内部交流　请勿外传

数字系统 II 补充教材

南京大学　电子科学与工程学院

方元　编

2022 年　夏

目录

第 1 章

微处理器基础

电子计算机的产生与发展，是人类历史上一次深刻而伟大的科学技术革命，将我们带入信息时代。它对人类历史发展的影响是第一次工业革命、第二次工业革命所不能相比的。目前，它与人类社会已融为一体，并日益广泛而深入地影响着社会生活的方方面面。

1.1 微处理器的发展

1.1.1 电子计算机的诞生

目前我们所使用的电子计算机，其诞生最早可追溯到第二次世界大战之前。

20 世纪初，英国科学家弗莱明[1] 和美国工程师弗雷斯特[2]先后发明了电子二极管和三极管 (或称真空管)，人类跨入了电子时代。20 世纪 30 年代，一些目光敏锐的学者已经注意到，使用电子管技术在机械式手摇计算机的基础上大大提高计算速度的可能性。1937 年，美籍保加利亚学者阿塔纳索夫 [3] 由于在求解数学方程时遇到困难而对计算技术发生了兴趣，并考虑引进电子技术。在他的一名学生贝利 (Clifford E. Berry) 的帮助下开始实施他的计划。1942 年，这台机器终于问世，并被命名为 “ABC 计算机”(Atanasoff-Berry Computer)。虽然这台计算机没有解决他们的数学问题，并且也没有保留下它的原样，但作为第一台电子数字计算机却是当之无愧的。图 1.1 是 ABC 的复制品，现展示于爱荷华州立大学达勒姆中心。

在第二次世界大战期间，对战各方出于军事上的迫切需要，开始利用电子技术研究可以进行快速计算的机器。为了破解德国的恩尼格玛 (Enigma) 加密系统，盟军招募了一批民间数学家和密码专家进行秘密的破译工作，艾伦 · 图灵[4]赫然在列。他们为战胜法西斯作出了杰出的贡献。2014 年上映的电影《模仿游戏》(The Imitation Game) 反映的就是这一情节。图灵也成为现代计算机理论的开山鼻祖。美国计算数学学会 1966 年开始以他的名字设立的奖项 Turing Award，是目前计算机领域中公认的最高荣誉。

[1]John Ambrose Fleming(1864.11.29 –1945.4.18)，生于英国兰卡斯特，英国电气工程师和物理学家。

[2]Lee De Forest(1873.8.26–1961.7.30)，生于美国康瑟尔布拉夫斯。

[3]John Vincent Atanasoff (1903.10.4–1995.6.15)，美籍物理学家，生于美国纽约，1930 年获得维斯康辛大学麦迪逊分校理论物理学哲学博士学位。在他的母国保加利亚有多所以他的名字命名的学校、研究机构及各种荣誉奖项。

[4]Alan Mathison Turning(1912.6.23–1954.6.7)，英国计算机科学家、数学家，计算机科学与人工智能之父。生于英国伦敦，1938 年获得普林斯顿大学博士学位。

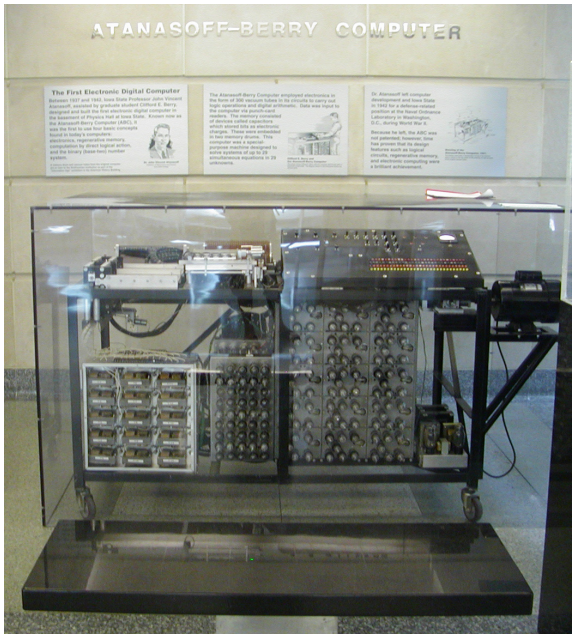


图 1.1: 爱荷华州立大学达勒姆中心的 ABC 复制品

1.1.2 电子计算机的发展

以电子管作为逻辑元件构造的计算机被认为是第一代电子计算机。电子管体积庞大, 耗电量高。1947 年, 美国物理学家巴丁、肖克利和布拉顿[5]发明了晶体管。用晶体管代替电子管作为逻辑元件, 是第二代电子计算机的标志。随着集成电路的出现和发展, 计算机也经历了从中小规模逻辑电路的发展到大规模、超大规模集成电路 (**V**ery **L**arge **S**cale **I**ntegration, VLSI) 时代。此时的计算机开始向微型化和大型化两个方向发展。微型计算机及其各种应用产品已遍及我们生活的各个方面, 而巨型计算机和超级计算机则是各国高技术领域竞争的主战场之一。

早期的计算机主要用于数学计算, 如今则远远超出了计算领域。它不同于以往人类发明的代替人的体力的机器, 人们开始将分析能力、逻辑推理能力乃至学习能力赋予计算机, 使其具有人类的思维能力, 进而代替人类的智力。由此看来, 这样的机器称作 “电脑” 可能更为恰当。

1.2 计算机的数学基础

1.2.1 数制

人类社会中普遍采用的是十进制 (decimal) 计数法, 早期的计算机也有的采用的是十进制计数, 如大名鼎鼎的 ENIAC[6]。但十进制和计算机并没有天然联系。现在的计算机几

[5] John Bardeen (1908.5.23–1991.1.30), 生于威斯康星麦迪逊, 1936 年获普林斯顿大学博士学位。William Shockley(1910.2.13–1989.8.12), 美国物理学家, 生于英国伦敦。1936 年获得麻省理工学院博士学位。Walter Houser Brattain(1902.2.10–1987.10.13), 生于中国厦门, 美国物理学家。1929 年获得明尼苏达大学博士学位。以上三人因发明晶体管共同获得 1956 年诺贝尔物理学奖。

[6] **E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer, 二战后期由美国陆军资助、宾夕法尼亚大学莫尔电气工程学院设计制造, 1946 年 2 月 14 日完成并于次日投入使用。

乎全部都是以二进制 (binary) 为基础的, 因为在自然界中找到二元对立统一的事物最容易: 有和无、高和低、亮与暗..... 在电子计算机中, 电路通和断、电压的高和低很容易构成一对确定的逻辑关系, 它们自然也成为构成电子计算机的逻辑基础。在数字逻辑电路中, 习惯上用 0 表示低电平, 1 表示高电平。理论上说, 反过来表示也是允许的, 这种情况我们称之为 “负逻辑”。

对于任意进制 (radix) r 来说, 一个数的表示方法 $A_{n-1}A_{n-2}\ldots A_0A_{-1}\ldots A_{-m}$ 和它的真值 v 有如下关系:

$$\begin{aligned} v &= A_{n-1}r^{n-1} + A_{n-2}r^{n-2} + \ldots + A_0r^0 + A_{-1}r^{-1} + \ldots + A_{-m}r^{-m} \\ &= \sum_{i=-m}^{n-1} A_i r^i \end{aligned} \tag{1.1}$$

A_i 的取值范围为 0$\sim r-1$。当 $r=10$ 时, 就是我们常用的十进制计数体系。在二进制体系中, A_i 只有两种取值: 0 和 1, 并且它们之间的基本计算形式也非常简单, 例如, 加法运算只有 0+0 、0+1、1+0 和 1+1 四种情况。只是在书写一个数字时, 可能要用到的符号比较多。用二进制表示一年的天数需要 9 位数, 阅读起来会很累。不过这是人类的问题, 计算机自己并不操心。

利用 (1.1) 式可以直接将一个二进制序列转换成十进制 (或任意进制)。而将一个十进制转换成二进制, 式 (1.1) 给我们提供了长除法和长乘法的思路: 对于整数部分, 不断地将商用 2 整除, 保留余数, 最后将余数串接起来, 就是所求的结果。而对于小数部分, 则是不断地将小数部分乘以 2, 记录整数出现 0 和 1 的序列, 最后将这个 0、1 序列串接起来即为所求结果。需要注意的是, 将一个二进制小数转化成十进制, 有可能是 “乘不尽” 的, 常常是一个无限不循环小数。

