文本文件、二进制文件

文本文件只由 *ASCII* 字符构成，其他所有文件则为二进制文件。

https://blog.csdn.net/sxhelijian/article/details/29594687

文本文件（*ASCII* 文件）：特殊的二进制文件。人能看得懂的形式。

二进制文件:用和内存中一样的方式保存数据,为机器能理解的形式,人不太容易直接看懂。

通常效率更高。

编译系统

图 1.3 编译系统

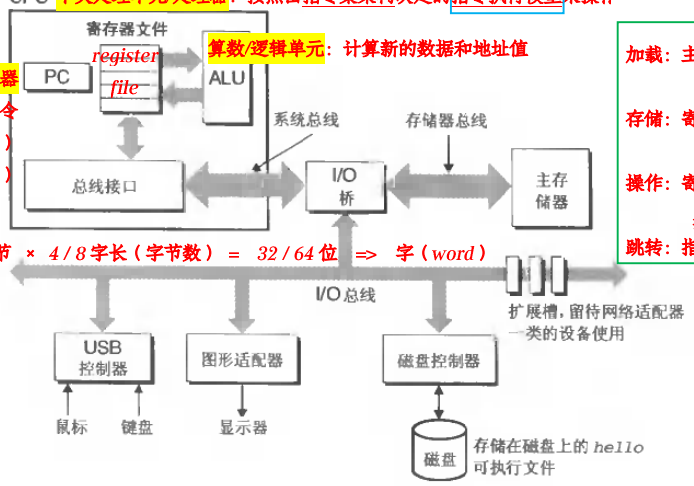
- 预处理阶段。预处理器（cpp）根据以字符#开头的命令（directives），修改原始的 C 程序。比如 hello.c 中第一行的#include <stdio.h>指令告诉预处理器读取系统头文件 stdio.h 的内容，并把它直接插入到程序文本中去。结果就得到了另一个 C 程序，通常是以.i 作为文件扩展名。
- 编译阶段。编译器（cc1）将文本文件 hello.i 翻译成文本文件 hello.s，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切地描述了一条低级机器语言指令。汇编语言是非常有用的，因为它为不同高级语言的不同编译器提供了通用的输出语言。例如，C 编译器和 Fortran 编译器产生的输出文件用的都是一样的汇编语言。
- 汇编阶段。接下来，汇编器（as）将 hello.s 翻译成机器语言指令，把这些指令打包成为一种叫做可重定位（relocatable）目标程序的格式，并将结果保存在目标文件 hello.o 中。hello.o 文件是一个二进制文件，它的字节编码是机器语言指令而不是字符。如果我们在文本编辑器中打开 hello.o 文件，呈现的将是一堆乱码。
- 链接阶段。请注意，我们的 hello 程序调用了 printf 函数，它是标准 C 库中的一个函数，每个 C 编译器都提供。printf 函数存在于一个名为 printf.o 的单独的预编译目标文件中，而这个文件必须以某种方式并入到我们的 hello.o 程序中。链接器（ld）就负责处理这种并入，结果就得到 hello 文件，它是一个可执行目标文件（或者简称为可执行文件）。可执行文件加载到存储器后，由系统负责执行。

计算机系统硬件组成

CPU 从 *PC* 指向的内存处读取指令并解释指令中的位，执行该指令指示的简单操作（加载、存储、操作、跳转），然后更新 *PC* 使其指向下一条指令。

指令集架构：每条机器代码指令的效果

微体系结构：实际上处理器如何实现效果



加载：主存 --复制--> 寄存器

存储：寄存器 --复制--> 主存

操作：寄存器 *A* + 寄存器 *B* --复制--> *ALU* --算术运算--> 寄存器 *X*

跳转：指令 --抽取并复制--> *PC*

图 1.4　一个典型系统的硬件组成

CPU：中央处理单元；ALU：算术/逻辑单元；PC：程序计数器；USB：通用串行总线。

总线

贯穿整个系统的是一组电子管道，称做总线，它携带信息字节并负责在各个部件间传递。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（即字长）是一个基本的系统参数，各

个系统中也不尽相同。比如，Intel Pentium 系统的字长为 4 字节，而服务器类的系统，例如 Intel Itaniums 和高端的 Sun 公司的 SPARCS 的字长为 8 字节。用于汽车和工业中的嵌入式控制器之类较小的系统的字长往往只有 1 或 2 字节。为了便于描述，我们假设字长为 4 字节，并且假设总线一次只传 1 个字。

