第一单元学习笔记

yinxuhao [xuhao_yin@163.com]

December 16, 2022

Contents

December 12, 2022

1 信息就是位 + 上下文

1. 源程序实际上就是一个由 0 和 1 组成的位 (又称为比特) 序列，8 个位被组织成一组，称为字节。1 byte = 8 bits

2. 系统中所有的信息——包括磁盘文件、内存中的程序、内存网络中存储的的用户的数据，**都是由一串比特表示**。区别不同数据对象的唯一方法是**我们读到的数据对象时的上下文**。

2 程序被其他程序翻译成不同的格式

`gcc -o hello hello.c` 可以读取源文件`hello.c`，并将其翻译成一个可执行目标文件`hello`。翻译过程分为 4 个阶段，执行这四个阶段的程序分别是预处理器、编译器、汇编器和链接器，它们一起构成了编译系统。

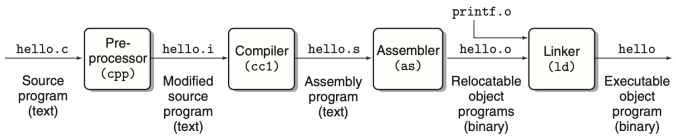

Figure 1: 编译系统

2.1 预处理阶段

`#include <stdio.h>`命令告诉预处理器 (`cpp`) 读取系统头文件内容，并把它直接插入程序文本中。结果得到另一个 C 程序，通常以`.i`作为文件扩展名。

2.2 编译阶段

编译器 (`ccl`) 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。

2.3 汇编阶段

汇编器 (`as`) 将`hello.s`翻译成机器指令，把这些指令打包成一种叫做可重定位目标程序 (relocatable object program) 的格式，并将结果保存在目标文件`hello.o`中。该文件为二进制文件。

2.4 链接阶段

链接器 (`ld`) 负责将预编译的目标文件 (例如`printf.o`) 合并到`hello.o`中。结果是可执行目标文件 (简称可执行文件)`hello`，可以被加载到内存中，由系统执行。

3 了解编译系统的益处

1. 优化程序性能
2. 理解链接时出现的错误
3. 避免安全漏洞

4 运行可执行程序

在shell中执行可执行程序：./hello

shell是一个命令行解释器，它输出一个提示符，等待输入一个命令行，然后执行该命令。

4.1 系统的硬件组成

4.1.1 总线

一组电子管道，它贯穿整个系统。它携带信息字节并负责在各个部件间**传递**。通常被设计成**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，各个系统都不尽相同。64 位系统是 8 个字节。我们需要明确在上下文中一个字的大小。

4.1.2 I/O 设备

系统与外部世界的联系通道。一般由：

作为用户输入的键盘鼠标，作为用户输出的显示器及长期存储数据和程序的磁盘驱动器。

每个I/O设备都通过一个控制器 或适配器 与I/O总线相连。

控制器和适配器之间的区别在于它们的封装方式。控制器是I/O设备本身或者系统的主印制电路板 (主板) 上的芯片组。而适配器是插在主板卡槽上的卡。它们的功能都是在I/O总线和I/O设备之间传递信息。

4.1.3 主存

主存是**临时存储**设备。在处理器执行程序时，用来存放程序和程序处理的数据。

从物理上讲，主存是由一组动态随机存取存储器 (DRAM) 芯片组成的。从逻辑上讲，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)，这些地址从零开始。

4.1.4 处理器

中央处理单元，是解释存储在主存中指令的引擎。

处理器的核心是以一个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都指向主存中某条机器语言指令 (含有该指令的地址)

处理器从系统通电开始，直到系统断电，一直在不断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。处理器看上去是按照一个非常简单的指令执行模型来操作，这个模型是由**指令集架构**决定的。