人类在学习和使用计算机数值时, 为了避免书写或阅读一长串二进制造成的感官疲劳, 又引入了一些特殊的进制, 比较常用的是八进制 (octet) 和十六进制。以十六进制 (Hexadecimal) 为例, 16 恰好是 2 的 4 次方。在公式 (1.1) 中, $r=2$, 以小数点为中心, 向左、向右每四位分为一组, 提出公因子 2^4, 将基换成 $r=16$, 用 0 到 9 这 10 个数字以及 A 到 F 这 6 个字母一共 16 种符号表示每组内的一个可能的组合, 就形成了十六进制。为了区分不同的进制, 在数的尾部用对应英文单词的首字母加以区分。以十进制 248.625 为例, 表 1.1 是以上提到的几种进制表示方法。

表 1.1: 不同进制表示的 248.625

进制	英文单词	表示方法
二进制	Binary	11 111 000.101B
八进制	Octave	370.5O (370.5Q)
十进制	Decimal	248.625D
十六进制	Hexadecimal	F8.AH

针对表1.1有几点说明:

1. 当一个数使用十进制表示时, 符号 D 可以省略, 不会造成误解;
2. 由于作为表示八进制的字母 O 容易和数字 0 混淆, 有时候也会用 Q 表示八进制。此处的 Q 不对应任何单词。

3. 十六进制的首位有可能是字母, 这在一些计算机程序语言中容易造成与变量名的混淆。为避免这样尴尬的问题, 有时候会在字母前面补一个前导 0。

1.2.2 码制

确定了二进制基础以后, 下面要解决的问题是如何扩展数的表示范围。首先要解决的问题是, 如何表示正数和负数。这就出现了数的编码 (encoding) 问题。

计算机中, 所有的信息都只能以二进制表示, 表示正、负也必须用二进制。最容易想到的办法是将一个二进制序列中的某一位用来表示符号, 而将其余位表示这个数的绝对值。习惯上, 用最高位作为符号位。如果一个数字是负数, 则符号位是 1, 否则是 0。用这种编码方式表示的数字, 我们称之为 “原码” (sign-magnitude)。

用原码表示的数字和人的思维方式接近, 看上去比较直观, 容易比较大小。但对计算机来说并不是一个好的选择, 因为它不能直接进行计算, 符号位必须单独处理。例如, 2 个 8 位的二进制数 0000 0010B 和 1000 0010B, 它们分别对应 +2 和 −2。二者相加, 结果是 1000 0100B, 无论如何不能与正确的数学运算结果建立联系。因此, 在计算机中, 数值量更多地是采用 “补码” (two’s complement) 的形式表示。

补码建立在原码的基础上, 它的规则是:

1. 对于正数和 0, 补码即是原码;
2. 对于负数, 在其正数的原码基础上, 各位按位逻辑取反 (即将 1 变成 0、将 0 变成 1, 称为 “反码” (one’s complement)), 再加 1, 即得到补码。

补码的最高位不仅反映了该数的符号, 也可以直接参与运算。仍以上面的 +2 和 −2 为例, 它们的补码分别是 0000 0010B 和 1111 1110B。二者相加, 结果是 1 0000 0000B。由于参与运算的单元只有 8 位, 因此最高位的 “1” 不会被保留。只要运算过程没有溢出 (overflow), 0000 0000B 就是正确结果。

反码也是可以直接进行加减运算的。两个数反码按位相加, 如果有进位, 将进位位移到最低位再做一次加法, 结果就是两个数相加的反码。减法的运算规则类似。

1.2.3 进位与溢出

上面的例子中, 最高位多出的 “1” 被称为 “进位” (carry) [7]。虽然我们直接扔掉了这个多出的一位, 仍然得到了正确的结果, 但实际情况并不总是正确的。我们看下面的例子:

$$\begin{array}{rr} & 99 \\ + & 100 \\ \hline & 199 \end{array} \quad \rightarrow \quad \begin{array}{rr} & 0110\ 0011 \\ + & 0110\ 0100 \\ \hline & 1100\ 0111 \end{array}$$

用 8 位二进制表示 99 (0110 0011B) 和 100 (0110 0100B), 将这两个数字相加, 结果是 199 (1100 0111B)。根据补码的规则, 1100 0111B 对应 −57 的补码。两个正数相加怎么会得到负数呢? 究其原因, 我们发现, 8 位二进制补码由于牺牲了最高位用于表示符号, 导致实际表示数字的范围缩小, 只能在 −128 到 +127 之间。运算结果超出这个范围则不能正确表示。我们把出现这种情况的运算结果称为 “溢出”。

溢出和进位很容易造成混淆, 但它们确实是完全不同的两个概念。从上面的两个例子也可以看到: 有进位的不一定溢出, 而产生溢出的也不一定发生进位。计算机根据最高位

[7]超出计算单元的位数部分的高一位被称为进位位。一般地说, 没有进位位, 也可视为进位为 0。习惯上说的有进位或无进位, 指的是运算结果在这一位上产生了 1 还是 0。

和次高位的进位情况判断溢出: 当最高位和次高位的进位不一致时则是溢出, 反之则不溢出。针对实际问题处理时, 进位是无符号数运算结果需要考虑的问题, 而溢出则是有符号数运算结果需要考虑的问题。

1.2.4 定点数和浮点数

以上介绍的数值表示方法, 无论是原码还是补码, 都属于定点数 (fixed point)。“定点”是指小数点位置固定的意思。在定点格式中, 表示小数点不需要额外的信息位, 只需要在处理数据时默认小数点的位置。当小数点位置对齐后, 加减运算与整数加减运算完全相同。纯整数是定点数的一个特例, 默认小数点在最低位之后。另一个特例是纯小数, 小数点在符号位之后、最高位之前。纯小数的优点是乘法运算不会产生溢出, 这可以在一些应用场合下简化程序设计。

所有处理器都支持定点运算。定点数所能表示的数值范围受到计算机字长的限制。字长 (word) 指二进制位数, 它通常与计算机的位数密切相关。在 16 位的计算机系统中, 单次运算指令最多可以处理 16 位的二进制运算。而一个 16 位的二进制可以表示的无符号最大数值只有 65535 ($65535 = 2^{16} - 1$)。即使在 32 位的计算机上, 一个定点数也只能表示大约 42 亿范围以内的数字 ($2^{32} - 1$), 用于统计世界人口已经不够了。如果想表示更大的数字, 一种可能的方法是将多个 32 位拼接成更多的位数, 而由此带来的负面影响是, 在进行数值运算时, 计算量会成倍增加。即使如此, 用这种方法表示数量级相差很大的数仍不现实。例如以 g 为单位的电子质量 (9.1×10^{-28}g), 需要 4 个 32 位二进制单元, 才能在小数点后面补上足够的 0, 以描述如此微小的数字。

为了在有限位的前提下, 尽量扩大数的表示范围, 计算机软件中引入了浮点数 (floating point) 的概念。在这种数值表示方案中, 小数点的位置不是固定的, 而是浮动的, 它占用一部分二进制位来表示 2 的幂次, 即指数 exponent部分 (又称阶码), 用来表示小数点浮动的位置; 剩余的位用来表示有效数字 (significand), 称作尾数部分。完整的浮点数格式由下面 4 部分表示:

阶符	阶码	尾符	尾数

除符号位以外, 其它位的长度由设计要求决定。例如:

$$24.5 = 0.110001B \times 2^5$$

如果用阶符 1 位、阶码 3 位、尾符 1 位、尾码 7 位表示 24.5 这个浮点数, 可以写成 0101 01100010B, 这里阶码和尾数都采用补码形式。根据不同应用场合, 阶码和尾数也可以选择其他码制。此外, 同一个数字也可以用不同的指数和尾数组合表示, 例如上面的数字在同样的格式下也可以写成 0110 00110001B 或 0111 00011000B (由于字长限制, 小数点前移导致精度损失)。习惯上, 将尾数的绝对值在 0.5 到 1 之间的表示方法称作规格化浮点数, 否则称为非规格化浮点数。规格化浮点数可以充分利用尾数的精度。

从 1970 年代起, 国际上统一采用了 IEEE[8] 标准浮点格式。IEEE 754 标准规定的浮点数由三部分构成:

IEEE 754 (对应 ISO/IEC 60559) 规定了多种不同精度的浮点格式标准, 以常用的单精度和双精度浮点格式为例, 说明如下:

- 符号位 (sign bit), 1 位。0 表示这个数为正, 1 表示负数。

[8] **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers, 全球最大的专业技术组织, 始建于 1963 年, 总部在纽约。

表 1.2: IEEE 754 单精度浮点数格式

s	e	m	Number
0	0	0	0
1	0	0	-0
s	0	$\neq 0$	$(-1)^s \times 0.m \times 2^{-126}$
s	$0 < e < 255$	m	$(-1)^s \times 1.m \times 2^{e-127}$
0	255	0	$+\infty$
1	255	0	$-\infty$
s	255	$\neq 0$	NaN(not a number)

- 指数部分 (exponent), 8 位或 11 位, 表示尾数中的小数点左移或右移的位数。为了保证指数是正数, 规定指数部分为实际指数和一个固定的偏码之和。IEEE-754 标准中规定了单精度 (single precision) 浮点数 (32 位) 的指数部分是 8 位, 偏码 (biased exponent) 为 127; 双精度 (double precision) 浮点数 (64 位) 的指数部分是 11 位, 偏码是 1023。
- 尾数 (mantissa), 23 位 (单精度) 或 52 位 (双精度)。表示数的有效数值部分, 其整数部分为 1。由于在二进制中, 一个非 0 数字的最高位总是 1, 为了充分利用精度, 这个 “1” 作为默认, 不出现在机器码中。

例如, 有下面的浮点数

4				1				C				4				0				0				0				0			
0	1	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s	**exponent**							**mantissa**																							

$$
\begin{aligned}
\text{s} &= 0 \\
\text{e} &= 1000011\text{b}=128+2+1=131 \\
\text{m} &= 1.10001...00\text{b}=1.53125 \\
\text{该数} &= (-1)^0 \times 1.53125 \times 2^{(131-127)} = 1.53125 \times 2^4 \\
&= 24.5 \text{ (十进制)}
\end{aligned}
$$

浮点方案大大延伸了数值范围。单精度浮点数表示的最大/最小非 0 绝对值可以接近 $2^{\pm 128}$, 相当于 $10^{\pm 38}$, 双精度浮点数更可以达到 $10^{\pm 308}$。由于浮点数格式的复杂性, 进行数值计算也相当复杂, 需要将指数、尾数分解后再作处理, 处理结果还需要将二者按规则重新拼合。在没有硬浮点处理能力的计算机上, 浮点运算需要通过定点算法的函数调用来实现, 一次浮点加减运算通常相当于几十条定点指令。这对于定点处理器是一个不小的挑战。

1.2.5 BCD 码

由于我们日常生活主要采用的是十进制, 计算机在数值输入输出时常需要用到二进制和十进制之间的转换, 由此也产生了用二进制代码表示十进制的特殊形式, 这种形式就是 BCD 编码 (**B**inary **C**oded **D**ecimal)。

BCD 码用四位二进制表示一个十进制数码。由于四位二进制有 16 种状态, 故理论上 BCD 码可以有不止一种形式。最常用的是 8421 码, 即用 8、4、2、1 依次对应从高到低这四位的权重, 它们与十进制的 0 到 9 恰好一一对应。

写成十六进制的 BCD 码和十进制码在形式上是完全一样的, 比如 75D = 0111 0101BCD, 而 0111 0101B = 75H。

BCD 码的运算也要遵从十进制运算规则。一些处理器专门设计了 BCD 运算指令, 它们通常是建立在二进制运算基础上的数值调整。事实上, 使用 BCD 码进行数值运算并不方便, 它们主要用于人机交互的场合。

1.3 非数值信息的表示

如今的计算机应用已经远远超出了数值计算领域, 除了对数值信息进行计算处理以外, 还要处理大量的非数值信息, 包括文字、图像、声音等等。这些非数值信息在计算机内部也是用二进制形式表示的。不同类型的信息, 表示方法各不相同。

1.3.1 ASCII 码

为了表示文字信息, 美国国家标准学会 (**A**merican **N**ational **S**tandard **I**nstitute, ANSI) 于 1963 年开始制定美国信息交换标准编码 (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange, ASCII)。它包括数字符号、英文大小写字母、标点符号以及当时电传打字机的一些控制符号等等共计 128 个符号。该标准几经修订, 并被国际标准化组织 (**I**nternational **O**rganization for **S**tandardization) 接纳 (ISO/IEC 646), 目前已成为表示英文字母和符号的国际标准。character encoding

表 1.3: ASCII 字符集

0XXX X000	000	001	002	003	004	005	006	007
000 0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
000 1	BS	HT	LF	VT	FF	CR	SO	SI
001 0	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
001 1	CAN	EM	SUB	ESC	FS	GS	RS	US
010 0	SP	!	"	#	$	%	&	'
010 1	(	)	*	+	,	-	.	/
011 0	0	1	2	3	4	5	6	7
011 1	8	9	:	;	<	=	>	?
100 0	@	A	B	C	D	E	F	G
100 1	H	I	J	K	L	M	N	O
101 0	P	Q	R	S	T	U	V	W
101 1	X	Y	Z	[	\	]	^	_
110 0	`	a	b	c	d	e	f	g
110 1	h	i	j	k	l	m	n	o
111 0	p	q	r	s	t	u	v	w
111 1	x	y	z	{	|	}	~	DEL
	008	009	00A	00B	00C	00D	00E	00F

ASCII 码是一种信息交换代码, 它并不直接表示计算机处理的数值。这一点在使用数字符号时很容易引起混淆。当计算机在显示屏上打印出 “123” 这个数字时, 一般是依次打印 “1”、“2”、“3” 的 ASCII 码, 而存储在计算机中的数值信息则可能是 0111 1011B, 它和我们看到的打印输出信息的编码 0011 0001B、0011 0010B、0011 0011B 没有什么相似性。在真值和打印输出符号之间需要做一些转换工作, 这些转换工作通常是通过软件完成的。无论是打印还是文本保存, 计算机处理的数值都需要转换成特定的编码, 信息才能与人交流。

ASCII 码集一共定义了 128 个字符, 可以用 7 位二进制表示。通常计算机的存储单元以字节 (byte) 计, 一个字节由 8 位二进制构成。多出的一位可用来表示校验信息, 用以检验信息在传递过程中是否出现了错误。一个比较简单的校验方式是奇偶校验 (parity), 即数据收发双方约定每个数据中 1 的个数是奇数 (odd) 还是偶数 (even), 校验位用来标识其奇偶性。当在数据传输过程中发生了一次 0 到 1 或是 1 到 0 的错误时, 接收方就可以结合校验位知道数据在传输中出了问题。这种方法适合于出错概率较低的场合, 且只能检测出错误, 无法知道具体是哪一位出了错误。

1.3.2 汉字编码

中国很早就开始了汉字的计算机标准化工作。中国国家标准总局于 1980 年公布了 “国家标准信息交换用汉字编码基本字符集 (GB2312–1980)”, 共收入汉字六千多个, 除了次常用汉字区的个别的繁体字 (如 “後”) 以外, 全部是简化汉字; 此外还包括一些西文字符、日文、俄文、希腊字母及常用制表符号等等。该标准规定一个字符用两个字节表示。整个字符集分成 94 个区, 每区有 94 个位, 每个区位上只有一个字符, 因此可用所在的区和位来对汉字进行编码, 因此又叫区位码。将区、位码对应的字节各自加上 20H, 就得到国标码。在进行计算机信息存储时, 为了与 ASCII 码区分, 在每个字节的最高位补上 1, 就得到了汉字的内码。

例如, “汉” 位于第 26 区的 26 位, 其区位码为 0001 1010 0001 1010B (1A1AH)、国标码为 3A3AH, 在国标码汉字文本中, 该字被保存为 BABAH, 占 2 个字节。

随着信息化的迅速发展, 国人很快发现这个标准已经远远不够用了。为了能支持更多的字符集, 1995 年又颁布了 “汉字编码扩展规范 (GBK)”, 在保持与 GB2312–1980 国家标准所对应的内码标准兼容的基础上, 将汉字编码扩展到最多 4 个字节, 并于 2000 年 3 月发布了《信息技术中文编码字符集》的国家标准 GB18030–2000。该系列最新标准是 GB18030–2005, 共收录 7 万多汉字, 此外还包括我国一些主要少数民族的文字。

除中国以外, 各国同期也都制定了符合本国信息编码需要的文字标准。这虽然部分解决了计算机处理文字信息的问题, 但新的问题也出现了: 各国间的文字编码互不兼容 (compatible)。例如, 虽然汉字的中国国家标准中包含了日文字母, 但用这种方式编码的电子邮件发到日本, 对方是完全看不懂的。