I/O 设备

I/O（输入/输出）设备是系统与外界的联系通道。我们的示例系统包括四个 I/O 设备：作为用户输入的键盘和鼠标，作为用户输出的显示器，以及用于长期存储数据和程序的磁盘驱动器（简单地说就是磁盘）。最开始，可执行程序 hello 就放在磁盘上。

每个 I/O 设备都是通过一个控制器或适配器与 I/O 总线连接起来的。控制器和适配器之间的区别主要在于它们的组成方式。控制器是 I/O 设备本身中或是系统的主印制电路板（通常被称做主板）上的芯片组，而适配器则是一块插在主板插槽上的卡。无论如何，它们的功能都是在 I/O 总线和 I/O 设备之间传递信息。

第 6 章会更多地说明磁盘之类的 I/O 设备是如何工作的。在第 11 章中，你将学习如何在应用程序中利用 Unix I/O 接口访问设备。我们尤其关注特别有趣的网络类设备，不过这些技术也适用于其他设备。

主存

主存是一个临时存储设备，在处理器执行程序时，它被用来存放程序和程序处理的数据。物理上来说，主存是由一组 DRAM（动态随机存取存储器）芯片组成的。逻辑上来说，存储器是由一个线性的字节数组组成的，每个字节都有自己惟一的地址（数组索引），这些地址是从零开始的。一般来说，组成程序的每条机器指令都由不定量的字节构成。与 C 程序变量相对应的数据项的大小是根据类型变化的。比如，在运行 Linux 的 Intel 机器上，short 类型的数据需要 2 字节，int、float 和 long 类型则需要 4 字节，而 double 类型需要 8 字节。

第 6 章具体说明存储技术，比如 DRAM 是如何工作的，以及它们又是如何组合起来构成主存的。

处理器

中央处理单元（CPU）简称处理器，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被称为程序计数器（PC）的字长大小的存储设备（或寄存器）。在任何一个时间点上，PC 都指向主存中的某条机器语言指令（内含其地址）。[1]

从系统通电开始，直到系统断电，处理器一直在不假思索地重复执行相同的基本任务：从程序计数器（PC）指向的存储器处读取指令，解释指令中的位，执行指令指示的简单操作，然后更新程序计数器指向下一条指令，而这条指令并不一定在存储器中和刚刚执行的指令相邻。

这样的简单操作的数目并不多，它们在主存、寄存器文件（register file）和算术逻辑单元（ALU）之间循环。寄存器文件是一个小的存储设备，由一些字长大小的寄存器组成，这些寄存器每个都有惟一的名字。ALU 计算新的数据和地址值。下面是一些简单操作的例子，CPU 在指令的要求下可能会执行这些操作。

- 加载：从主存拷贝一个字节或者一个字到寄存器，覆盖寄存器原来的内容。
- 存储：从寄存器拷贝一个字节或者一个字到主存的某个位置，覆盖这个位置上原来的内容。
- 更新：拷贝两个寄存器的内容到 ALU，ALU 将两个字相加，并将结果存放到一个寄存器中，覆盖该寄存器中原来的内容。
- I/O 读：从一个 I/O 设备中拷贝一个字节或者一个字到一个寄存器。
- I/O 写：从一个寄存器中拷贝一个字节或者一个字到一个 I/O 设备。
- 转移：从指令本身中抽取一个字，并将这个字拷贝到程序计数器（PC）中，覆盖 PC 中原来的值。

