CSAPP NOTES

INRAINBOWS

2022 年 7 月 11 日

目录

Chapter 1

计算机系统漫游

```c
#include <stdio.h>

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

本章，通过跟踪该 hello 程序的生命周期进行学习。

1.1 信息就是位 + 上下文

大部分计算机使用 ASCII 标准表示字符，即用单字节整数值表示各个字符。图 1-1 为 `hello.c` 的 ASCII 表示。

#	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	SP	p	r	i	n	t	f	(	"	h	e	l
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	

图 1-1. hello.c 的 ASCII 文本表示

- hello 程序的生命周期由其源程序（文件）开始，即 `hello.c`。
- 源程序是由 0 和 1 组成的位 *bit* 序列。
- 8 个 bit 一组，称为字节 *byte*。
- 各字节表示程序中的某些文本字符。
- 每个文本行由一个不可见的'`\n`'结尾。
- 像 `hello.c` 这样仅含 ASCII 字符的称为*文本文件*，其它的都叫*二进制文件*。
- `hello.c` 的启示：系统中的所有信息—磁盘文件、内存程序、内存存放的用户数据以及网络传输数据，均是由比特序列表示的。
- 区分不同数据对象的方法：分辨读取这些数据对象时的上下文。比如，在不同上下文中，一个相同的字节序列可能表示一个整数、浮点数、字符串或机器指令。

我们需要了解数字的机器表示，因为它们与实际整数和实数不同，是对真值的有限逼近。第二章有详细描述。

1.2 程序被其它程序翻译成不同的格式

为使 `hello.c` 在系统上运行，每条 C 语句必须被其它程序转为一系列低级机器语言指令。这些指令被打包为可执行目标程序格式，并以二进制文件形式存放于磁盘。目标程序也称可执行目标文件。

在 Unix 上，源文件到目标文件的转换由编译器套件完成：`linux> gcc -o hello hello.c`

GCC 读取源程序 `hello.c`，并将其编译为可执行目标文件 `hello`。此编译过程分四个阶段：

