# 01-A Tour of Computer Systems-计算机系统漫游

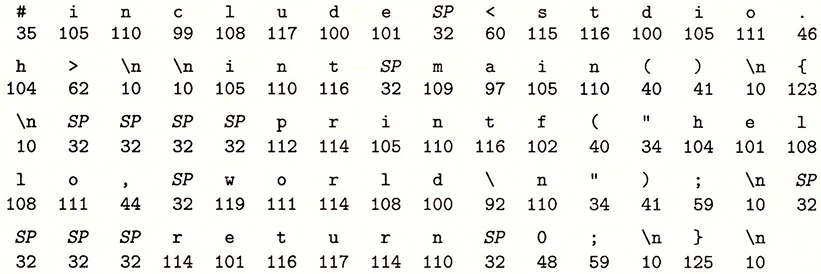
计算机系统由硬件和系统软件组成，他们共同工作，但是具体的实现方式随着时间不停变化。不过系统的内在概念没有改变。我们可以通过理解软硬件的相互工作，深入理解来明白这些组件是如何影响程序的正确性和性能。不过这一切的一切我们大抵都要从我们的HelloWorld开始。byd兜兜转转还是回到了HelloWorld对吗？

好的，先让我们找个傻逼代码：

#include<stdio.h>  
  
int main()  
{  
 printf("hello, world\n");  
 return 0;  
}

# 1.1 信息 = 位 + 上下文

helloword程序的生命周期是从一个源文件开始的。程序员通过编写文件《hello.c》——一堆由0和1组成的位序列，8个组成一组，成为字节。由字节来表示程序中的大部分文本字符。



啥？详细的表示你去看ASCII就好了，这个应该只要学过计算机都知道。像这样用ASCII码组成的文件都叫做**文本文件**，其他的我们统称为**二进制文件**

为什么用C语言呢？因为C语言是一个与Unix内核相关的语言，大部分操作系统的核心都是Unix，而C语言一开始只是为了Unix系统而开发出来的程序语言。因此保留了C语言的第一个特性：移植性。  
C语言非常简单，因为它是由一个人掌控的而并非一个协会。所以这个语言简洁明了，没有什么多余的设计。  
C语言因为是为了Unix系统而开发的语言，其实他作为一个老东西，是系统级编程的首选（因为它可以写很多底层的东西），再者，其实用它来开发什么程序也完全不会差。不过傻逼学不明白，很显然的一点，因为他保留了底层最关键的东西：指针——一个让人需要理解大量抽象知识才能完全掌握的东西。  
C语言为数不多的缺点是其实他没有什么抽象的显式支持，例如类、对象、异常等等。于是C++和Java就横空出世了。

这个文件其实告诉了我们计算机一个最基础的思想：系统中所有的信息：磁盘文件，程序，内存还是网络上传输的数据都是由一串bits组成的，不同的上下文中，同样的一串bits可以是整数、浮点数、字符串、亦或者是什么机器指令。这也组成了各种我们能遇见的奇怪事情：也就是同样的数据也许会以不同的方式表示。

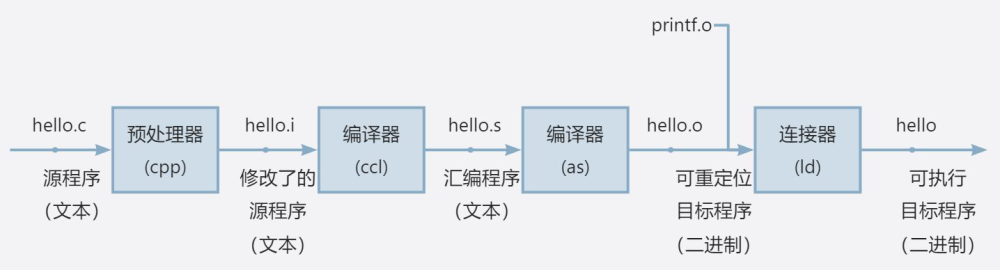
# 1.2 程序被其他程序翻译成不同的格式

为什么C语言是文本文件？当然因为这种格式可以让人看懂，接下来我们要让计算机看懂了。为了在系统上运行，我们需要把所有的C语言语句转化成一条条低级机器语言指令，然后把这些指令按照一种称为*可执行目标程序*的格式打包好后，以二进制的形式存放起来。目标程序也被叫做——可执行目标文件。

比如在Unix（这边我们拿ubuntu做实验环境）



我们用GCC编译器驱动程序来读取hello.c，并且把它编译成一个可执行的“hello”文件。这是一个翻译过程，具体如下：

在cpp，ccl，as，ld（C preprocessor - C预处理器、Clang Compiler Frontend - Clang编译器、Assembler汇编器、Linker链接器）这四项组合后，就构成了编译系统（Compilation system）。其中：

**预处理阶段：**cpp根据以字符#开头的命令，修改原始的C语言程序，把这些文件全部插入到程序文本中。结果就得到了另一个C程序，我们一般以.i作为文件扩展名。

**编译阶段：**ccl将文本文件hello.i翻译成hello.s，这个文件由C语言文件变成了*汇编语言程序*，该程序包含我们对main的定义：

main:  
 sub rsp, 8  
 mov edi, str  
 call puts  
 mov eax, 0  
 add rsp, 8  
 ret