总线：传送定长的字节块。字节 × 4 / 8 字长（字节数） = 32 / 64 位 => 字（word）

批注 [M1]: 控制器和适配器的区别？

高速缓存存储器（*cache memory*，简称 *cache* 或高速缓存）

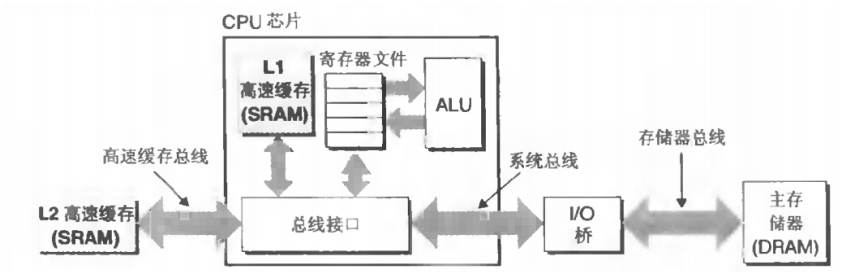


图 1.8　高速缓存存储器

由静态随机访问存储器（*SRAM*）硬件技术实现。

CPU 从各存储器读取一个字的时间对比：

T（磁盘）> *T*（主存）> *T*（*L2* 高速缓存）> *T*（寄存器）≈ *T*（*L1* 高速缓存）

T（磁盘）≈ *T*（主存）× *10000000*

T（主存）≈ *T*（*L2* 高速缓存）× *[5, 10]* ≈ *T*（寄存器）× *100*

存储器层次结构

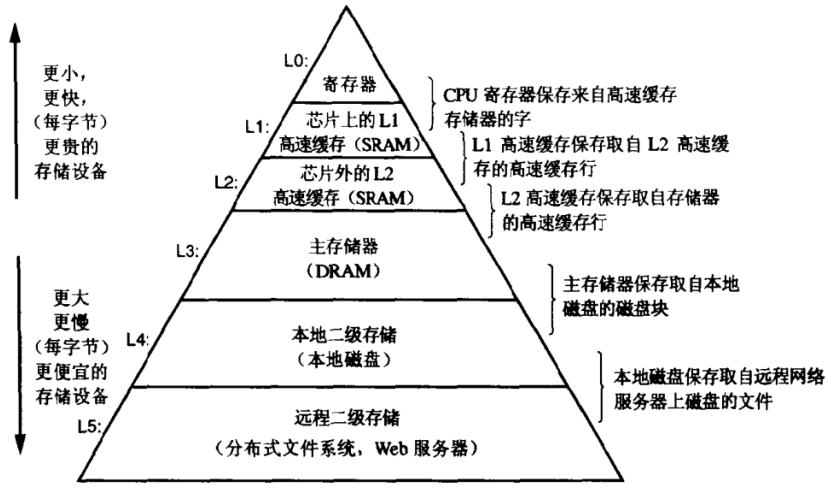


图 1.9　一个存储器层次模型的示例

上一层的存储器是低一层存储器的高速缓存。

操作系统

基本功能：

① 防止硬件被失控的应用程序滥用；

② 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。

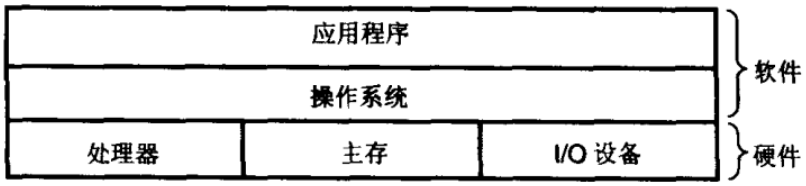


图 1.10　计算机系统的分层视图

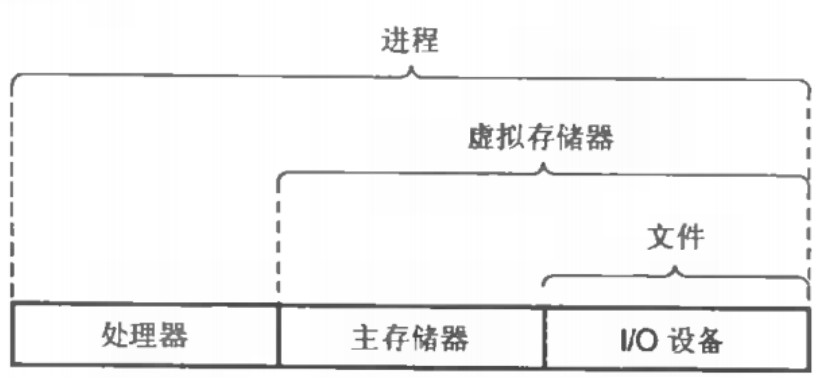


图 1.11　操作系统提供的抽象表示

进程与线程

进程：一个正在运行的程序。由多个线程组成。

线程：执行单元。每个线程都运行在进程的上下文中，并共享同样的代码和全局数据。

并发运行：交错执行进程 *A* 和进程 *B* 的指令。

上下文切换：操作系统实现交错执行的机制。

多核处理器能同时执行多个程序。