1. 预处理阶段：预处理器 *cpp* 将 `hello.c` 中以 # 开头的命令进行展开（`#define`、`#include`），并得到新文件 `hello.i`。
2. 编译阶段：编译器 *cc1* 将 `hello.i` 的 C 语句编译为汇编语句，变为 `hello.s`。如下所示：

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

3. 汇编阶段：汇编器 *as* 将 `hello.s` 编译为机器指令，并打包为可重定位目标程序 `hello.o`。此时 `hello.o` 为二进制文件。
4. 链接阶段：注意 `hello` 调用了 `printf` 函数。该函数被置于 C 标准库预编译好的 `printf.o` 中，它须以某种方式被合并到 `hello.o` 中。链接器 *ld* 负责完成该操作，并得到 `hello` 可执行文件。

1.3 了解编译系统如何工作是大有益处的

有一些重要原因促使程序员了解编译系统是如何工作的：

- 优化程序性能。
- 理解链接时出现的错误。
- 避免安全漏洞。

1.4 处理器读并解释存储在内存中的指令

现在 `hello.c` 已被编译为 `hello` 可执行文件。要在 Unix 上运行之，可运行以下 shell 指令：

```
linux> ./hello
hello, world
linux>
```

shell 是一个命令行解释器，它输出一个提示符，等待一个命令行输入，并执行该命令。若命令行的第一个单词非内置 shell 指令，则 shell 假设其为一个可执行文件的名字，shell 加载并运行之。

1.4.1 系统的硬件组成

图 1-2 展示了 Intel 系统产品族的模型。

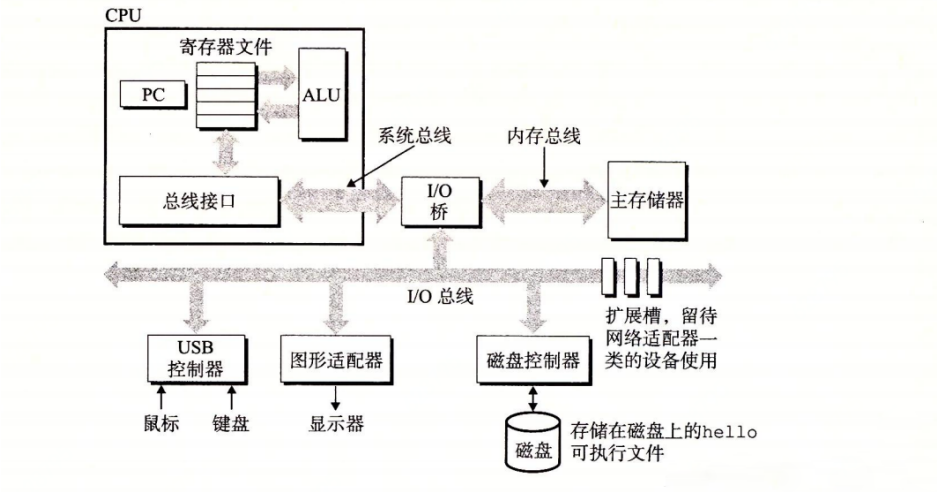


图 1-2. 一个典型系统的硬件组成
CPU: 中央处理单元；ALU：算数逻辑单元; PC：程序计数器；USB：通用串行总线

1. 总线——贯穿整个系统的一组电子管道。其携带信息字节在各部件间传递。总线传送的是定长字节块，即字 (word)。字包含的字节数（字长）与目标系统相关。大多数机器字长为 4bytes/32bits 或 8bytes/64bits。

2. I/O 设备——在图例中包含四个 I/O 设备：作为输入的鼠标键盘、作为输出的显示器、磁盘驱动器。各 I/O 设备均由一个控制器或适配器与 I/O 总线相连。控制器与适配器的区别为封装方式。控制器是 I/O 设备自身或主板上的芯片组；适配器为一块插在主板上的卡。无论如何，他俩的功能相同。