Unicode

上世纪 80 年代, Unicode Consortium 和 ISO 两个组织差不多同时开始了国际文字信息标准的制定工作。随后他们意识到, 完全没必要搞两套不同的标准, 这项工作很快合并到一起。目前两套标准是一致的。我们习惯所称的 Unicode 与国际标准化组织 ISO/IEC 10646 标准相同。Unicode 2.0 之后的版本采用了与 ISO 标准相同的字库和编码。

Unicode 仅规定了一个字符和一串二进制序列的对应关系, 并没有规定这个二进制序列在计算机中的存储格式 (内码)。实现存储格式的方案可以有很多种, 目前使用最多的是

UTF-8 (8-bit **U**nicode **T**ransformation **F**ormat)[9], 即将 Unicode 的二进制序列拆分到若干个 8 位的字节中。具体实现方法是:

1. 第一个字节最高位如果是 0, 则是一个单字节符号, 它对应 ASCII 码;
2. 第一个字节最高位如果不是 0, 高位连续 1 的个数表示该符号的字节数, 第一个 0 之后的剩余位用于填充编码数据;
3. 从第二个字节开始, 每个字节以 10 起头, 剩下的 6 位用于填充编码数据;
4. 将 Unicode 二进制序列从最后一个字节开始由低位到高位填充。

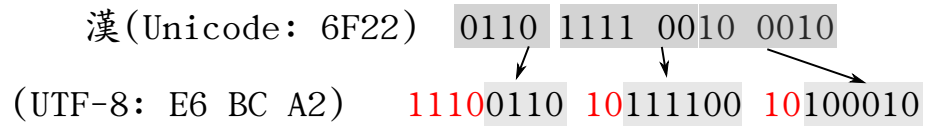
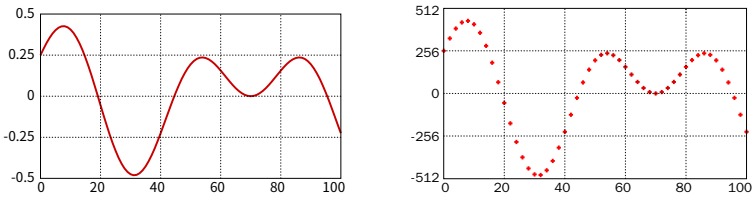


图 1.2: UTF-8 编码示例

这种表示方法有这样一些好处: 一是它完全兼容 ASCII; 第二, 由于是以单字节为单位的, 不存在字节顺序 (endian) 问题, 同样的文字在大端 (big endian) 或小端 (little endian)[10] 计算机都是一样的; 第三是容易纠错: 丢掉一个字节, 最多只会导致一个字符的错误, 不会引起后面大段文字乱码。

基于 Unicode 的其他内码形式还有 UTF-16 和 UTF-32 等方案, 即一个字符用 16 位或 32 位表示。由于减少了编码位, 它们用于存储长编码文字信息时可以占用较少的存储空间。

1.3.3 其它类型的信息



(a) 模拟量 (b) 离散化

图 1.3: 模拟量和数字量

除了数字、文字等信息以外, 计算机还处理大量的声音、图像等信息。这些信息来自我们人类的社会生活。人类所感知的信息大量的都是模拟的 (在时间和幅度上都是连续的), 计算机无法对模拟信息进行直接处理, 必须要通过一种转换装置将它们变成数字 (digital) 信号才能让计算机处理。这种转换器被称作模拟–数字转换器 (**A**nalogue to **D**igital

[9]此编码方案由 Ken Thompson 设计。他也是 B 语言 (C 语言的前身) 的发明人。他还因开发 UNIX 操作系统, 与 Dennis Ritchie 共同获得 1983 年图灵奖。

[10]Endian: 计算机存储多字节数据的顺序。当一个数字由多个字节拼接而成时, 存放在存储器中就有了前后顺序问题。一些计算机把高字节数据 (Most Significant Byte, MSB) 放在低端地址单元, 把低字节数据 (Least Significant Byte, LSB) 放在高端地址单元, 这种数据存放顺序称为 Big Endian, 反之称为 Little Endian。目前采用 Little Endian 的处理器占多数。

Converter, ADC)。转换过程中, 模拟信号 (analogue) 在时间上和幅度上都进行了量化。例如, 人耳听到 1 秒钟的声音, 实际上是耳膜在这段连续时间内对声压连续的变化的感受。通过传声器 (microphone) 采集这个变化的声压, 将其转换成 0 到 1V 之间的电压, 按毫伏的精度、每万分之一秒保存一次这个电压值, 于是就得到了一组 1 万个存储单元、每个单元不超过 10bit 的数字量 (图1.3)。一旦编码形式确定, 这组信息就有了明确的含义, 可以发挥计算机数值运算的优势, 由计算机进行信息处理。处理之后的数据如需还原成模拟信号, 还应通过数字–模拟转换器 (**D**igital to **A**nalogue **C**onverter, DAC)。

为了在不同计算机平台进行数据交换, 人们还为不同的声音信息编码制定了不同的格式标准, 将信息组装在文件中, 由此形成了 WAV、MP3 等不同格式的文件。其他类型的媒体信息, 如视频、图像, 乃至各种需要让计算机进行数值处理的物理量莫不如此。

模拟信号转换成数字信号后, 这些信息的数据量通常都非常巨大。计算机在处理这些信号时, 为了节省存储空间、提高处理速度, 有时还要考虑在传输过程中占用的带宽 (bandwidth), 常常需要对这些数据进行压缩、编码等处理工作。如何有效地对这些信息进行压缩编码/解码, 是目前信息处理中重要的研究内容。解决这类问题, 数字信号处理是其重要的理论基础之一。

1.4 计算机系统构成

计算机由硬件和软件两部分组成。硬件 (hardware) 是指组成计算机的集成电路、印刷板、电源、存储器 (memory) 等硬设备, 包括键盘、显示器、打印机等外围设备 (device)。它们的基础是电子技术, 特别是集成电路技术。而软件 (software) 通常是指计算机程序, 包括系统软件和应用软件, 它们的基础是计算机的指令系统 (instruction set) 。其中系统软件是指操作系统 (operating system) 、数据库管理系统、语言处理程序 (编译、解释程序), 还有一些管理计算机工作的软件; 应用软件是指用户根据具体任务编制的程序。作为一个整体, 计算机的一部分功能由硬件完成, 另外一些功能由软件完成。哪些任务由硬件、哪些任务由软件实现, 根据不同的技术条件以及不同的目的, 可以采用不同的策略。

现代计算机体系结构由美国科学家冯 · 诺伊曼[11]首先提出。1945 年 6 月 30 日, 冯 · 诺伊曼提交了一份划时代的报告《关于 EDVAC 的报告草案》, 系统地介绍了制造电子计算机和程序设计的新思想, 也就是冯 · 诺伊曼体系结构。在这种思想中, 计算机以存储器为中心, 由包括运算器、控制器、输入和输出设备在内的共五部分组成。计算机按存储程序原理 (stored program architecture) 工作, 程序预先保存在存储器中, 由控制器依次从中取出指令并执行。存储程序原理奠定了计算机体系结构的基础, 至今仍极大地影响着计算机体系结构的设计。

从工艺器件的角度来看, 计算机由中央处理单元 (**C**entral **P**rocessing **U**nit, CPU, 又称微处理器 **M**icro **P**rocessor **U**nit)、存储器、输入输出接口 (interface) 及总线 (bus) 构成。中央处理器包含了运算器和控制器部分。输入/输出设备通过相应的接口连接到计算机。各部件之间通过总线传递和交换信息。图 1.4 是计算机系统硬件基本组成框图。

1.4.1 总线和数据传输

总线是连接微型计算机各个部件的公共通道。总线大大简化了器件之间的连接, 同时也为计算机系统单元模块化建立了基础, 使微机系统具有更好的可扩展性和可维护性。

[11] John von Neumann, (1903.12.28 – 1957.2.8), 生于匈牙利布达佩斯, 美籍犹太数学家, 现代电子计算机与博弈论的重要创始人。

