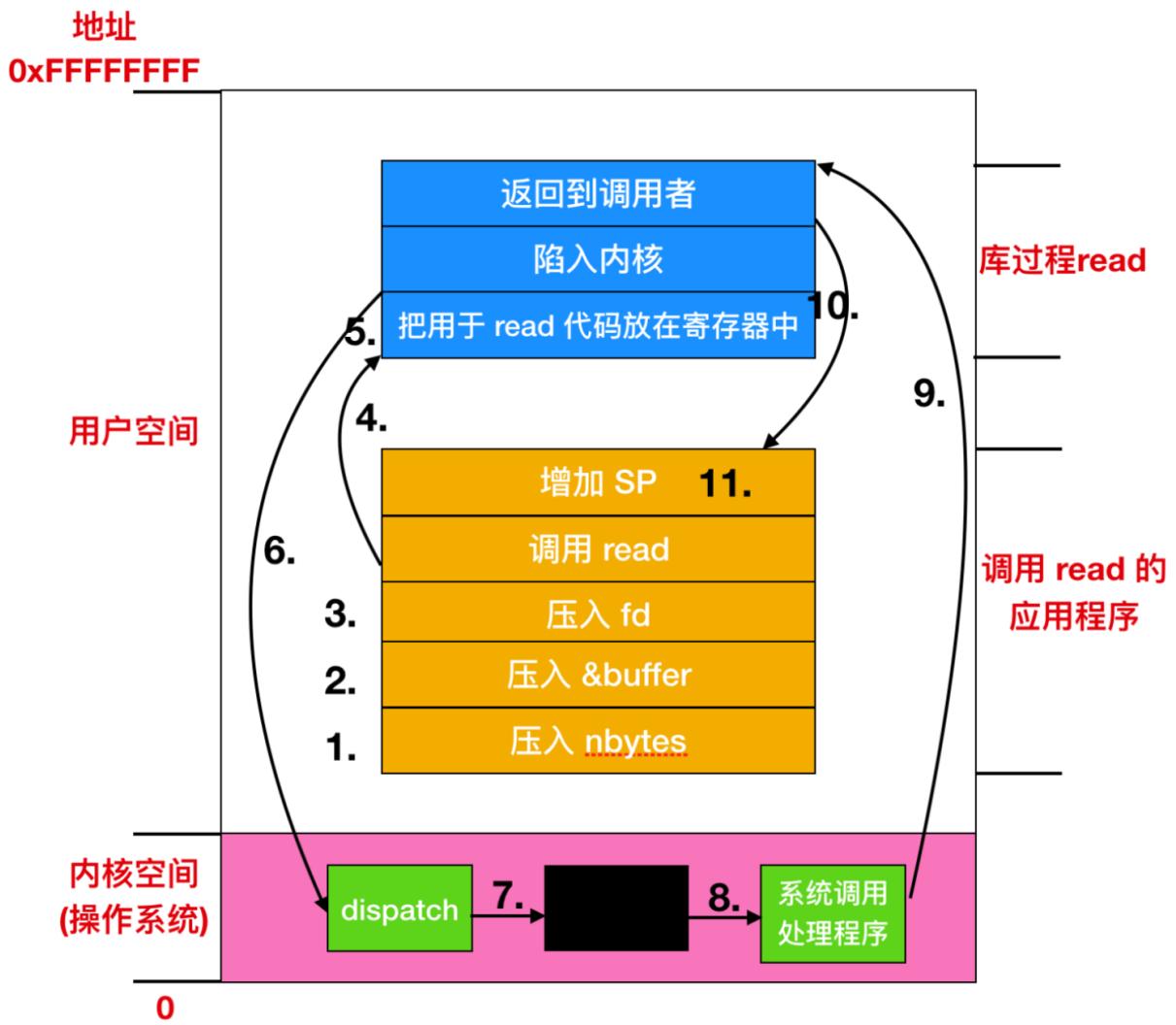


因此，当我们说一个 CPU 只能真正一次运行一个进程的时候，即使有 2 个核（或 CPU），**每一个核也只能一次运行一个线程**。

由于 CPU 会在各个进程之间来回快速切换，所以每个进程在 CPU 中的运行时间是无法确定的。并且当同一个进程再次在 CPU 中运行时，其在 CPU 内部的运行时间往往也是不固定的。

如下图所示，从一个进程到另一个进程的转换是由操作系统 内核(kernel) 管理的。内核是操作系统代码 常驻 的部分。当应用程序需要操作系统某些操作时，比如读写文件，它就会执行一条特殊的 系统调用 指令。