第六章详解了磁盘 I/O 工作原理；第十章讨论了 Unix I/O 接口的使用。

3. 主存——一种临时存储设备，其存放处理器需要的程序与数据。主存在物理上由若干动态随机存取存储器 (DRAM) 芯片组成；在逻辑上是一个线性的字节数组，每个字节存放一个地址，地址从零开始。第六章将具体介绍存储器技术。

4. 处理器——中央处理单元 (CPU)。负责解释执行主存中的指令。CPU 核心为一个大小为一个 word 的寄存器，即程序计数器 (PC)。PC 在任何时候均指向主存中的某条机器指令地址。CPU 从通电到断电，遵循指令集架构决定的操作模型来进行一系列操作。这些操作围绕主存、寄存器文件 (register file) 与 ALU 进行。寄存器文件由一些名字不同的单字长寄存器组成；ALU 计算新的数据与地址。下面是一个简单例子：

   - 加载：从主存负值一个 byte 或一个 word 到寄存器，以覆盖寄存器原本内容。
   - 存储：从寄存器负值一个 byte 或一个 word 到主存某处，以覆盖该处原本内容。
   - 操作：将两个寄存器的内容负值到 ALU，ALU 对这两个 word 做算术运算，将结果存入某寄存器中，并覆盖该寄存器原本内容。
   - 跳转：从指令本身抽取一个 word，将其复制到 PC 中，以覆盖 PC 原值。

   实际上 CPU 使用了复杂机制来加速程序执行。因此将 CPU 的指令集架构与 CPU 微体系结构区分开来。指令集架构描述每条机器指令的效果；微体系结构描述 CPU 实际上的实现方式。第三章讨论了指令集架构，第四章则着重介绍微体系结构。第五章展示了一个现代 CPU 的工作模型。

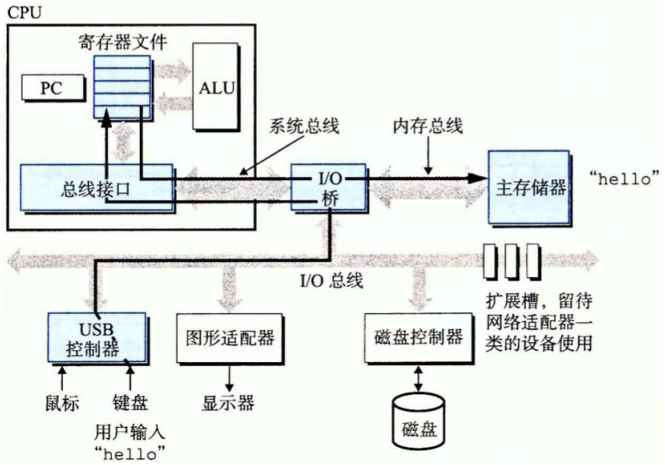


图 1-3. 从键盘上读取 hello 命令

1.4.2 运行 hello 程序

当在键盘上输入"./hello"后，shell 将字符逐一读入寄存器，再放进内存，如图 1-3。

敲击回车后，shell 执行一系列指令加载 `hello`，将 `hello` 目标文件中的代码与数据复制到主存。

直接存储器存取 (DMA，第六章介绍) 技术可使数据不通过 CPU 而直达主存，如图 1-4。

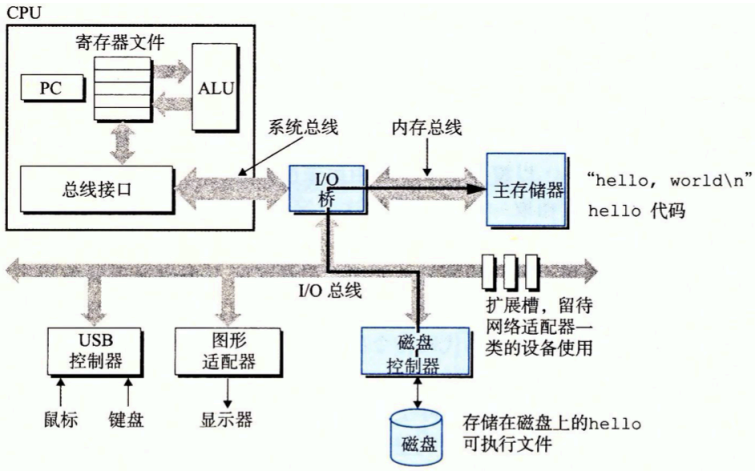


图 1-4. 从磁盘加载可执行文件到主存