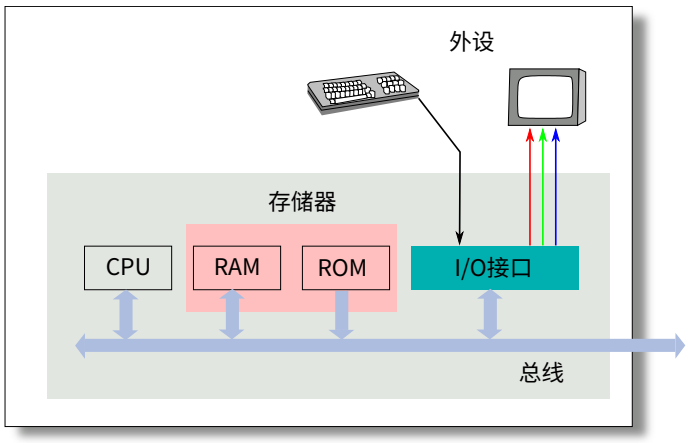


图 1.4: 计算机系统硬件基本组成

根据在总线中传递信息性质的不同, 同一组总线又分为数据总线 (data bus)、地址总线 (address bus) 和控制总线 (control bus) 三部分, 俗称 “三总线结构”:

数据总线: 传输数据的通道。数据总线是双向的, 它允许 CPU 读入数据, 也允许 CPU 通过它写出数据。

地址总线: 明确数据传输的位置 (地址)。站在 CPU 的角度看, 地址总线是单向输出信号。

控制总线: 产生用来保证数据传输的控制信号。控制总线中既有输出信号也有输入信号, 它们决定了总线的特色。

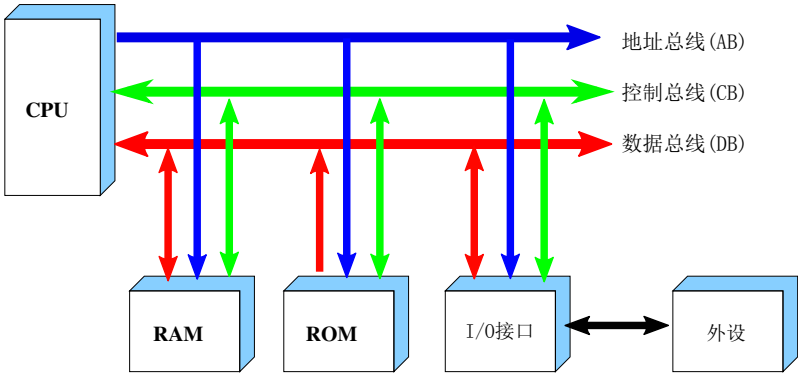
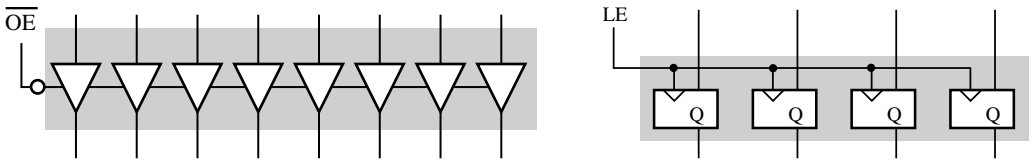


图 1.5: 总线结构组成示意图

根据数据位的宽度, 不同总线形式可分为并行 (parallel) 总线和串行 (serial) 总线两大类。并行总线在同一个周期内可以同时传输多位二进制信息, 而串行总线每个时钟节拍只能传送 1 位信息。图 1.5 是并行总线示意图。在传统的概念中, 并行传输方式下一次可以传输多个二进制位, 传输速度比串行总线高。而串行传输只需要一、两根传输导线, 它的优势是成本低、结构简单。但随着计算机时钟频率的提升, 并行总线已丧失了它的速度优势。原因是在高速数据传输的情况下, 器件、导线的信号延迟已开始影响到数据位与位之间的同步。我们如今看到越来越多的 SATA (**S**erial **AT** **A**ttachment) 硬盘、PCIe (**P**eripheral **C**omponent **I**nterconnect **E**xpress) 接口、USB (**U**niversal **S**erial **B**us) 这样的串行总线接口, 并不仅仅是因为成本问题。

总线的这三组信号在逻辑上是独立的, 但在不同的物理实现时则有可能是交叠的。例如, 典型的 32 位并行总线 PCI 有 32 位地址/数据复用信号 AD0~AD31, 在总线周期的不同阶段起到不同的逻辑信号功能, 这种形式称为分时复用 (**T**ime-**d**ivision **M**ultiplexing TDM)。在串行总线中, 地址信息和数据信息可以是分时复用的, 也可以是通过总线协议实现的。

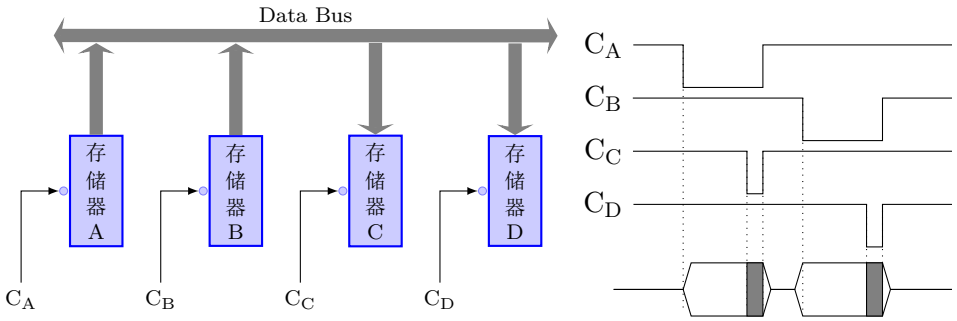


(a) 缓冲器 (b) 锁存器

图 1.6: 缓冲器和锁存器

由于多个器件挂接在同一总线上, 每个器件的输入输出状态如不加适当的限制, 将会造成总线上信息的混乱。为了解决这一问题, 需要在总线的各个器件之间增加可控制的隔离措施。构成总线隔离的器件一般是缓冲器 (buffer) 和锁存器 (latch)。图 1.6 是缓冲器和锁存器的基本结构。

缓冲器由一组三态门 (tri-state) 组成。图 1.7 说明了缓冲器在总线数据传输中的作用。图 1.7 中, 如需将 A 中的数据传到 C, 可将 A 的控制门 C_A 打开, 并将信号传递的方向向外通往总线; 在此期间将 C_C 的门打开并使信号传递方向向内。由于 B、D 器件的控制门关闭, 它们与总线连接的通路处于高阻状态, 只有器件 C 接收数据。最后, 依次将 C_C、C_A 关闭, 完成一次数据传输。



(a) 总线连接 (b) 数据传输

图 1.7: 总线数据传送

锁存器的基本单元是触发器。当有效时钟脉冲作用在锁存允许 LE (Latch Enable) 时, 当前输入端的电平送到输出端并被锁定。在下一个锁存允许有效之前, 输入端的变化不会影响到输出端, 从而将器件两边的逻辑信号隔离。

衡量总线传输性能的一个重要指标是总线带宽, 它表示在单位时间内传输数据的能力。对于并行总线来说, 总线带宽的单位是每秒字节数。例如上面提到的 32 位 PCI 总线, 总线时钟 33MHz, 总线带宽是 132MB/s (32×33M)。而对于串行总线, 总线带宽的单位是每秒比特位, 如 USB3.0 的性能是 5Gb/s。

1.4.2 CPU

CPU 是计算机系统的心脏, 它负责解释指令、处理数据。典型的内部包含算术逻辑单元 (**A**rithmatic **L**ogical **U**nit, ALU)、控制器、寄存器组、指令译码器等部件, 各部件通过内部总线连接。图 1.8 是 CPU 内部逻辑结构图。

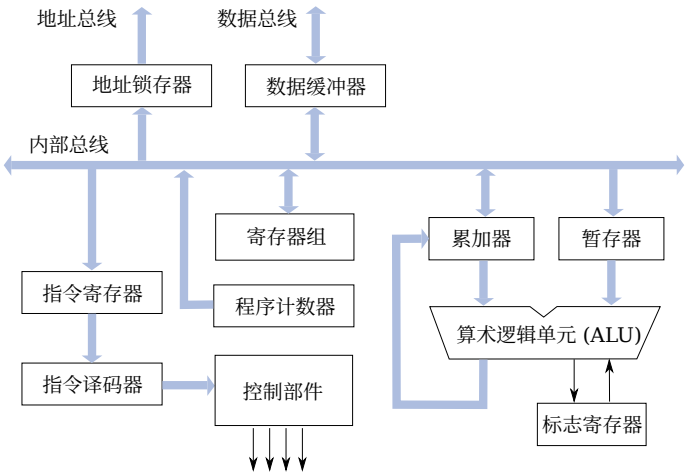


图 1.8: 简化的 CPU 内部逻辑结构

CPU 中一些比较特殊的寄存器有下面几个:

程序计数器 (program counter, PC): 有的地方又叫指令指针 (Instruction Pointer), 用于指向当前指令的地址。程序计数器根据程序运行步骤自动变化, 一般不能由程序直接改变。

堆栈指针 (stack pointer): 堆栈是存储器的一种组织形式, 在访问这块存储器时, 堆栈针指向这段存储器的顶部 (称为 “栈顶”)。向这块存储器添加一个数据时, 栈顶会自动上移; 取走一个数据后, 栈顶也会自动下移。堆栈在子程序调用和返回、中断 (interrupt) 调用和返回起到不可替代的作用。

标志寄存器 (flags register): 用于标记算术逻辑运算结果的特征, 如进位、溢出、正负等信息。标志寄存器作为整体, 一般不参与运算, 特定的标志位会对运算结果产生影响, 程序也常常会根据标志位的情况决定下一步的走向。

累加器 (accumulator): 早期的处理器设计中, 只有个别寄存器才能在算术逻辑单元 ALU 中操作, 这个寄存器被叫作累加器。现代处理器中, 累加器与通用寄存器已没有明显的界限了。

CPU 根据程序计数器 PC 的值到存储单元中取得指令, 随后 PC 的值自动调整指向下一条指令的地址。指令寄存器和指令译码器对获得的指令代码进行分析解释, 通过控制电路产生相应的电信号, 协调 CPU 内部各部分的工作。标志器用来保存算术逻辑运算结果的状态, 以便判断参与算术逻辑运算的操作数之间的关系, 并根据需要对程序进行分支转移。

每执行一条指令, 典型的工作流程可分为取指、译码、执行和回写四个动作。

取指 (fetch): CPU 根据程序计数器产生地址信息。该地址信息生成地址总线信号, CPU 从该地址指向的存储器中读取一条指令, 控制部件将取得的指令送往译码单元。

译码 (decode): 在这一阶段, 指令被拆解为有意义的片断。其中一部分是操作码 (opcode), 表示要进行何种运算, 另一部分是操作数 (operand), 作为参与运算的数值。数值可能来自指令指定的寄存器 (register)、存储器, 或者就在指令本身。操作码送往控制器产生各种控制信号, 操作数送往运算单元, 进入下一个阶段。

执行 (execute): 在这个阶段, 运算单元在控制信号的作用下产生相应的动作, 产生对应的输出。例如, 要完成一个加法运算, 运算器的加法部件输出计算结果, 同时将运算的副产品送往另一个特殊的寄存器—标志寄存器。标志寄存器会记录下这个运算是否产生了溢出、是否产生了进位、运算结果是正数还是负数, 等等。并非所有的指令都需要通过运算单元, 如处理器控制类、分支转移类指令就不会通过运算单元, 存储器读写指令也不会通过运算单元。

回写 (write-back): 有运算结果的指令, 在这一阶段将上一步的结果回写到寄存器或者存储器。一些指令不需要直接运算结果, 只需要设置标志寄存器的状态 (如比较指令), 用于控制将来程序的走向。

每条指令完成后, 程序计数器 PC 会自动调整指针, 以便 CPU 取得下一条指令, 重复上面的过程。当指令顺序执行时, PC 自然增加。Arm32 每条指令 4 个字节 (32 位), 因此调整方式就是加 4。当遇有分支转移、子程序调用、返回、或遇有中断时, PC 会载入新的计数器值。

指令集架构 (**I**nstruction **S**et **A**rchitecture, ISA) 是处理器的工作基础。较为著名的指令集有: X86、Arm、MIPS 等。不同厂商设计的处理器可能使用相同的指令集。个人计算机系统 (Intel 处理器或 AMD 处理器) 采用 X86 指令集, Arm 指令集则见于各种 Arm 架构或 Cortex 架构的处理器。同架构指令集中, 通常高版本兼容低版本; 在不同架构之间则完全没有兼容性。

早期设计的处理器为了节省存储空间, 希望让一条指令做尽可能多的事情, 指令功能设计得相对比较复杂。随着机器性能的不断提高, 寻址方式越来越复杂, 指令也越来越多。然而实际上, 根据二八定律[12], 平常密集使用的指令只有其中的 20%, 而另外的 80% 只有 20% 的场合才会用到。到 1980 年代前后, 出现了精简指令集计算机 (**R**educed **I**nstruction **S**et **C**omputer, RISC) 的概念, 它的指导思想是简化指令功能, 由此带来的好处是指令单元结构简单, 有助于提高可靠性、提高主频、降低成本。与 RISC 相反的设计思想目前被称为复杂指令集计算机 (**C**omplex **I**nstruction **S**et **C**omputer, CISC)。目前没有 RISC 的标准, 它与 CISC 之间也没有明显的界限。通常认为 RISC 具有下面的特点:

1. 指令集数量少, 所有运算都在寄存器内完成;
2. 指令结构整齐、简单, 普遍使用单周期指令, 指令长短一致, 便于流水线操作;
3. 访问存储器只有 2 条指令: 读存储器和写存储器;
4. 有大量的寄存器支持。

上面提到的使用 X86 指令集的就属于复杂指令集计算机。X86 指令集由 Intel 设计, 目前生产 X86 指令集处理器的厂家只有 Intel 和 AMD, 它们主要用于个人计算机。Arm、MIPS 以及近年崭露头角的 RISC-V 则属于精简指令集计算机。Arm 中的字母 “r” 即表示 “RISC”。Arm 指令集由 Arm 公司设计。大量的第三方芯片制造企业通过向 Arm 公司购买 IP 核和指令集授权生产芯片, 由此出现了各种不同厂家、不同型号的 Arm 处理器。Arm 处理器广泛用于移动设备、服务器及各种嵌入式计算机产品中。RISC-V (V 表示第

[12] 又称帕累托法则 (Pareto principle)。此概念由意大利经济学家 Vilfredo Pareto 最早提出。它表述的是, 约仅有 20% 的因素影响 80% 的結果。

5 代) 则是开源架构的精简指令集, 项目来自加州大学伯克利分校, 以 BSD 版权协议[13] 发布。

Arm 指令集历经发展, 出现了若干个的版本, 版本之间略有差别。表 1.4 是 armv5 32 位的指令集简表。

表 1.4: Arm 指令一览

指令类型	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
数值计算	cond				0	0	I	opcode				S	Rn				Rd				operand 2											
乘法运算	cond				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm			
长乘运算	cond				0	0	0	0	1	U	A	S	RdHi				RdLo				Rs				1	0	0	1	Rm			
数据交换	cond				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm			
数据传输	cond				0	1	I	P	U	B	W	L	Rn				Rd				offset											
未定义	cond				0	1	1	------																				1	------			
块传输	cond				1	0	0	P	U	S	W	L	Rn				Register List															
跳转	cond				1	0	1	L	offset																							
(C) 数据传输	cond				1	1	0	P	U	N	W	L	Rn				CRd				cp_num				offset							
(C) 数值运算	cond				1	1	1	0	CP opc				CRn				CRd				cp_num				CP			0	CRm			
(C) 寄存器传输	cond				1	1	1	0	CP opc			L	CRn				Rd				cp_num				CP			1	CRm			
软中断	cond				1	1	1	1	software interrupt number																							

A Accumulate: 乘法累加

B Byte/Word: 字节或字传输

C 协处理器指令

I Immediate (立即数操作数)

L Link (连接位): 跳转指令中用 R14 保存 PC

L Load/Store : 数据传输指令中表示传输方向

N (协处理器) 传输长度

P Pre/Post: 地址增量方式 (先/后)

S Set condition code: 设置条件码

W Write-back: 回写

U Unsigned: 无符号数 (数值运算指令中)

U Up/Down: 基址偏移方向 (数据传输指令中)

图 1.9 以数值运算指令为例说明 Arm 指令格式的结构。Arm 的每条指令都可以带有条件执行码, 根据标志位的状态 (零标志、进位标志、符号位标志、溢出标志等等) 决定是否执行指令。Arm 共有 37 个寄存器, 数值运算指令中可以使用其中的 16 个, 分别标以 R0~R15, 在指令中由 4 位二进制编码指定。其中 R15 承担程序计数器 (PC) 功能。