简单操作包括：

加载：从主存复制一个字节或一个字到寄存器以覆盖寄存器原来的内容。

存储：从寄存器复制一个字节或者一个字到主存的某个位置，以覆盖这个位置上原来的内容。

操作：把两个寄存器的内容复制到ALU，ALU对这两个字做算术运算，并将结果存放到一个寄存器中，以覆盖其原来的内容。

跳转：从指令本身中抽取一个字，并将这个字复制到程序计数器 (PC) 中，以覆盖PC原来的值

需要区分处理器的**指令集架构**和处理器的**微体系结构**：

指令集架构描述的是每条机器代码指令的效果；微体系结构描述的是处理器实际上是如何实现的。

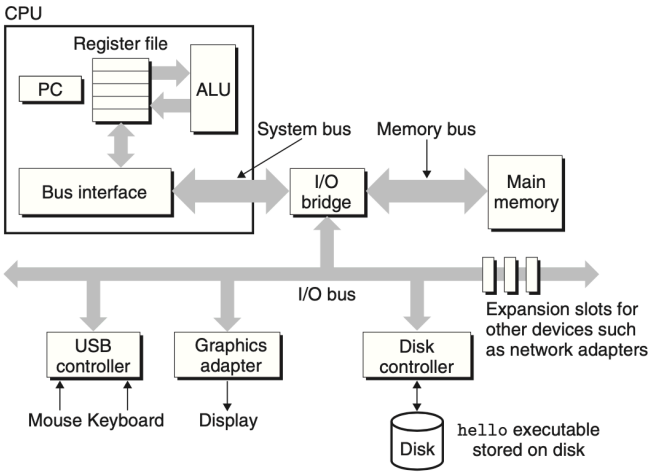

Figure 2: 一个典型系统的硬件组成

4.2 运行 hello 程序

当在shell中键入./hello后，shell将字符逐一读入寄存器，再把它存放到内存中。见图片Figure 3敲击回车后，shell就知道了命令已结束。然后会执行一系列操作来家在可执行的hello文件，这些指令将hello目标文件中的代码和数据从磁盘复制到主存。

利用直接存储器存取 (DMA)，数据可以不通过处理器直接从磁盘到达主存。该步骤可见图Figure 4。

一旦数据被加载到主存，处理器就开始执行机器语言指令。这些指令将"hello, world\n" 字符串中的字节从主存中复制到寄存器文件，再从寄存器文件中复制到显示设备，最终显示在屏幕上，见图Figure 5.

December 14, 2022

5 高速缓存

系统总是花费大量时间把信息从一个点那个挪到另一个地方，这种复制操作减慢了程序真正的工作。

较大的存储设备要比较小的存储设备运行得慢，而高速设备的造价远高于同类低速设备。例如，一个寄存器只有几百字节的信息，主存里可容纳几十亿字节。处理器从寄存器中读数据比从主存中读数据几乎快 100 倍。