`hello` 被加载完成后，CPU 开始执行 `hello` 中 `main` 函数中的机器指令。指令将串"`hello, world\n`"中的字节从主存复制到寄存器文件，接着从寄存器文件复制到显示设备并显示，如图 1-5。

1.5 高速缓存至关重要

根据机械原理，较大存储设备比较小的运行得慢，而快速设备造价远高于低俗设备。典型的便是寄存器文件和主存，CPU 从寄存器中读数据比主存要快 100 倍。针对这种差异，可采用高速缓存存储器 (cache memory，简称 cache、高速缓存) 作为 CPU 需要信息的临时存放点。图 1-6是一个典型系统中的高速缓存。

- CPU 上的 L1 cache 速度与寄存器文件差不多。
- L2 cache 由一条特殊总线与 CPU 连接，其速度比 L1 慢 5 倍，但仍比主存快 5 10 倍。
- L1 与 L2 是用静态随机访问存储器 (SRAM) 实现的，现代 CPU 有 L3 cache。

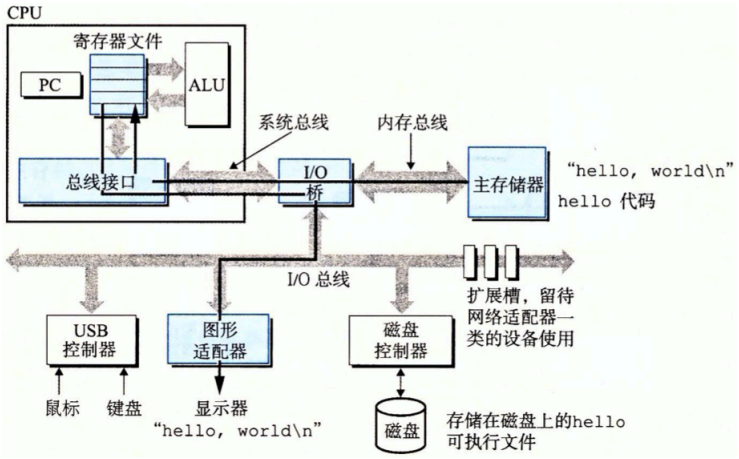


图 1-5. 将输出字符串从存储器写到显示器

- cache的*局部性原理*：cache中会尽可能存放程序经常在主存中访问的数据与代码，使得大部分主存操作可在cache中完成。

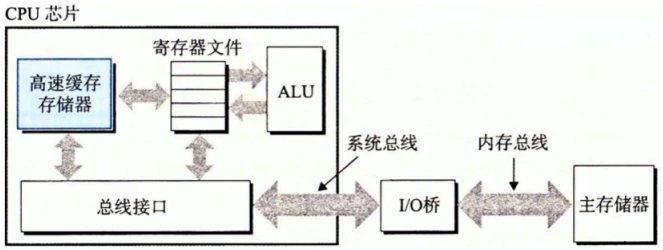


图 1-6. 高速缓存存储器

1.6 存储设备形成层次结构

- 每个计算机系统中的存储设备都被组织成了一个*存储器层次结构*，如图 1-7。
- 从上至下/速度越慢/容量越大/逐字节造价约便宜。
- 该层次结构思想为上一层存储器作为第一层存储器的高速缓存。

1.7 操作系统管理硬件

操作系统 (OS) 是程序与硬件之间的媒介，如图 1-8。OS 两个基本功能：(1) 防止程序滥用硬件；(2) 为程序提供不同硬件的统一抽象。OS 通过几个抽象概念来实现这两功能（进程、虚拟内存、文件），如图 1-9。

1.7.1 进程

*进程*是 OS 对运行时程序的抽象。一个 OS 上可同时运行多个进程，各进程像是独占地在使用硬件。*并发运行*指一个进程与另一个进程的指令交错执行。在现代单/多核 CPU 系统中，单 CPU 看起来像并发地执行多个进程，这是通过处理器的进程切换实现的。OS 这种交错执行机制称为*上下文切换*。

- OS 保持跟踪进程运行状态信息。这种状态便是*上下文* (context)