汇编语言的一大好处就是，他统一了一个软件系统的所有汇编指令，不同语言写法用法和功能侧重点都不同，不过经过编译之后都变成了一样的汇编指令。然后我们可以通过汇编指令进行汇编操作。

**汇编：**接下里，汇编器as会把hello.s翻译成机器语言指令，这些指令打包成一种叫做*可重定位目标程序（relocatable object program）*，结果是一个二进制程序保存在hello.o中。他将包含main的指令编码。直接打开将是一串乱码。

**链接：**因为我们在hello.c中使用了printf这个函数，这个是C语言编译器提供的stdio.h标准库中的一个函数。而这个stdio.h如果要找到printf，就需要找到printf.o这个文件，这个文件必须以某种方式存在在我们最终的目标程序上，链接就是干这一步的。如此这般，就可以得到最终的*可执行目标程序（可执行文件Executable program）*，这样就可以被加载到内存中，由系统执行了。

GCC是GNU（GNU's Not Unix）项目开发出来的工具之一。至于为什么叫做这个名字，是因为GNU是一个包含Unix系统的类Unix系统，目的是为了开源让所有人都可以不受限制的修改其源代码。不过内核除外，因为内核是由Linux项目独立开发出来的。  
GNU取得了非常非凡的成绩，拥有众多编译器和编译环境，能够为不同的机器生成不同的代码。Linux系统的成功，GNU是功不可没的。  
Linux系统是模仿Unix系统做的类Unix系统，因为Unix系统是要钱的（商业软件）。再者，Unix可以选择不同的内核（kernel），而Linux一般只有一个内核。而MacOS是基于Unix的，虽然一开始他的名字叫做XNU（X's Not Unix），其中MacOS的XNU内核，是Mach+BSD，而BSD就是从Unix继承下来的一个内核。

# 1.3 知道这些玩意有啥用

其中很大的一个作用就是，作为一个程序员，我们可以通过这个编译系统如何工作的流程了解到很多信息：

**1、优化程序性能：**我们可以通过大量的调试来了解一些东西：switch是否总是比ifelse更加高效，for和while谁才是爹，一个函数调用时候的开销有多大？指针比数组优秀吗……很多很多的问题都可以通过这样的方式理解。

**2、理解链接时候出现的错误：**各种静态库和动态库的问题都可以通过理解链接这一步骤理解完毕。想想看，当你用vs辛辛苦苦写出来一个程序想要炫耀的时候，如果不是release版本而是debug版本发给别人的话，那么别人就会报一堆链接错误。亦或者是自己写程序的时候也会出现这样那样的错误。

**3、避免安全漏洞：**这是我学CSAPP很大的一部分原因，这个流程既可以了解到如何保护程序，也可以破解程序。几乎所有的安全漏洞都是缓冲区溢出之类的问题。

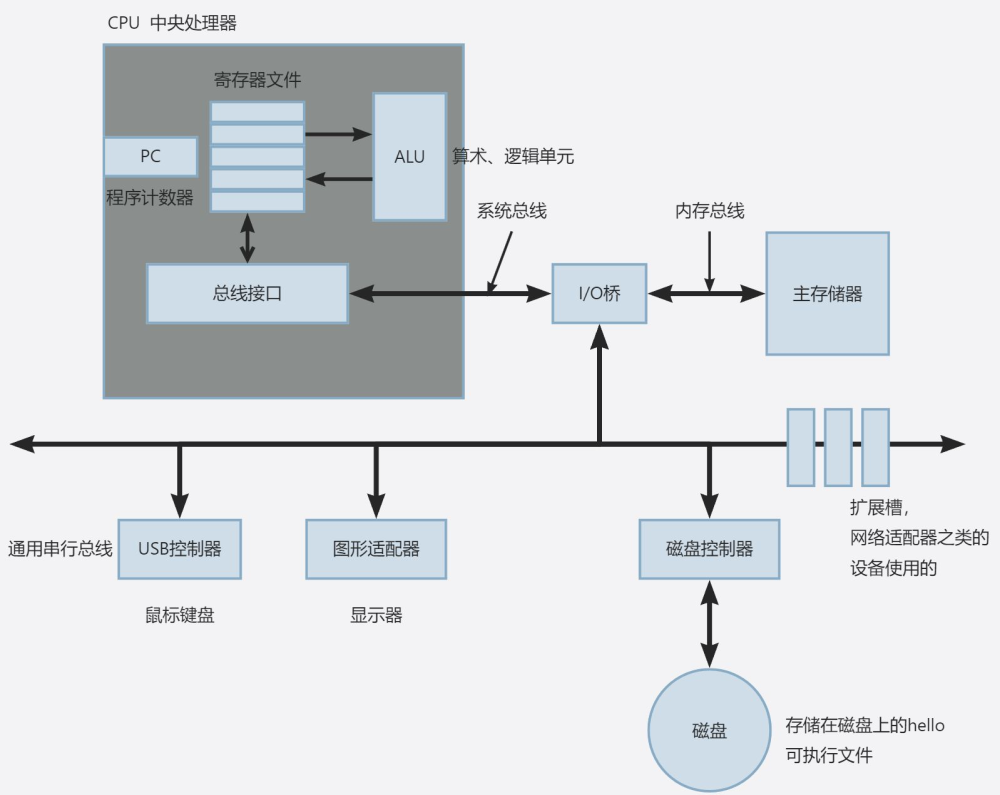
# 1.4 处理器 读取并解释 储存在内存中的 指令

现在，我们已经得到一个hello文件了，可以通过linux自带的shell工具打开。是的，也就是命令行可以直接打开：



## 1.4.1 系统的硬件组成

为了理解我们运行hello的时候到底发生了什么，我们需要理解一系列硬件组织。就拿Intel芯片举例吧：

这张图看起来很简单，其实很复杂。接下来我会慢慢解释，先简单讲一下：

**1、总线：**贯穿整个系统的电子管道。他携带的信息字节负责在各个部件之间进行传输。总线被设计成传输固定的字节块，也就是*字（word）*。字中的字节数是一个基本的系统参数，每个系统中都不一样。32位的系统和64位的系统应运而生，分别为4个字节或者8个字节。