注意：内核不是一个独立的进程。相反，它是系统管理全部进程所用代码和数据结构的集合。



完成系统调用 `read(fd, buffer, nbytes)` 的 11 个步骤

我们会在后面具体介绍这些过程

## 线程

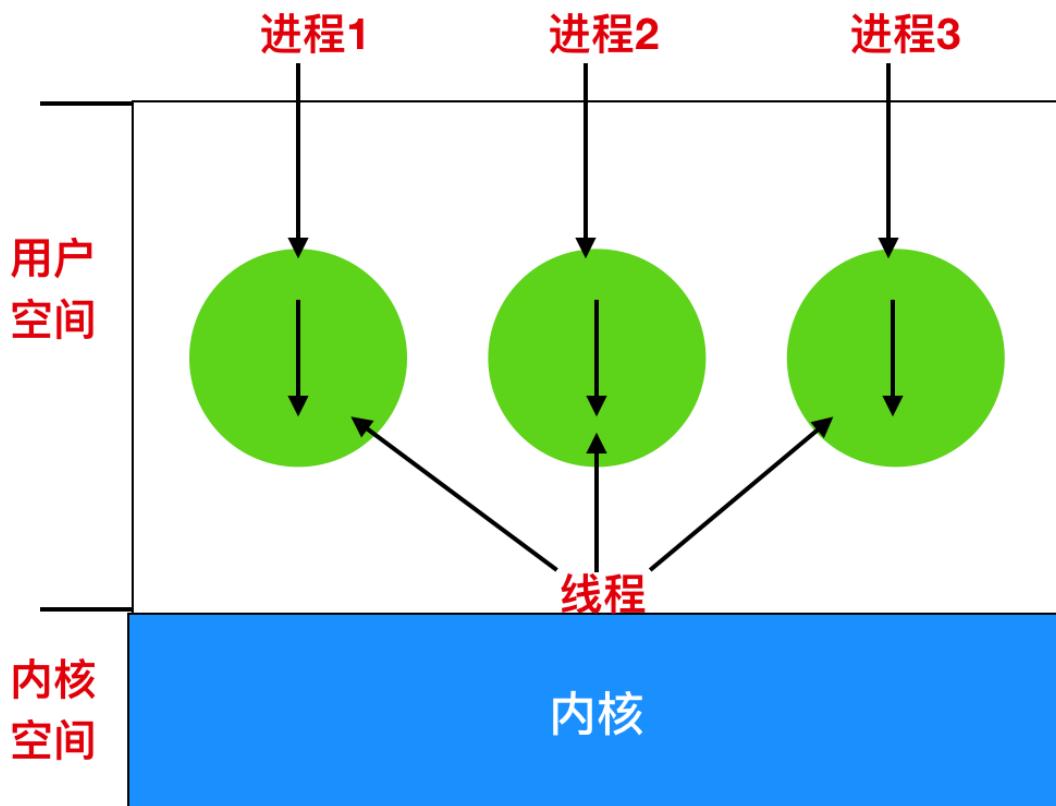
在传统的操作系统中，每个进程都有一个地址空间和一个控制线程。事实上，这是大部分进程的定义。不过，在许多情况下，经常存在同一地址空间中运行多个控制线程的情形，这些线程就像是分离的进程。准确的说，这其实是进程模型和线程模型的讨论，回答这个问题，可能需要分三步来回答

- 多线程之间会共享同一块地址空间和所有可用数据的能力，这是进程所不具备的
- 线程要比进程更轻量级，由于线程更轻，所以它比进程更容易创建，也更容易撤销。在许多系统中，创建一个线程要比创建一个进程快 10 - 100 倍。
- 第三个原因可能是性能方面的探讨，如果多个线程都是 CPU 密集型的，那么并不能获得性能上的增强，但是如果存在着大量的计算和大量的 I/O 处理，拥有多个线程能在这些活动中彼此重叠进行，从而会加快应用程序的执行速度