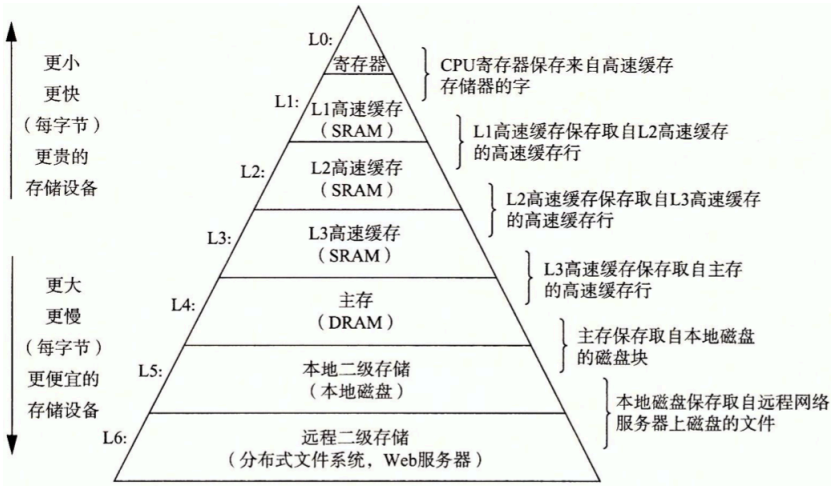


图 1-7. 一个存储器层次结构的示例

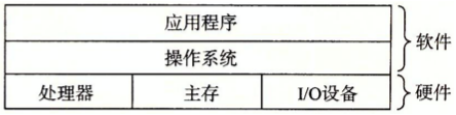


图 1-8. 计算机系统的分层视图

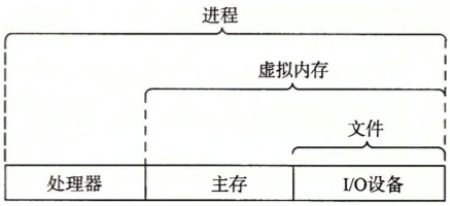


图 1-9. 操作系统提供的抽象表示

- 上下文包含 PC、寄存器文件、主存等内容。
- 单 CPU 系统仅能执行一个进程的代码，OS 可以通过*上下文切换*在进程之间转移控制权。
- *上下文切换*保存当前进程的上下文，恢复新进程的 context，然后将控制权转到新进程。如图 1-10

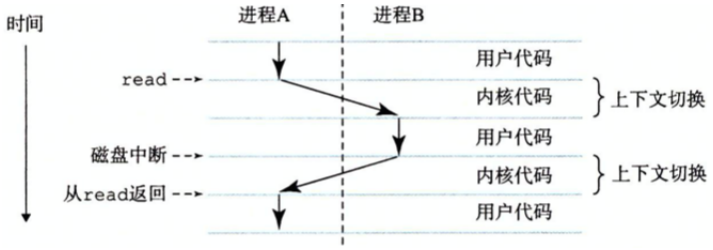


图 1-10. 进程的上下文切换

在之前例子中有两个并发进程：shell 与 `hello`。首先是 shell 在运行，敲入命令后，shell 调用系统 Api 执行请求，此时控制权交由 OS。OS 保存 shell 的 context 并创建新 `hello` 进程及其 context，接着将控制权交由 `hello` 进程。它终止后，OS 恢复 shell 的 context，并将控制权传回它。

如图 1-10，进程间切换由操作系统*内核* (kernel) 管理。kernel 是常驻主存的部分 OS 代码。程序需调用 OS 功能时，执行一条 *系统调用* (system call) 指令将控制权传给 kernel；kernel 执行请求操作并返回程序。

kernel 非独立进程，其为 OS 管理全部进程所用代码与数据结构的集合。

1.7.2 线程

现代 OS 中单进程可由多个叫做*线程*的执行单元组成，各线程运行在进程的 context 中，共享相同代码与全局数据。线程间数据共享比进程间容易，因为线程一般比进程高效。

1.7.3 虚拟内存

虚拟内存是一个抽象概念。

1.8 重要主题

1.8.1 Amdahl 定律

思想：对系统某部分进行加速的操作对系统整体性能的影响取决于该部分的**重要性**与**加速程度**。

- 系统执行某程序所需时间为 T_{old}.
- 系统某部分所需执行时间与 T_{old} 比例为 α.
- 该部分性能提升比例为 k.
- 可得出，该部分初始执行时间为 αT_{old}，加速后时间为 $(\alpha T_{old})/k$.