**2、I/O设备：**图中的USB控制器，图形适配器还有磁盘控制器等等都是IO设备中的一环。我们制作的hello.c程序就是存在磁盘中的。每一个外部设备（IO设备）都会通过一个控制器或者适配器与总线进行连接，进行数据传输。

控制器和适配器之间的区别在于他们的封装方式。控制器是IO设备本身或者系统的主印制电路板（也就是主板）上的芯片组，而适配器则是一块插在主板上插槽的卡。不过他们的功能是一样的：在IO总线和IO设备之间传输数据。

**3、主存：**主存是一个零食存储设备，用来放程序和程序处理的数据（在程序被处理器处理（执行）的时候），从物理上来说存储器是由一组*动态随机存取存储器（DRAM）*的芯片组成的。从逻辑上来说，存储器是一个线性的字节数组，每个字节都有其唯一的地址（数组索引），这些地址都是从0开始的。一般而言，组成程序的每条机器指令又都不同的数量的字节构成。其实大部分时候这一块我们会直接叫做内存。

与C程序的变量相对应的数据项的大小是更具类型变化的。比如在运行Linux的X86\_64系统上，short需要2个字节，而int和float需要4个字节，long和double需要8个字节。

**4、处理器：**中央处理单元（CPU），也就是处理器，是解释或执行存储在主存中指令的引擎。处理器的核心是一个大小为一个字的存储设备（也就是寄存器），称为*程序计数器（PC）*，在任何时刻，PC都是只想主存中某条机器语言指令的（含有该指令的地址）。

从系统通电开始，直到系统断电。整个过程中处理器会不停的执行程序计数器指向的指令，再更新程序计数器——指向下一条指令。处理器看上去是按照一个非常简单的指令执行模型来操作的，这个模型是由*指令集架构*所决定的。在这个模型中，指令按照严格的顺序执行，执行一条指令中包含一系列的步骤。

处理器从程序计数器指向的内存中的包含所需指令的地址，解释指令中的位，执行指令中的操作，然后更新PC使其执行下一条包含指令的地址。这条指令并不再内存中刚刚执行的指令相邻。

这样的简单操作并不会太多，他们围绕着主存，寄存器文件和算数/逻辑单元进行。寄存器就是一个超级小超级快的内存，由单一的字长组成。每个寄存器都有一个他的名字。ALU计算新的数据和地址值。CPU在指令的要求下可能会执行这些操作：加载、储存、操作和跳转。我们会在讲寄存器的时候详细的讲解。

处理器看上去是他的指令集架构的简单实现，但是实际上现代的处理器使用了非常复杂的机制来加速程序的执行。

## 1.4.2 运行hello程序

这一块我们直接简单的讲解一下理解一下就好。

1、USB获取键盘数据（输入./shell 回车)，将数据通过IO总线发送给IO桥，IO桥将数据发送到主存储器中，并且把这条“打开程序”指令发送给CPU进行梳理。再CPU中，数据通过系统总线传到总线接口，发送给寄存器。

2、这一系列指令将hello目标文件中的代码和数据从磁盘直接进行复制发送到主存。

3、将代码发送给寄存器文件进行处理，寄存器指向了内存中的文件命令地址，然后和CPU做运算逻辑处理。把处理之后的数据发送给图形适配器，发送给显示器显示“hello，world\n”

# 1.5 高速缓存

这个实例最明显的一点就是，我们发现大部分时候计算机系统就是把一部分数据复制到另外一个地方上。每一笔赋值操作都是时间开销，减缓了程序真正的工作。

为了处理这个速度问题（主要是又快又大的主存开销很大，所以必须要有外存的存在），系统设计者采用了更小更快的存储设备，被叫做*高速缓存存储器（cache memory，一般简称cache）*，作为暂时的集结区域。存放CPU近期可能会需要的信息。

位于处理器上的L1高速缓存容量可以达到数万字节，其速度几乎和访问寄存器一样快，随后是一个更大的L2高速缓存器，拥有数百万的存储空间，其代价是速度是L1高速缓存的5倍。最新的处理器也许还会有一个L3高速缓存器。