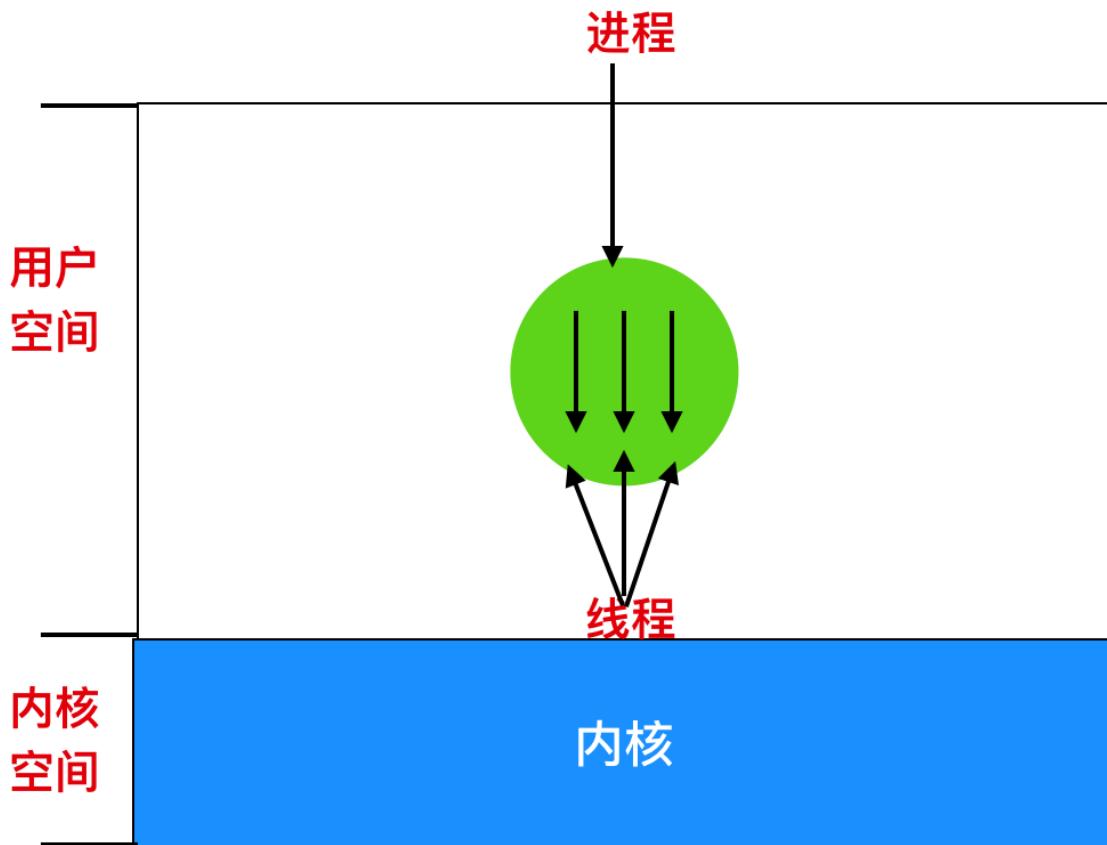
进程中拥有一个执行的线程，通常简写为 线程(thread)。线程会有程序计数器，用来记录接着要执行哪一条指令；线程还拥有 寄存器，用来保存线程当前正在使用的变量；线程还会有堆栈，用来记录程序的执行路径。尽管线程必须在某个进程中执行，但是进程和线程完全是两个不同的概念，并且他们可以分开处理。进程用于把资源集中在一起，而线程则是 CPU 上调度执行的实体。

线程给进程模型增加了一项内容，即在同一个进程中，允许彼此之间有较大的独立性且互不干扰。在一个进程中并行运行多个线程类似于在一台计算机上运行多个进程。在多个线程中，各个线程共享同一地址空间和其他资源。在多个进程中，进程共享物理内存、磁盘、打印机和其他资源。因为线程会包含有一些进程的属性，所以线程被称为 轻量的进程 (lightweight processes)。多线程(multithreading)一词还用于描述在同一进程中多个线程的情况。

下图我们可以看到三个传统的进程，每个进程有自己的地址空间和单个控制线程。每个线程都在不同的地址空间中运行



下图中，我们可以看到有一个进程三个线程的情况。每个线程都在相同的地址空间中运行。



## 虚拟内存

虚拟内存的基本思想是，每个程序都有自己的地址空间，这个地址空间被划分为多个称为页面(page)的块。每一页都是连续的地址范围。这些页被映射到物理内存，但并不是所有的页都必须在内存中才能运行程序。当程序引用到一部分在物理内存中的地址空间时，硬件会立刻执行必要的映射。当程序引用到一部分不在物理内存中的地址空间时，由操作系统负责将缺失的部分装入物理内存并重新执行失败的指令。