- 则系统新总执行时间为：$T_{new} = (1-\alpha)T_{old} + (\alpha T_{old})/k = T_{old}[(1-\alpha) + \alpha/k]$

即可得出加速比：

$$\begin{aligned} S &= T_{old}/T_{new} \\ &= \frac{1}{(1-\alpha)+\alpha/k} \end{aligned} \tag{1.1}$$

练习题 1.1　运送土豆，全程 2500km。因为限速，你的平均速度为 100km/h，全程耗时 25h。

A. 若行程中总长 1500km 的蒙大拿州速度可以达到 150km/h，则整体行程加速比为多少?
解 1：$t_{part} = 1500km/150km/h = 10h$，则 $T_{new} = (2500km - 1500km)/100km/h + 10h = 20h$。
所以 $S = T_{old}/T_{new} = 25h/20h = 1.25$。
解 2：易知 $\alpha = 1500km/2500km = 0.6$、$k = 150/100 = 1.5$
则根据 Amdahl 公式 (1.1)，$S = 1/[(1-0.6) + 0.6/1.5] = 1.25$。

B. 若想让整体加速比为 1.67X，则必须以多块的速度通过蒙大拿州?
解 1：$S = T_{old}/T_{new} = 25h/T_{new} = 1.67 \Rightarrow T_{new} = 2500/167$。而 $T_{new} = 10h + 1500km/V$
由题意，$10h + 1500km/V \leq 2500/167 \Rightarrow V \geq 301.8km/h$
对 V 取整，则速度至少为 302km/h。
解 2：根据 Amdahl 公式 (1.1)，$S = 1/[(1-0.6) + 0.6/k] = 1.67$
解得 $k \approx 3.02$，则 $V = 3.02 \times 100km/h = 302km/h$。

练习题 1.2　下版本软件性能改进 2X，该任务分配给你。你已确认仅 80% 的系统能被改进。则该部分需改进多少才可达到整体性能目标?
解：由题意，$2 = 1/[(1-0.8) + 0.8/k]$，解得 $k = 2.67$

Amdahl 定律一个特殊情况是 $k \to \infty$ 的情况，此时对应子部分加速到几乎不花时间，则有：

$$S_\infty = \frac{1}{(1-\alpha)} \tag{1.2}$$

从这个式子看出，即使局部被无限加速，整体的提升仍是有限的。

Chapter 2

信息的表示和处理

我们将研究三种数字表示：

1. 无符号 (unsigned) 编码基于传统二进制，表示大于等于零的数字。
2. 补码 (two's-complement) 编码表示有符号整数，即可以为正或负的数字。
3. 浮点数 (floating-point) 编码表示实数的科学计数法的以 2 为基数的版本。

2.1 信息存储

- 大多数计算机使用字节 (byte)，一个 8bits 的块作为最小内存寻址单位。
- 虚拟内存 (virtual memory)，各字节由一个唯一数字表示并作为其地址 (address)
- 所有可能地址的集合称为虚拟地址空间 (virtual address space)。
- VAS 的实际实现（第九章）是将 DRAM、闪存、磁盘、特殊硬件与 OS 软件结合，为机器级程序提供一个看上去统一的字节数组。

2.1.1 十六进制表示法

- byte 的值域（二进制）：$00000000_2 \sim 11111111_2$；十进制下为 $0_{10} \sim 255_{10}$。
- 二进制太冗长，十进制与位模式转换比较麻烦。替代方案为十六进制 (hexadecimal)，简写 hex。
- hex 使用'0' ~ '9' 与字符'A' ~ 'F' 表示 16 个值。一个 hex 数字对应四个二进制位。
- byte 的值域（十六进制）：$00_{16} \sim FF_{16}$
- C 语言中 hex 值以 `0x/0X` 开头；'A' ~ 'F' 非大小写敏感。
- binary 转 hex：每 4 位一组转一个 hex 数字。若 binary 的位模式不是 4 的倍数，将最左边一组用 0 补足。

练习题 2.1详见实物书 Page 26。该题涉及技巧——设值 $x = 2^n$，则将 x 转为 hex 形式的方法：