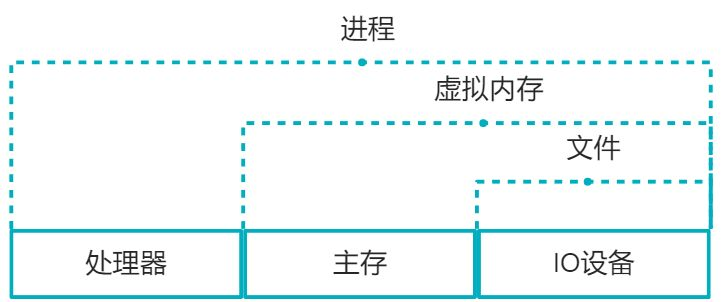
# 1.6 存储器设备形成层次结构

如此这般，组成了一个存储器层次结构：

# 1.7 操作系统 管理硬件

当我们回到hello程序的例子，当shell加载和运行hello程序的时候，以及hello程序输出自己的消息的时候，shell和hello都没有直接访问键盘显示器磁盘等空间，而是通过*操作系统*来进行服务，操作系统就是应用程序和软件程序之间插入的一层软件。所有的系统程序对硬件的操作都必须通过操作系统。

操作系统有两个操作：1是防止硬件被失控的程序滥用，2是向应用程序提供简单一致的机制来控制复杂而又通常大不相同的低级硬件设备。操作系统通过几个基本的抽象概念（进程，虚拟内存，文件）来实现这两个功能：

文件是对IO设备抽象地表示，虚拟内存是对主存和磁盘IO空间的抽象表示，进程则是对处理器，主存和IO设备的抽象表示。

20 世纪 60 年代大型复杂操作系统盛行，如 IBM OS/360（成功项目）、Honeywell Multics（未广泛应用 ）。贝尔实验室因 Multics 复杂、进展慢，1969 年 Ken Thompson 等基于 DEC PDP - 7，用机器语言开发更简单系统，思想源于 Multics（如层次文件系统、shell 概念 ）。1970 年 Brian Kernighan 命名 “Unix”（双关暗指 Multics 复杂 ），1973 年用 C 重写内核，1974 年正式发布 。  
贝尔实验室向学校慷慨开放源代码，Unix 在高校获支持发展。20 世纪 70 年代末 - 80 年代初，加州大学伯克利分校推出含虚拟内存、Internet 协议的 4.xBSD；贝尔实验室发布 System V Unix 。Sun 等厂商的 Solaris 等系统，从 BSD、System V 衍生 。  
20 世纪 80 年代中期，Unix 厂商为差异化加不兼容特性致麻烦。IEEE 推动标准化，Richard Stallman 命名为 “Posix”，涵盖系统调用 C 接口、shell、线程、网络编程等标准 。后续 “标准 Unix 规范” 与 Posix 统一标准，Unix 版本差异基本消失 。

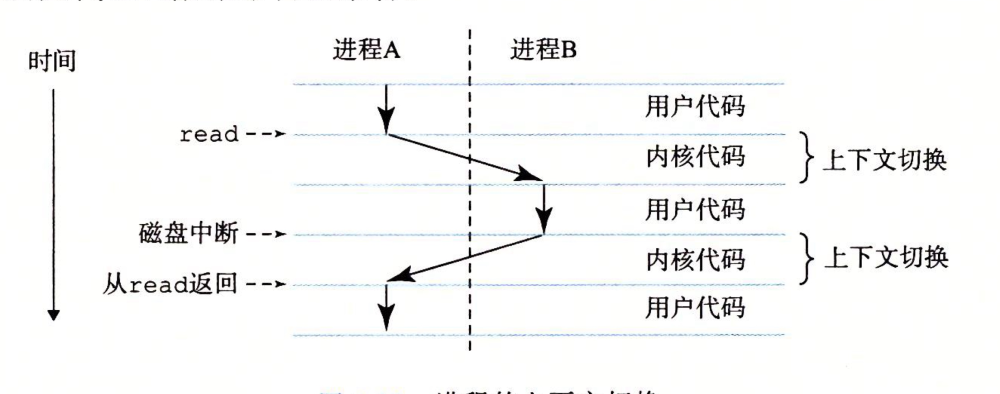
## 1.7.1 进程

我们会发现，像是hello这样的程序在现代系统上运行的时候，操作系统会提供一种假象：就好像系统上就只有这个程序在运行。程序看上去只是独占的使用处理器、主存和IO设备。似乎只是不间断的一条一条的执行当前程序的指令。

实际上，这些假象都是通过“*进程*”这个概念来实现的。进程是操作系统中对一个正在运行的程序的一种抽象。它可以让我们觉得一个系统上运行着好多的进程且每个进程都在独占地使用硬件。

而这个则是*并发运行，*就是一个进程的指令和另外一个进程的指令是操作执行的。在大多数系统中，需要运行的进程数是多余可以运行他们的CPU个数的，传统系统中在一个时刻只能执行一个进程，而现代多核处理器则可以一个时刻处理多个进程。为了简化学习过程接下来是以传统系统来讲解。

操作系统首先会对每一个进程运行所需的状态信息进行跟踪和保存。这种状态也就是*上下文*，包括许多信息，例如PC的位置，寄存器的值，主存的内容等等。在任何一个时刻，单处理器都只能执行一个进程的代码。当操作系统决定要把控制权从当前进程转移到另外一个进程的时候，则会进行“*上下文切换*”——及保存当前进程的上下文，恢复另一个进程的上下文，然后将控制权传递给第二个进程（后面叫做新进程），新进程获取控制权后，就会从他上次停止的地方开始继续运行。如下图所示：