在某种意义上来说，虚拟地址是对基址寄存器和变址寄存器的一种概述。8088 有分离的基址寄存器（但不是变址寄存器）用于放入 text 和 data。

使用虚拟内存，可以将整个地址空间以很小的单位映射到物理内存中，而不是仅仅针对 text 和 data 区进行重定位。下面我们会探讨虚拟内存是如何实现的。

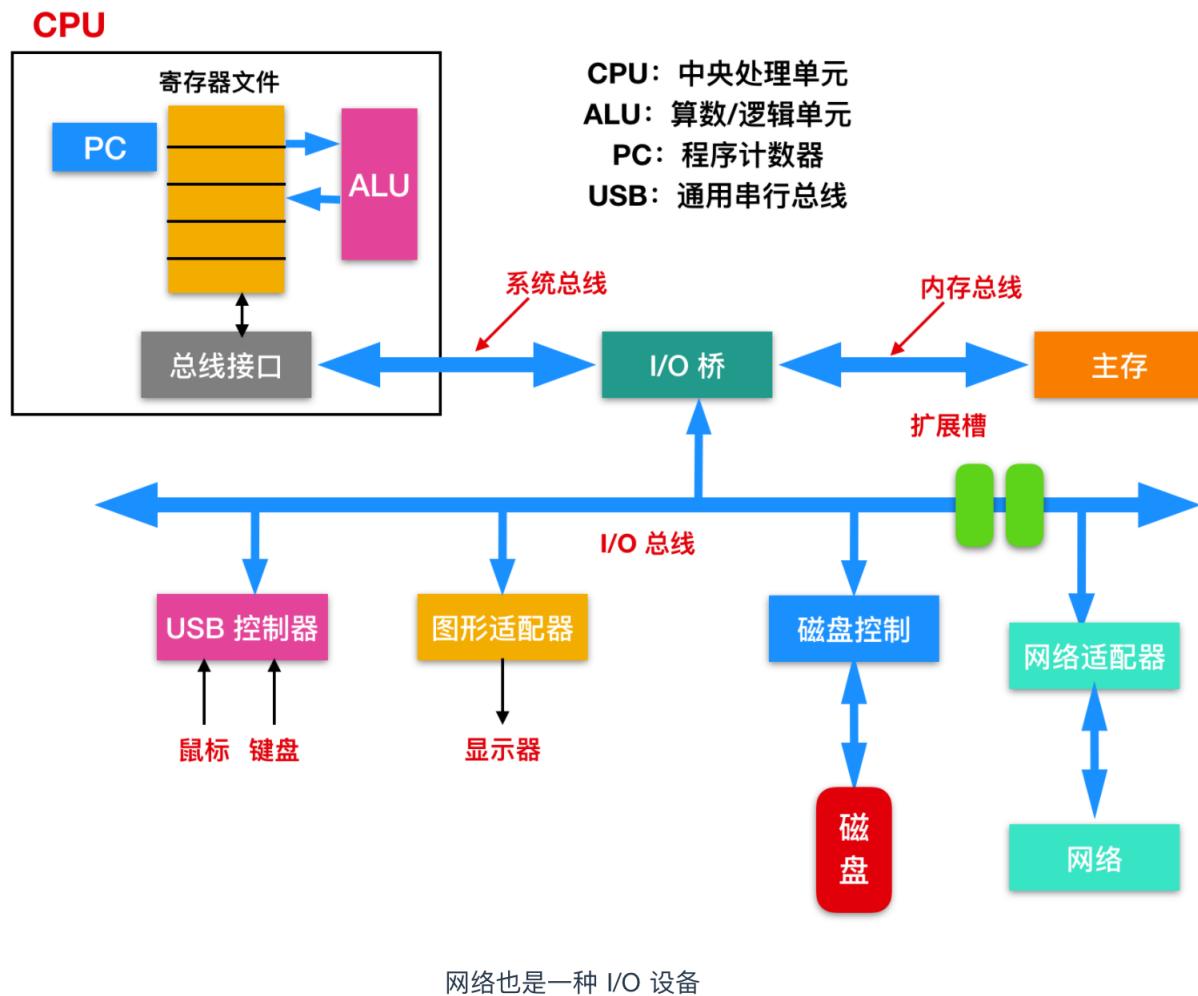
虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中，当一个程序等待它的一部分读入内存时，可以把 CPU 交给另一个进程使用。

## 文件

文件(Files)是由进程创建的逻辑信息单元。一个磁盘会包含几千甚至几百万个文件，每个文件是独立于其他文件的。它是一种抽象机制，它提供了一种方式用来存储信息以及在后面进行读取。

## 网络通信

现代系统是不会独立存在的，因此经常通过网络和其他系统连接到一起。从一个单独的系统来看，网络可以视为 I/O 设备，如下图所示



当系统从主存复制一串字节到网络适配器时，数据流经过网络到达另一台机器，而不是说到达本地磁盘驱动器。类似的，系统可以读取其他系统发送过来的数据，把数据复制到自己的主存中。

随着 internet 的出现，数据从一台主机复制到另一台主机的情况已经成为最重要的用途之一。比如，像电子邮件、即时通讯、FTP 和 telnet 这样的应用都是基于网络复制信息的功能。