- 首先清楚 x 的 binary 表示为 1 后面跟 n 个 0，且 $0_{16} \Leftrightarrow 0000_2$。
- $n \div 4 = j \cdots i \quad (0 \geq i \geq 3)$。则目标 hex 形式为：$2^i$ 开头且后跟 j 个 0。

- 比如 $x = 2048 = 2^{11}$，易知 $j = 2, i = 3$，则 x 的 hex 表示为 `0x800`

练习题 2.2详见实物书 Page 26。该题涉及技巧——dec 与 hex 之间的互转：

- dec 数 x 转 hex：将被除数 x 与除数 16 应用于短除法，结果为余数从下到上的排列表示。
- hex 转 dec：用 hex 的各位数乘以相应 16 的幂，幂的次数从 0 开始从右到左递增。

2.1.2 字数据大小

- 每台计算机均有一个字长 (word size)，其为指针的标称大小 (nominal size)。
- 虚拟地址以这样的一个 word 进行编码。
- 若机器的 word size 为 w bits，则其虚拟内存地址范围为 $0 \sim 2^w - 1$，程序最多访问 2^w 个 bytes。

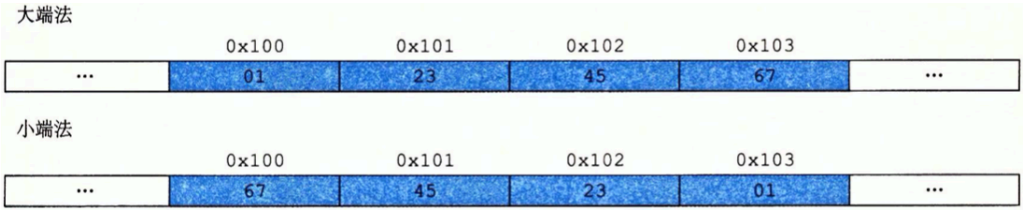
基础知识：1Byte = 8Bits ; 1KB = 1024Bytes ; 1MB = 1024KBs ; 1GB = 1024MBs ; 1TB = 1024GBs
拓展：1GB = 2^30Bytes ; 1MB = 2^20Bytes ; 1KB = 2^10Bytes

- 32 位机器的虚拟地址空间最高能访问 2^{32}bytes，即 4GB，也即刚超过 4×10^9bytes。而 64 位机器为 16EB，约 1.84×10^{19}bytes。
- 作为向后兼容，多数 64 位机器可运行 32 位程序，反过来则不行。
- C 的数据类型 `char` 在直觉上是用来存储单个字符的，但也常被用来存储整数值。
- 程序员应力图使程序具备可移植性，其中一方面便是让程序对不同数据类型的确切大小不敏感。

2.1.3 寻址和字节顺序

- 小端法 (little endian)：在内存中按从最低到最高有效字节的顺序存储对象。
- 大端法 (big endian)：与小端法相反。

打个比方，`x` 类型为 `int`，其值为 `0x01234567`，位于地址 `0x100`。以下是两种表示法的区别：



2.1.4 表示字符串

详见实物书 Page 34。

2.1.5 表示代码

详见实物书 Page 35。

2.1.6 布尔代数简介

网络旁注：关于布尔代数和布尔环的更多内容

- 考虑长度为 w 的位向量上的 ^、& 和 ~ 运算，得到一种叫*布尔环* (Boolean ring) 的数学形式。
- 整数运算中一个属性是每个值 x 均有一个*加法逆元* (additive inverse)$-x$，使得 $x + (-x) = 0$。Boolean ring 中对应的运算为 ^，此时每个元素的加法逆元是其自身。
- 也就是说，对任何值 a，a ^ a = 0。这里的 0 表示全 0 的位向量。因此会有 (a ^ b) ^ a = b。

2.1.7 C 语言中的位级运算

练习题 2.10详见实物书 Page 38。（该题涉及技巧——利用 Boolean ring 逆元运算达到值交换目的）

位级运算常见用法是实现*掩码运算*，掩码是一个位模式，表示从一个字中选出的位集合。
比如，掩码 `0xFF` 表示一个字的低位字节。比如 `x=0x89ABCDEF` 与该掩码的运算结果为 `0x000000EF`。

练习题 2.12详见实物书 Page 39。该题涉及技巧——掩码运算
练习题 2.13详见实物书 Page 39。该题涉及技巧——掩码运算

2.1.8 C 语言中的逻辑运算

- C 语言逻辑运算符 `||`、`&&`、!，分别对应命题逻辑中的 OR、AND、NOT。
- 逻辑运算和位运算本质区别：
  1. 逻辑运算中，所有非零值均视为 TRUE，零自身为 FALSE。
  2. 若第一个值就能确定表达式结果,则逻辑运算符会无视第二个值。比如 `a&&5/a` 不会发生零除错误,`p&&*p++` 不会导致空指针引用。
- TRUE 与 FALSE 对应返回值分别为 1 与 0。

2.1.9 C 语言中的移位运算

C 语言提供一组*移位*运算。

- 左移运算 `x<<k` 为左移 k 位，丢弃最高的 k 位，在右端补 k 个 0。
- 右移运算分*逻辑右移*和*算术右移*：
  1. 逻辑右移在左端补 k 个 0.