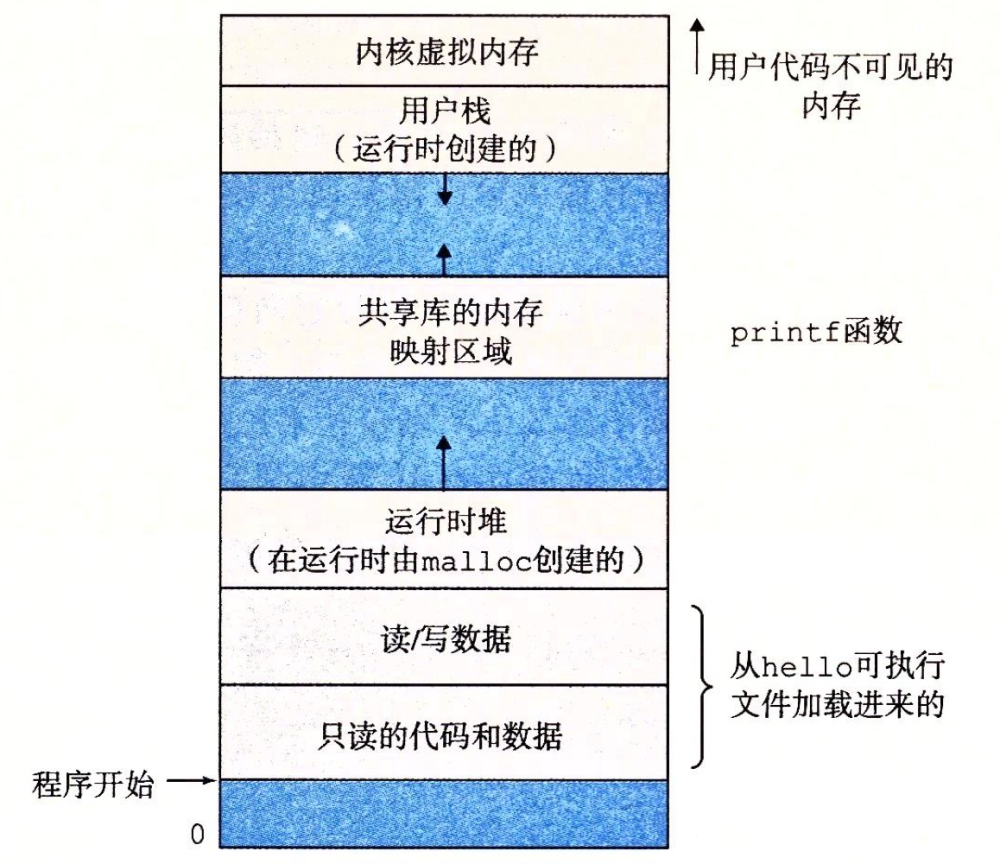
实现进程这个抽象，需要底层硬件系统和操作系统软件的紧密合作。

## 1.7.2 线程

除了进程，还有线程这个概念。尽管通常我们认为一个进程只有一个单一的控制流，但是在现代系统上，一个进程实际上可以有多个称之为线程的执行单元组成。例如java中的tread、runnable这些。每个线程都运行在进程的上下文中，并且共享同样的进程中的代码和全局数据。这是因为网络服务器中对于并行处理的需求，线程成为越来越重要的编程模块了。当有多处理器可用的时候，多线程也成为了一个可以让程序运行的更快的办法之一。

## 1.7.3 虚拟内存

虚拟内存是一个抽象概念，属于是进程之下的一个抽象概念——即每个进程都在独占地使用主存。每个进程看到的虚拟内存空间几乎无异，把这个叫做*虚拟地址空间*。在linux或unix、类unix系统中虚拟内存布局如下所示：



这里的内容我懒得赘述，可以去查看：[pwn及计算机原理基础知识](https://www.yuque.com/kanyo-er9pl/bn7s26/div2ywly40p42zpw)

不过较后面的笔记会详细解释里面的东西。

虚拟内存的运作需要硬件和操作系统软件之间精密复杂的交互，包括对处理器生成的每个地址的硬件解释等。核心就是把一个进程的虚拟内存的内容保存在磁盘上，然后用主存作为磁盘的高速缓存。

## 1.7.4 文件

文件就是字节序列。在linux系统中被称作“一切皆文件”。磁盘、键盘、显示器、网络这些IO设备都可以看作是文件。系统中的所有输入输出都是通过使用一小组称为UnixIO的系统函数调用读写文件来实现的。

其实文件这个抽象概念是很强大的，因为把一切东西都用了一种统一的视图来查阅，因为把一切都抽象成一串可以供处理磁盘文件的应用程序员进行处理的内容。不然的话在不同的磁盘技术上，就要学习不同的磁盘数据保存方法。