Arm 的每条指令都包含了条件码, 这使得条件判断指令得以简化, 例如要完成下面的运算:

```
if (R3 == 0)
    R3 = R2 + (R4 >> 3);
```

对应 Arm 的两条指令:

[13] Berkeley Software Distribution, 开源版权协议的一种。

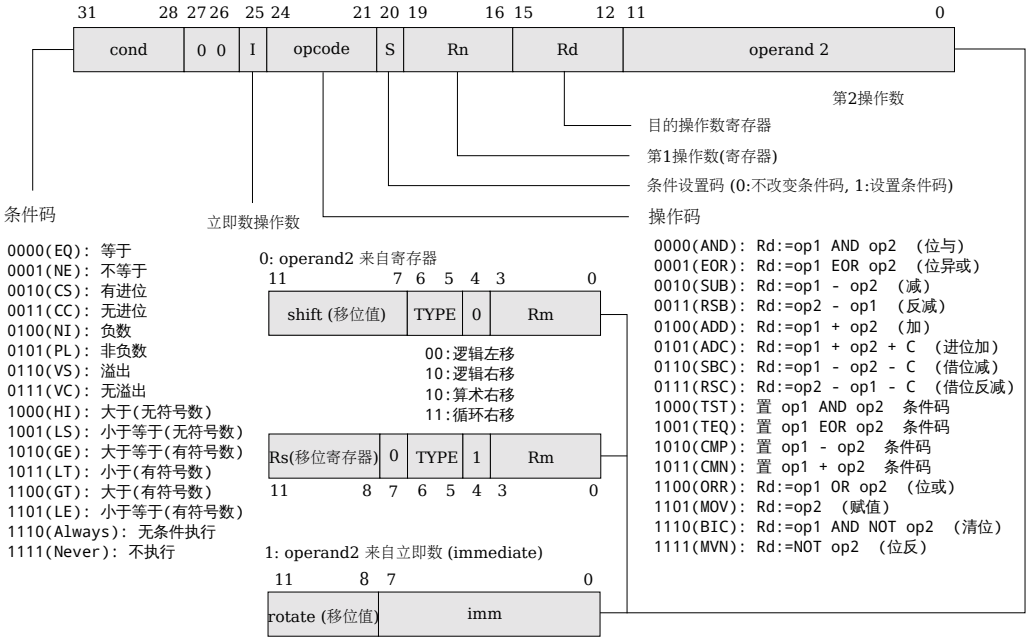


图 1.9: 数据运算指令格式

```
地址     指令码          助记符
1008:   e3530000        CMP     R3, #0
100c:   008331c4        ADDEQ   R3, R2, R4, ASR #3
```

在取指令阶段, CPU 依次从程序存储器中取得 `e3530000` 和 `008331c4` 等等这样的指令码; 指令译码器 (instruction decoder) 的任务就是根据指令集的设计, 将指令码拆解为执行条件、寄存器操作数、运算规则等等的单元, 产生相应的控制信号, 将操作数送往运算单元; 执行阶段, 运算单元在控制信号的作用下完成运算, 最后将运算结果回送到寄存器, 并设置标志寄存器的一些标志位, 程序计数器加 4, 准备获取下一条指令。

1.4.3 流水线

以上各个阶段的处理过程中, 各部件的工作都是相对独立的。为了提高工作效率, 现代处理器会采用流水线 (pipeline) 作业方式: 当第一条指令进入译码阶段后, 取指令单元同时获取下一条指令, 如图 1.10。

指令节拍	1	2	3	4	5	6	7
LD	取指	译码	执行	回写			
LD		取指	译码	执行	回写		
CMP			取指	译码	执行	回写	
ADDEQ				取指	译码	执行	回写

图 1.10: 4 级流水线作业方式

以上每个步骤都在一定的节拍控制下工作。提供给 CPU 的时钟起到的就是节拍作用。并非每一个步骤都是一个节拍, 也并非每一种处理器都严格按照四个步骤设计。将指令执

行过程分解得足够细以后, 每个操作单元可以设计得很简单, 甚至简单到只需要一个时钟周期就可完成。简单的电路结构也有助于提高处理器的工作频率。从宏观上看, 虽然流水线级数很多, 但只要流水线不发生冲突、不被破坏, 每个时钟脉冲下都有一条指令进流水线、一条指令出流水线。这种情况下, 评估处理器运算性能的指标之一 MIPS (**M**illions of **I**nstructions **P**er **S**econd, 每秒执行指令的百万条数) 就可以直接以时钟频率来衡量。[14]

虽然流水线级数越多, 器件就可以做得越简单, 计算机的速度也就越容易提高。但级与级之间的数据传递时间是有下限的, 它受制于半导体工艺水平、并最终受制于光速这一宇宙物理常数。因此当级数增加到一定数值后, 级间延迟的影响作用越来越大, 导致再增加流水线级数也无法提高处理器的速度。

而另一方面, 流水线冲突的情况却是常常发生的。当后一条指令需要用到前一条指令的运算结果、而前一条指令还没完成回写, 由此产生的指令依赖关系是一种冲突形式; 取操作数、回写操作数都要访问同一单元, 由此导致的资源争抢也是一种冲突形式。目前解决冲突的主要方法是延迟和乱序。延迟即是让可能发生冲突的指令错过冲突的时间, 这可能导致流水线效率的降低; 乱序是打乱程序的执行顺序, 让结果有依赖关系的指令稍后执行, 将无关的指令调整到前面。

分支指令也是降低流水线效率的一个重要因素。当分支指令执行时, 在分支指令后跟进流水线的指令需要清除, 并从分支目的地处获取新的指令。这个代价被称为 “分支惩罚” (Branch Penalty)。随着流水线级数的增加, 分支惩罚的代价也变得更高。这种分支可以是显式的子程序调用、跳转, 也可以是不可预见的中断。现代处理器中的分支预测逻辑可以减少分支惩罚的影响。

1.4.4 超标量流水线

流水线技术发展的另一个结果是构造多条流水线, 让处理器能在同一时刻拥有一个以上的硬件单元来处理流水线阶段, 从而实现指令级并行处理。这样的处理器被称为超标量体系结构 (Super Scalar)。Intel Pentium 处理器之后、Arm Cortex-A8、Cortex-A9 架构均属于超标量处理器。

微处理器的发展过程中, 一方面性能越来越高, 集成电路规模越来越大; 而另一方面, 通用 CPU 在组成微型计算机或微处理器应用系统时, 还需要存储器芯片、输入输出接口芯片等一些外围器件。于是商家将 CPU 与这些器件都集成在一个芯片上, 构成单片机 (Micro Controller), 意思是由单个芯片构成的计算机。这类处理器目前广泛应用在嵌入式系统 (embedded systems) 上。

■ 本章练习

1. 十六进制与八进制相比有什么优点?
2. 比较以下不同进制数的大小:

   0100 1001B、45H、49BCD、80、111Q。
3. 将 123.456 转换成二进制和十六进制。
4. 将 1101 0010 1010.0101B 转换成十进制。
5. 证明: 有符号数的加减运算需要考虑溢出，无需考虑进位; 无符号数的加减运算需要考虑进位，无需考虑溢出。
6. 有如下规格的浮点数: 阶符和阶码 6 位，按补码表示，尾符和尾码 10 位，按原码表示。该浮点数规格化表示的最大数和最小数各是多少? 非 0 最小绝对值是多少? 当

[14]在不同架构之间, 使用 MIPS 指标评估处理器性能不尽合理。

允许非规格化时，非 0 最小绝对值是多少？0001000 1011100101B 表示的具体数值是多少？

7. 试编写一个适用于定点处理器的浮点加法程序。要求：除数据输入和输出存储单元以外，不得定义浮点变量和使用浮点运算函数库。
8. 一个 16 位的处理器系统，它的数据总线宽度应该是多少？如果它有 20 位地址线，则该系统能管理的最大内存是多少字节？

第 2 章

存储器与文件系统

存储器是微处理器系统的重要组成部分。根据它们在系统中的位置，通常又分为内存和外存。内存大多由半导体器件构成，其特点是速度快。但由于价格较高，所以容量一般比较小。而外存要求较大的容量，通常有磁盘、光盘等介质，也包括半导体材料的外存储器，它们的单位价格较低，且在掉电后仍能保存信息。本章主要介绍由半导体材料构成的存储器以及内存的组织方式。