操作系统内核（*kernel*）：操作系统代码常驻主存的部分，管理全部进程所用代码和数据结构的集合，不是一个独立的进程。

批注 *[M帐2]:* 是否是进程？

系统调用（*system call*）指令

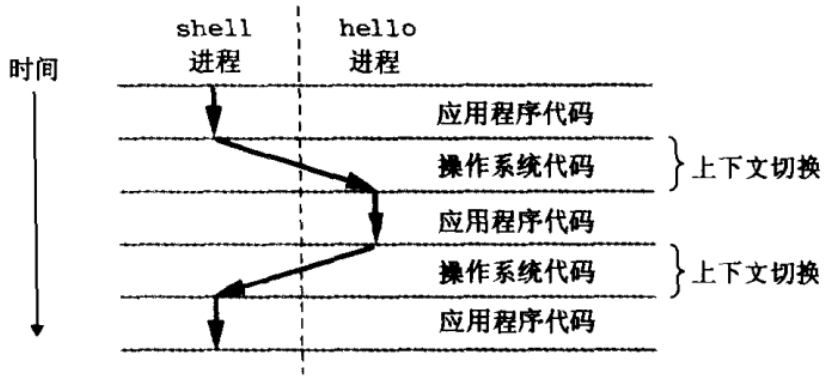


图 1.12　进程的上下文切换

虚拟内存

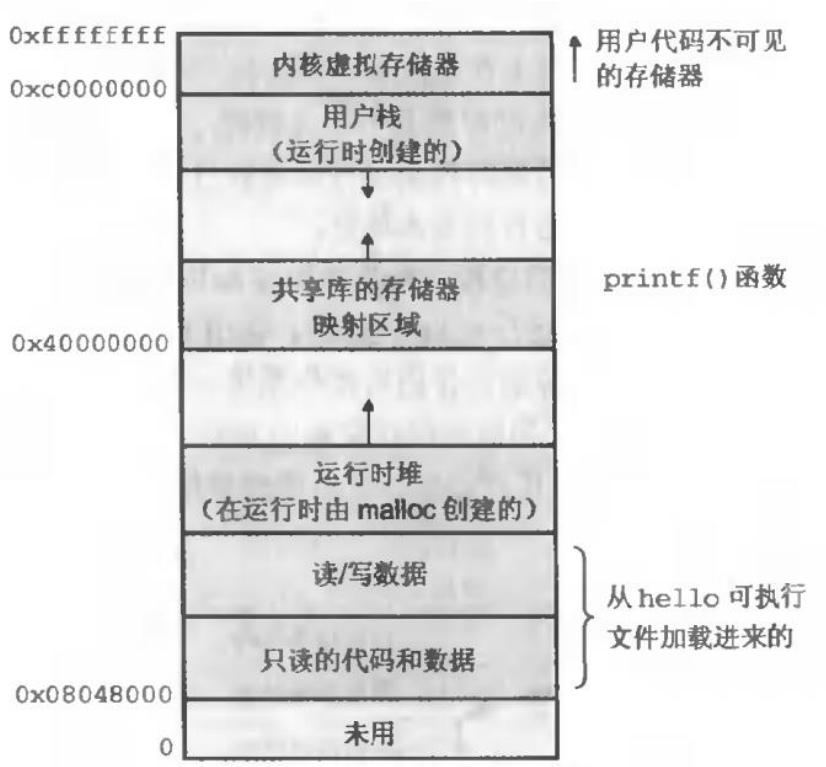


图 1.13　进程的虚拟地址空间

- *程序代码和数据*。代码是从同一固定地址开始，紧接着的是和 C 全局变量相对应的数据区。代码和数据区是由可执行目标文件直接初始化的，在我们的示例中就是可执行文件 hello。在第 7 章我们介绍链接和加载时，你会学习到更多有关地址空间中这部分的内容。
- *堆*。代码和数据区后紧随着的是运行时*堆*。代码和数据区是在进程一旦开始运行时就被指定了大小的，与此不同，作为调用像 malloc 和 free 这样的 C 标准库函数的结果，堆可以在运行时动态地扩展和收缩。在第 10 章学习管理虚拟存储器时，我们将更详细地研究堆。
- *共享库*。在地址空间的中间附近是一块用来存放像 C 标准库和数学库这样*共享库*的代码和数据的区域。共享库的概念非常强大，但是也是个相当难懂的概念。在第 7 章我们学习动态链接时，将学习共享库是如何工作的。
- *栈*。位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。和堆一样，用户栈在程序执行期间可以动态地扩展和收缩。特别地，每次我们调用一个函数时，栈就会增长。每次我们从函数返回时，栈就会收缩。在第 3 章中你将学习编译器是如何使用栈的。
- *内核虚拟存储器*。内核是操作系统总是驻留在存储器中的部分。地址空间顶部的四分之一部分是为内核预留的。应用程序不允许读写这个区域的内容或者直接调用内核代码定义的函数。