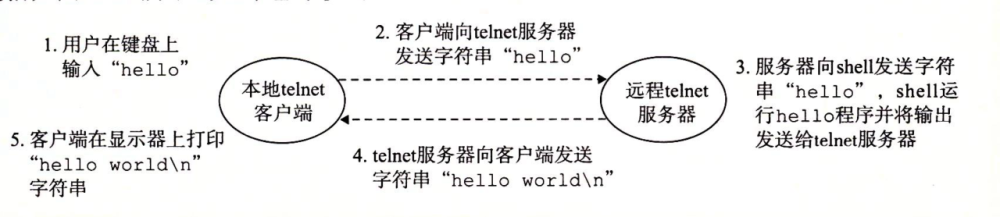
Linux小故事：  
1991 年 8 月，芬兰研究生 Linus Torvalds 谨慎地发布了一个新的类 Unix 的操作系统内核，内容如下。  
来自：torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)  
新闻组：comp.os.minix  
主题：在 minix 中你最想看到什么？  
摘要：关于我的新操作系统的小调查  
时间：1991 年 8 月 25 日 20:57:08 GMT  
 每个使用 minix 的朋友，你们好。  
 我正在做一个（免费的）用在 386（486）AT 上的操作系统（只是业余爱好，它不会像 GNU 那样庞大和专业）。这个想法自 4 月份就开始酝酿，现在快要完成了。我希望得到各位对 minix 的任何反馈意见，因为我的操作系统在某些方面与它相类似（其中包括相同的文件系统的物理设计（因为某些实际的原因））。  
 我现在已经移植了 bash（1.08）和 gcc（1.40），并且看上去能运行。这意味着我需要几个月的时间来让它变得更实用一些，并且，我想要知道大多数人想要什么特性。欢迎任何建议，但是我无法保证我能实现它们。:-)  
 Linus（torvalds@kruuna.helsinki.fi）  
就像 Torvalds 所说的，他创建 Linux 的起点是 Minix，由 Andrew S.Tanenbaum 出于教育目的开发的一个操作系统 。  
接下来，如他们所说，这就成了历史。Linux 逐渐发展成为一个技术和文化现象。通过和 GNU 项目的力量结合，Linux 项目发展成了一个完整的、符合 Posix 标准的 Unix 操作系统的版本，包括内核和所有支撑的基础设施。从手持设备到大型计算机，Linux 在范围如此广泛的计算机上得到了应用。IBM 的一个工作组甚至把 Linux 移植到了一块腕表中！

# 1.8 系统之间利用网络通信

我们一般而言把计算机系统作为一个孤立的硬件和软件的结合体。实际上现代的系统经常通过网络和希塔设备连接到一起。于是我们多了一个IO设备——网络适配器。

当系统从主存中复制一串字节到网络适配器的时候，数据流经过网络到达另一台机器，而不是本地的磁盘驱动器。相似的，系统可以读取从其他机器发过来的数据，并且把别的机器的数据放在自己的主存上。随着Internet这样的全球网络的出现，从一台主机复制信息到另外一台主机已经成为了计算机系统最重要的用途之一。

于是乎，我们也可以像在本地用./hello打开HelloWorld程序，也可以在远程的服务器上干这个事情。用一个熟悉的telnet应用连接上远程的telnet服务器等...



# 1.9 重要主题

这个是本章的小结部分了。我们得到了一个重要的观点那就是系统不仅仅是硬件，系统是软硬件结合互相交织的艺术品。他们必须共同协作才能达到运行应用程序的最终目的。

作为本章的结束，最后再强调几个贯穿计算机系统所有方面的重要概念。

## 1.9.1 Amdahl定律

Gene Amdahl对提升系统某一部分性能所带来的效果做出了简单却有见地的观察。这就是*Amdahl定律*。这玩意到底干了什么呢——他的意思是，当我们对系统的某个部分加速时，其对系统整体性能的影响取决于该部分的重要性和加速成都。

例如，

某系统执行某应用所需要的时间为[MISSING IMAGE: , ]

假设系统某部分所需执行时间与该事件的比例为

而该部分的性能提升比例为

即该部分初始所需时间为。现在所需要的时间则为。因此，总的时间则为：



因此，可以计算加速比为为



举个例子，系统的某个部分初始耗时为60%，其加速比例因子为3。那么我们就可以计算得出加速比为1/[0.4+0.6/3]=1.67倍。虽然我们加速了系统中某一个主要的部分做出了重大的改经。但是非常可惜的是系统的加速效率小于这个部分本来应该做到的加速速度。——所以要加速整个系统，就必须把系统中绝大部分的设备速度全部提升上来。

那么就由小朋友要问了：这他妈是啥啊……（其实很简单但是我代数基础非常差只能上分析了）

说实话哈，我也没看明白这里面的关联。不过非常有趣的是因为我薄弱的数学基础和大学期间突然苦练的高数。我倒是可以用导数和极限来处理一下：



当加速倍数K变化的时候（逐渐变大），S也会越来越大，符合我们的直觉：加速某一部分，整体加速。不过还有一个的部分是永恒不变的。这就用到导数了（我的妈啊，高数害死我了）：