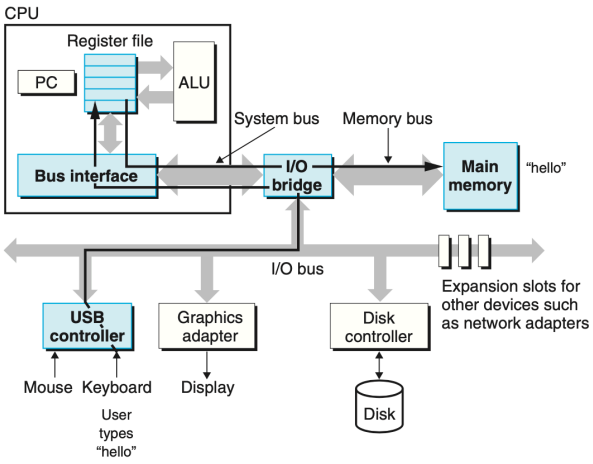

Figure 3: 从键盘上读取 hello 命令

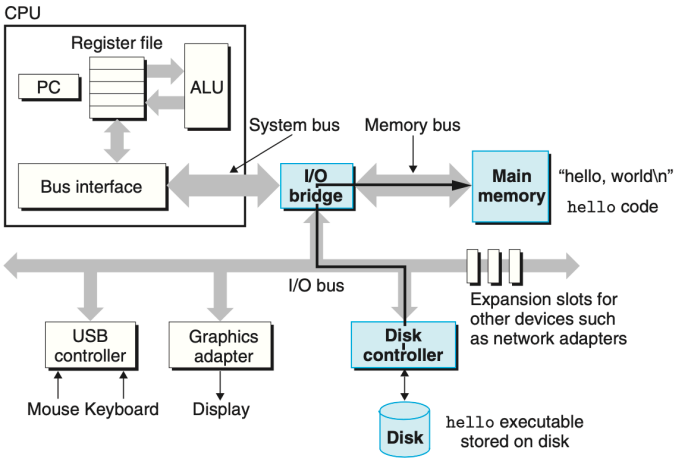

Figure 4: 从磁盘加载可执行文件到主存

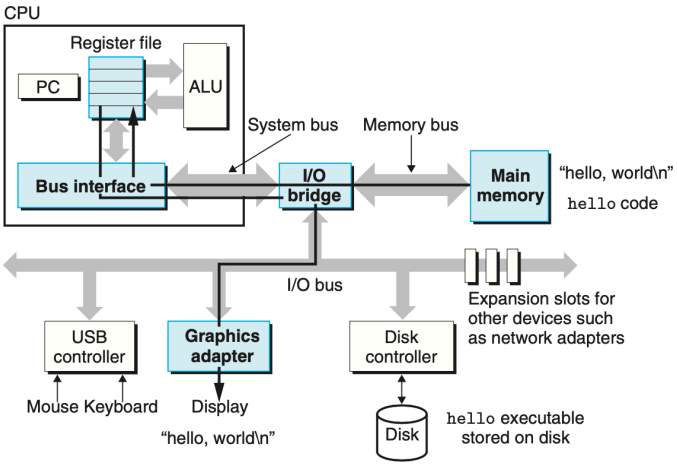

Figure 5: 将输出字符串从存储器写到显示器

针对这种现象，系统设计者采用了高速缓存存储器 (cache memory) 作为临时的字节区域，存放处理器近期可能会需要的信息。Figure 6展示了一个典型系统的高速缓存存储器。

高速缓存存储器存在的应用程序员能够利用高速缓存将系统的性能提高一个数量级。

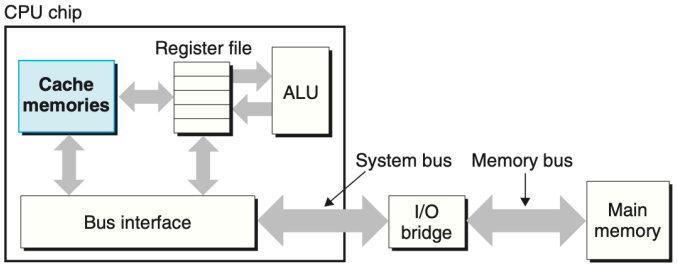

Figure 6: 高速缓存存储器

6 存储设备形成的层次结构

上述问题的一个解决方案就是在处理器和一个较大较慢的设备 (比如主存) 之间插入一个更小更快地存储设备 (比如高速缓存)。如此就形成了一个存储器层次结构，如图Figure 7所示。其主要思想是**上一层的存储器作为低一层存储器的高速缓存**。

7 操作系统

所有引用程序对硬件的操作尝试都必须经过操作系统。

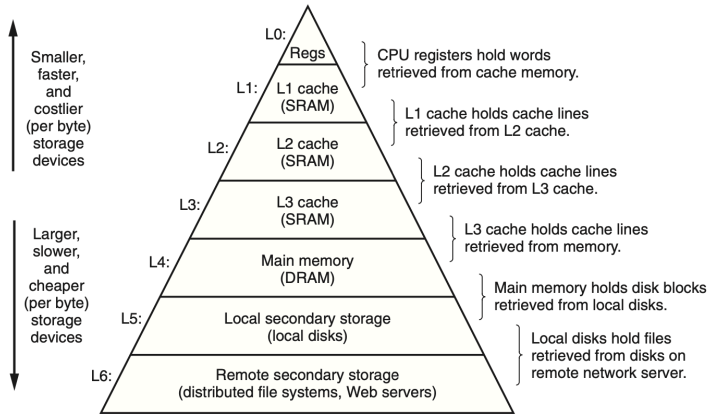

Figure 7: 一个存储器层次结构的示例

操作系统有两个基本功能：

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。

操作系统的抽象概念有：进程、虚拟内存和文件。

进程是对处理器、主存和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

文件是对I/O设备的抽象表示

7.1 进程

进程是操作系统对一个正在运行的程序的一种抽象。一个 CPU 看上去像是在并发的执行多个进程，这是处理器通过进程间切换实现的。操作系统实现这种交错执行的机制称为**上下文切换**.

操作系统保持跟踪进程运行所需的所有状态信息，这种状态，就是**上下文**。里面包含了诸如PC和寄存器文件的当前值，以及主存的内容。

在任一时刻，单处理器系统都只能执行一个进程的代码。当操作系统决定把控制权从当前进程转移到某个新进程时，会执行上下文切换，即保存当前进程的上下文，恢复新进程的上下文，然后将控制权传递到新进程。新进程书接上文，继续运行。见Figure 8所示。

内核不是一个独立的进程，它是系统管理全部进程所用代码和数据结构的集合。

7.2 线程

一个进程由多个称为线程的执行单元组成，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。

7.3 虚拟内存

它使得每个进程看到的内存都是一致的，称为虚拟地址空间。

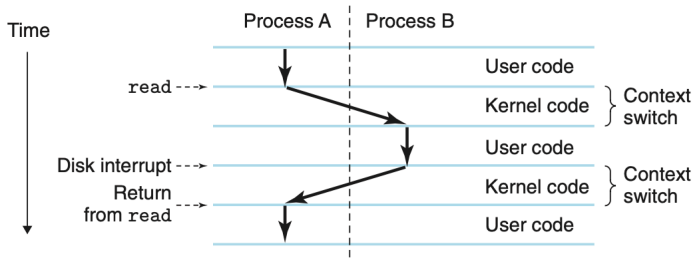

Figure 8: 进程的上下文切换

7.4 文件

文件就是字节序列，仅此而已。每个I/O设备，包括磁盘，甚至网络，都可以看作文件。

8 Amdahl 定律

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。

若系统执行某应用程序需要的时间为 T_{old}，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 $\frac{\alpha T_{old}}{k}$。因此，总的执行时间应为：