  2. 算术右移在左端补 k 个最高有效位的值。
- 假设位模式长度为 w，k 值一般介于 $[0, w-1]$。

C 未明确定义对 signed 数使用哪种右移，不过大多编译器的实现为默认算术右移。对于 unsigned，右移只能是逻辑的。

2.2 整数表示

实物书 Page 42 列出了 CSAPP 用于精确定义和描述计算机编码方式和操作整数的术语。

2.2.1 整型数据类型

详见实物书 Page 42。

2.2.2 无符号数的编码

设一个整数数据类型有 w 位，以及位向量 $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$。

原理：无符号数编码定义（$\doteq$ 表示左被定义为右）

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \tag{2.1}$$

举个例子：$B2U_4([1011]) = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$。

此时 w 位可表示值范围为：

- 最小值：[00...0]，即整数值 0。
- 最大值：[11...1]，即整数值 $UMax_w \doteq \sum_{i=0}^{w-1} 2_i = 2^w - 1$，令 $w = 4$，则 $UMax_4 = B2U_4([1111]) = 15$。

2.2.3 补码编码

原理：补码编码定义（$\doteq$ 表示左被定义为右）对向量 $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$：

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.2}$$

最高有效位 x_{w-1} 称*符号位*，其权重为 -2^{w-1}。该位为 1 时表示值为负，反之为正。

举个例子：$B2T_4([1011]) = -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -5$。

此时 w 位可表示值范围为：

- 最小值：[10...0]，整数值为 $TMin_w \doteq -2^{w-1}$。
- 最大值：[01...1]，整数值为 $TMax_w \doteq 2^{w-1} - 1$。

值得注意的点：补码范围不对称：$|TMin| = |TMax| + 1$。这是因为一半的符号位 1 的位模式表示负数，另一半符号位 0 的为非负数；而 0 也是非负数，占据了一个位置，使得能表示的整数比负数少一个；另外，在补码表示中-1 与 $UMax$ 有相同的位模式：[11...1]。

网络旁注：有符号数的其它表示法

1. **反码** (Ones' Complement)，最高有效位权为 $-(2^{w-1} - 1)$：
   $B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$
2. **原码** (Sign-Magnitude)，最高有效位为符号位：
   $B2S_2(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot (\sum_{i=0}^{w-2} x^i 2^i)$

注意补码 (Two's complement) 和反码 (Ones' complement) 的撇号位置不同。术语补码来源于：对于非负数 x，用 $2^w - x$ 计算 $-x$ 的 w 位表示；而术语反码来源于：用 $[111...1] - x$ 计算 $-x$ 的反码表示。

Chapter 3

程序的机器级表示

Chapter 4

处理器体系结构

Chapter 5

优化程序性能

Chapter 6

存储器层次结构

Chapter 7

链接

Chapter 8

异常控制流

Chapter 9

虚拟内存

Chapter 10

系统级 I/O