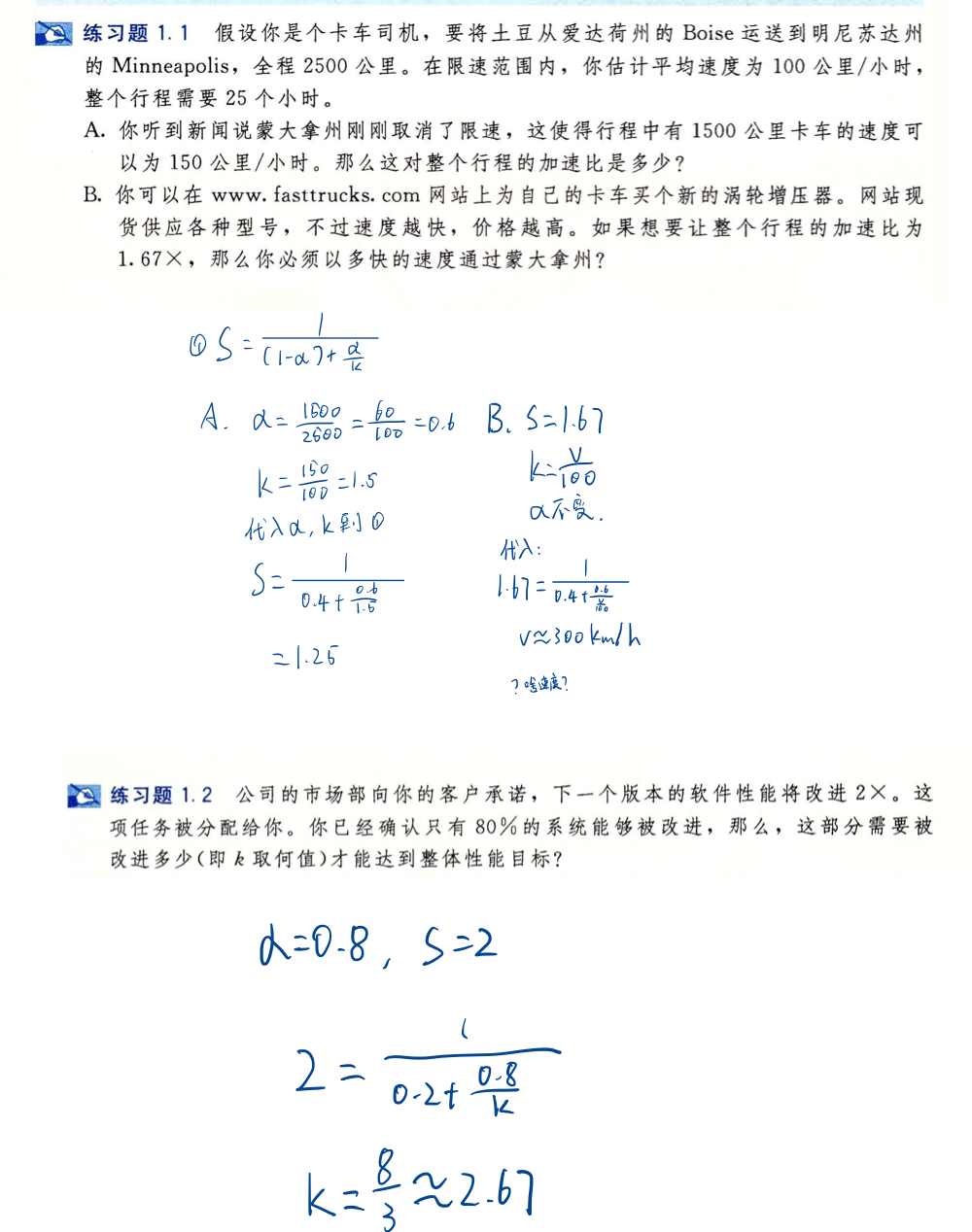






用导数来分析其实就很明显了，这边会计算出一个“加速变化率”，当我的k越来越大的时候，很明显的是整个加速效率会变低——即 k 越大，增加 k 对 S 的提升效果越弱（边际效益递减）。反之要加速整个系统，就必须把系统中绝大部分的设备速度都全部提升上来。

有两道例题：



虽然我代数差，这个应该算给小学生做的。还是很简单的。不过写完题目之后看到里面与预料相差较大的值，也能直观的领悟到这个公式的含义。

## 1.9.2 并发和并行

数字计算机的整个历史中，有两个需求是驱动进步的持续动力：一个是我们想要计算机做的更多，另一个是我们想要计算机做的更快。当计算机能够同时做更多事情的时候，这两个因素都会改进。