文件

文件即字节序列，每个 *I/O* 设备甚至于网络都可以被看成是文件。

网络通信

图 1.15　利用 telnet 跨越网络远程运行 hello

Amdahl 定律

1.9.1 Amdahl 定律

Gene Amdahl，计算领域的早期先锋之一，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。这个观察被称为 Amdahl 定律(Amdahl's law)。该定律的主要思想是，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程度。若系统执行某应用程序需要时间为 T_{old}。假设系统某部分所需执行时间与该时间的比例为 α，而该部分性能提升比例为 k。即该部分初始所需时间为 αT_{old}，现在所需时间为 $(\alpha T_{old})/k$。因此，总的执行时间应为

$$T_{new} = (1-\alpha)T_{old} + (\alpha T_{old})/k = T_{old}[(1-\alpha) + \alpha/k]$$

由此，可以计算加速比 $S = T_{old}/T_{new}$ 为

$$S = \frac{1}{(1-\alpha) + \alpha/k} \tag{1.1}$$

举个例子，考虑这样一种情况，系统的某个部分初始耗时比例为 60%(α=0.6)，其加速比例因子为 3(k=3)。则我们可以获得的加速比为 1/[0.4+0.6/3]=1.67 倍。虽然我们对系统的一个主要部分做出了重大改进，但是获得的系统加速比却明显小于这部分的加速比。这就是 Amdahl 定律的主要观点——要想显著加速整个系统，必须提升全系统中相当大的部分的速度。

旁注　表示相对性能

性能提升最好的表示方法就是用比例的形式 T_{old}/T_{new}，其中，T_{old} 为原始系统所需时间，T_{new} 为修改后的系统所需时间。如果有所改进，则比值应大于 1。我们用后缀“×”来表示比例，因此，“2.2×”读作“2.2 倍”。

表示相对变化更传统的方法是用百分比，这种方法适用于变化小的情况，但其定义是模糊的。应该等于 $100 \cdot (T_{old} - T_{new})/T_{new}$，还是 $100 \cdot (T_{old} - T_{new})/T_{old}$，还是其他的值？此外，它对较大的变化也没有太大意义。与简单地说性能提升 2.2×相比，“性能提升了 120%”更难理解。

并发（*concurrency*）和并行（*parallelism*）

并发（*concurrency*）：一个同时具有多个活动的系统。

并行（*parallelism*）：用并发来使一个系统运行得更快。可在计算机的多个抽象层次上运用。

批注 *IM帐3I:* 并发和并行的区别？

- 线程级并发：在一个进程中执行多个控制流。传统意义上，这种并发执行只是模拟出来的。

  超线程（*hyperthreading*）/ 同时多线程（*simultaneous multi-threading*）：允许一个 *CPU* 执行多个控制流。

- 指令级并行：一个 *CPU* 同时执行多条指令。

  流水线（*pipelining*）

  超标量（*superscalar*）处理器：可以达到比一个周期一个指令更快的执行速率。

- 单指令、多数据并行 / *SIMD* 并行：一条指令产生多个可以并行执行的操作。

抽象

I/O 设备 抽象 ------> 文件

程序存储器 抽象 ------> 虚拟内存

正在运行的程序 抽象 ------> 进程

计算机（包括操作系统、处理器和程序） 抽象 ------> 虚拟机

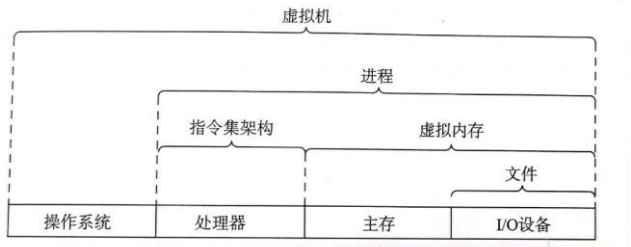


图 1-18 计算机系统提供的一些抽象。计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性