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

由此可以计算加速比 $S = T_{old}/T_{new}$ 为

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

December 16, 2022

Amdahl定律一个有趣的特殊情况是考虑k趋向于 ∞ 时的效果。因为这意味着，我们可以取系统的某一部分将其加速到一个点，在这个点上，该部分花费的时间忽略不计。由此可得：

$$S_\infty = \frac{1}{(1-\alpha)}$$

。

Amdahl 定律描述了改善任何过程的一般原则。

9 并发和并行

并发concurrency是一个通用的概念，指一个同时具有多个活动的系统；而术语**并行**parallelism指的是用并发来使一个系统运行得更快。

并行可以在计算机系统的多个抽象层次上运用。

9.1 线程级并发

传统意义上，并发执行只是**模拟**出来的，是使用一台计算机对它正在执行的进程间快速切换来实现的，就好像一个杂耍艺人保持多个球在空中飞舞一样。

这种并发形式允许一个用户同时从事多个任务，大多数实际计算也都由一个处理器来完成。这种配置称为**单处理器系统**。

多处理器系统: 随着**多核处理器**和**超线程**的出现而变得常见。

多核处理器是将多个核 (CPU) 集成到一个集成电路芯片上。

超线程：有时称为同时多线程，是一项允许一个CPU执行多个控制流的技术。它涉及CPU某些硬件有多个备份，比如程序计数器和寄存器文件，而其他的硬件只有一份，比如执行浮点算术运算的单元。。

比如，假设一个线程必须等到某些数据被装载到高速缓存中，那CPU就可以继续去执行另一个线程。

多处理器的使用,可以减少执行多个任务时模拟并发的需要，可以使程序运行得更快。但这要求程序以多线程的方式书写。

9.2 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为**指令级并行**。

流水线 (pipelining) 的使用能够使得执行速率接近于一个时钟周期一条指令。

如果处理器可以达到比一个周期一条指令更快的执行速率，就称之为**超标量** (superscalar)

9.3 单指令、多数据并行

现代许多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，这种方式称为单指令、多数据 (SIMD).

提供这些SIMD指令多是为了提高处理影像、声音和视频数据应用的执行速度。

10 计算机系统中抽象的重要性

抽象 的使用是计算机科学中最为重要的概念之一。例如，应用程序接口 (API) 使得程序员无需了解它内部的工作便可以使用该API。不同的编程语言提供不同形式和等级的抽象支持。

在处理器里，**指令集架构**提供了对实际处理器硬件的抽象。该抽象使得机器代码表现得好像运行在一个一次只执行一条指令的处理器上。

文件是对I/O设备的抽象，虚拟内存是对程序存储器的抽象，而进程是对一个正在运行的程序的抽象。**虚拟机**，提供对整个计算机的抽象，包括操作系统、处理器和程序。

各种抽象见下图:

11 小结

计算机系统由系统软件和硬件组成

其内部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

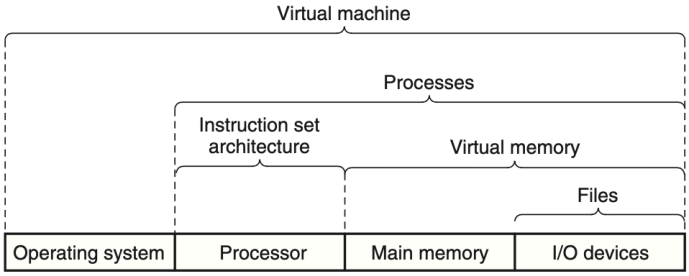

Figure 9: 计算机系统提供的一些抽象。计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性

计算机花费了大量时间在内存、I/O设备及 CPU 寄存器之间复制数据，故将系统中的存储设备划分层次结构，较高层次作为较低层次设备的高速缓存。

操作系统内核是应用程序和硬件之间的媒介。它提供三个抽象：

文件 ← I/O设备

虚拟内存 ← 主存和磁盘

进程 ← 处理器、主存和I/O设备

网络提供了计算机系统之间通信的手段。从某种特殊角度看，网络就是一种I/O设备。