我们用术语*并发concurrency*来指代一个同时具有多个活动的系统。

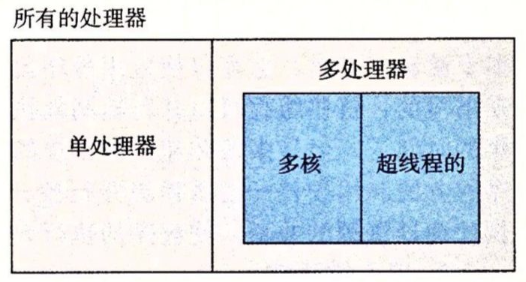
我们用术语*并行parallelism*来指代用并发来让一个系统运行的更快。

并行可以在计算机系统的多个抽象层次上运用：

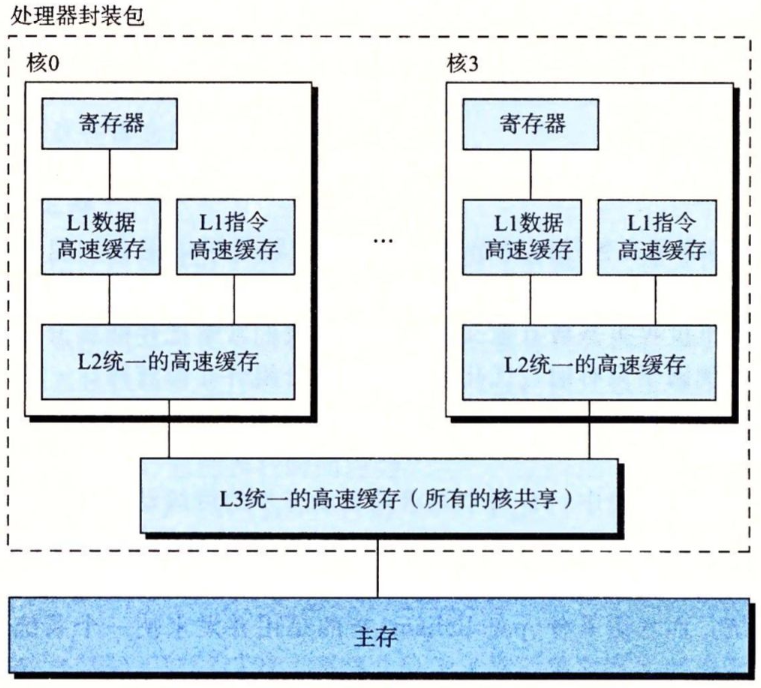
1. 线程级并发

构建在进程这个抽象之上，我们能设计出可以同时多程序执行的系统。这就导致了并发的出现。传统意义上，这种并发执行是模拟出来的假象。在单处理器系统中，处理器必须在多个任务之间切换，大多数实际的计算也都是由一个处理器来完成的。

当构建一个由但操作系统内核控制的多处理器组成的系统时候，我们就得到了一个多处理器系统。随着现在多核处理器和*超线程hyperthreading*的出现，这种系统就越来越常见了。



多核处理器时将多个CPU（叫做核）集成到一个集成电路芯片上。如下图所示：



这是一个典型的多核处理器的组织架构。其中微处理器芯片有四个cpu核，每个核都有自己的L1和L2高速缓存，其中L1高速缓存分为数据和指令的高速缓存。

超线程，有时候称作*同时多线程simultaneous multi-threading*，是一个允许一个CPU执行多个控制流的技术。这种CPU有多个备份例如PC和寄存器，而其它的硬件部分只有一份例如做浮点算术运算的单元。这样的CPU可以更好地利用他的处理资源——在单个周期的基础上决定要执行哪一个线程。比如：一个线程必须等到某些数据被加载到高速缓存中，那么CPU就可以去执行另外一个线程。例如Intel Core i7可以同时让每个核执行两个线程，那么4核系统就可以并行8个线程。

现在的人们突然发现为什么我们不用CPU的这个特性来写程序呢？如此这般多线程也就出现了，它是一种真正的可以算作实现了“并发”的“并行”技术。

1. 指令级并行

在较低的抽象层次上，现代处理器可以同时执行多条指令的属性称之为*指令集并行*。

如果处理器可以达到比一个周期一条指令更快的速度，那么就是*超标量superscalar*处理器。以前不行，不过现代的大部分处理器都支持超标量操作。我们依旧可以对这个进行理解，提升代码效率。

1. 单指令、多数据并行

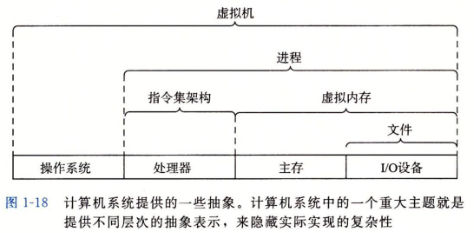
在最低层次上，许多现代处理器拥有特殊的硬件——允许一条指令产生多个可以并行执行的操作，这种方式叫做单指令、多数据。即为SIMD并行。例如，比较新的Intel和AMD处理器都可以并行处理8对单精度浮点数做加法的指令。

提供这些SIMD指令大多是为了提高处理多媒体数据应用的执行速度。虽然有些编译器会试图从C程序中抽取SIMD并行性，但是更可靠的方法是用编译器支持的特殊的向量数据类型来写程序。

## 1.9.3 计算机系统中抽象的重要性

抽象的使用时计算机学科中最为重要的改练至一。例如，为一组函数规定一个简单的应用接口API就是一个很好的编程习惯。程序员无需了解内部的工作原理便可以直接使用。不同的编程语言提供不同形式和等级的抽象，例如java的类和c语言的函数原型。

我们在之前已经讲过抽象了，现在我们在最外面套一个虚拟机——因为一些时候我们需要一个计算机上跑好几个不同的系统或者同一个系统的不同版本。虚拟机就此提供了对整个计算机系统的抽象。



而在后续，我们要把这些隐藏复杂性的“抽象们”全部都拆开来研究到底有“多复杂”了。

# 1.10 小结

计算机系统是由硬件和系统软件组成的，它们共同协作以运行应用程序。计算机内部的信息被表示为一组组的位，它们依据上下文有不同的解释方式。程序被其他程序翻译成不同的形式，开始时是 ASCII 文本，然后被编译器和链接器翻译成二进制可执行文件。

处理器读取并解释存放在主存里的二进制指令。因为计算机花费了大量的时间在内存、I/O 设备和 CPU 寄存器之间复制数据，所以将系统中的存储设备划分成层次结构 ——CPU 寄存器在顶部，接着是多层的硬件高速缓存存储器、DRAM 主存和磁盘存储器。在层次模型中，位于更高层的存储设备比低层的存储设备要更快，单位比特造价也更高。层次结构中较高层次的存储设备可以作为较低层次设备的高速缓存。通过理解和运用这种存储层次结构的知识，程序员可以优化 C 程序的性能。

操作系统内核是应用程序和硬件之间的媒介。它提供三个基本的抽象：

1) 文件是对 I/O 设备的抽象；

2) 虚拟内存是对主存和磁盘的抽象；

3) 进程是处理器、主存和 I/O 设备的抽象。

最后，网络提供了计算机系统之间通信的手段。从特殊系统的角度来看，网络就是一种 I/O 设备。