随着 *CPU* 性能的不断提高，如何控制大容量、高速存储器系统的成本就成了一个值得研究的问题，*cache* 由此应运而生。

存储在物理介质上的数据以某种方式组织起来，使用文件和目录这样的逻辑概念代替物理存储设备使用的数据块的概念，由此就形成了文件系统。只读存储器的一种重要使用方式是构建文件系统。

2.1 存储器概述

存储器 (memory) 是计算机系统中存储数据和程序的设备。根据存储器与 CPU 之间的连接方式, 可分为内部存储器 (内存) 和外部存储器 (外存)。

2.1.1 内存储器

内存是能被 CPU 直接寻址的部分, 它们通常由半导体器件构成。根据读写特性, 又分为读存储器 (**R**ead-**O**nly **M**emory, ROM) 和随机存储器 (**R**andom **A**ccess **M**emory, RAM)。

只读存储器未必真的只能读出不能写入。现代计算机系统中, 大多数只读存储器都是可以改写的, 只是在写入只读存储器时需要特定的程序步骤, 有时还需要提供特定的电压。目前只读存储器的概念是指在掉电后不会丢失存储信息的这类设备。具备这一特性的存储器又被称为非易失性 (non-volatile) 存储器。任何一个计算机都有 ROM, 它至少要预装一小段程序, 以保证每次计算机开机时执行一段确定的指令。个人计算机系统中, 这段代码被存放在 BIOS (**B**asic **I**nput **O**utput **S**ystem, 基本输入/输出系统, 个人计算机主板上的一小块只读存储器) 中, 它用来对计算机进行初始化并将操作系统加载到 RAM 里。

随机存储器也不真的是将数据随机地存储在设备中。RAM 的组织方式允许处理器对任意一个特定的地址直接进行存取操作。随机存储器的读写方式是相对磁带机而言的。由于早期的只读存储设备使用的是磁带, 磁带上的数据必须按顺序存取。也许, 恰当的名称应该是 “非顺序” 存储设备。目前, 多数只读存储器 (包括 Flash ROM、CD-ROM) 也可以做到直接访问 (即所谓的 “随机存取”), 但 “随机存储器” 已有专指, 不用于这些设备。

随机存储器又称为易失性 (volatile) 存储器, 掉电后, 原有的存储信息不能继续保存。

在随机存储器中, 利用晶体管不同的特性和构造, 又形成了静态随机存储器 SRAM

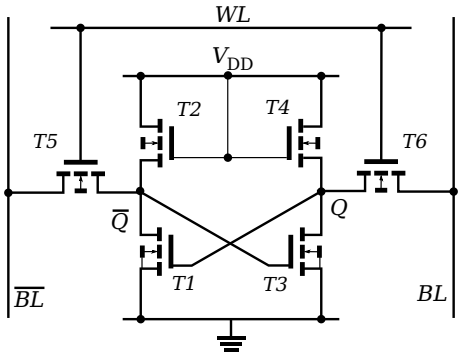


图 2.1: SRAM 基本存储单元

(**S**tatic **RAM**) 和动态随机存储器 DRAM (**D**ynamic **RAM**) 两大类。图2.1 是由 6 个 CMOS (**C**omplementary **M**etal-**O**xide-**S**emiconductor, 互补型金属氧化物半导体) 场效应晶体管构成的 1bit 静态随机存储单元。图中 T1、T2 和 T3、T4 构成两个交叉耦合的反相器, 位信息就保存在这两个反相器中 (Q 或者 $\overline{Q}$)。另两个晶体管 T5 和 T6 用于控制读写开关:

读 设当前存储值为 1 ($Q = 1$)。读周期开始时, 两条位线 (BL) 预置高电平 1。当字线 (WL) 产生高电平时, T5、T6 处于导通状态, 由于 Q 与 BL 逻辑值相同, BL 保持不变; $\overline{Q}$ 与 $\overline{BL}$ 不同, $\overline{BL}$ 经 T1 放电变为逻辑 0。当存储值为 0 时, 仿上述分析过程, 从 BL 获得输出结果。

写 将待写入的状态加到位线 (写入 1 时, $BL = 1$, $\overline{BL} = 0$), 字线加载高电平, 位线状态被存入该单元。

动态存储器利用晶体管结电容效应进行信息存储, 理论上 1bit 信息只需要 1 到 2 个晶体管, 因此集成度更高, 比 SRAM 价格低很多。图 2.2 是 16 个 DRAM 单元构成的 4×4 存储矩阵。由于电容集结的电荷会随时间逐渐放掉, 信息不能长时间保存。解决的办法是利用一个专门的电路定期对它们读一遍, 给带电荷的单元充电, 这一过程被称作刷新 (refreshing), 动态一词由此而来。刷新周期一般在毫秒数量级。

动态存储器的另一个结构特点是行列地址线时分复用, 通过行地址选择信号和列地址选择信号分两次将地址信息送入存储器芯片, 两组地址信号在片内重新组合成完整的地址信息。

2.1.2 外存储器

外存储器主要用于数据存储, 它们一般由只读存储器构成。只读存储器一般只允许被读取而不能被轻易写入或改写其内容。但有时候, 允许修改存储器的内容也是非常有意义的。下面列出了常见的只读存储器:

- 掩膜式只读存储器 (Masked ROM), 最早的只读存储器就是这种形式。该类芯片通过工厂的掩膜制作, 已将信息做在芯片当中, 不能后期更改。适用于大批量生产。
- 可编程只读存储器 (Programmable ROM, PROM)。该类芯片允许用户使用特殊的设备进行一次性写入编程, 但不可再擦除。这种功能可以让用户自己 “生产” 固化的软件而不需要昂贵的生产设备。

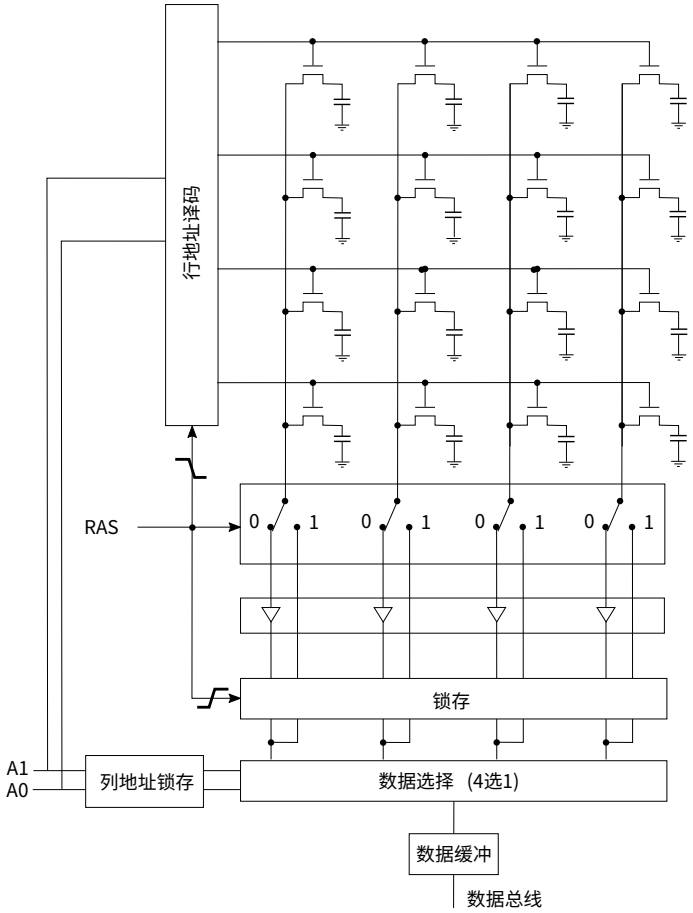


图 2.2: 4×4bit DRAM 存储矩阵

- 可擦除只读存储器 (Erasable PROM, EPROM), 一般指可用紫外光擦除的 PROM。该类芯片允许用户多次编程和擦除。擦除时, 通过向芯片窗口照射特定频率的紫外光的方法来进行。显然, 这种只读存储器比一次性的 PROM 使用更加方便。

- 电擦除只读存储器 (Electrically Erasable PROM, EEPROM), 也写作 E^2PROM。该类芯片可以用软件方法多次擦除和写入。至此, "只读" 的概念在这里已经模糊了。但要注意, 对这种存储器的读和写的控制不是对称的。改写其中的内容, 需要额外的电压控制。

- 闪速存储器 (Flash ROM)。闪速存储器可实现大规模电擦除, 比传统的电擦写只读存储器 E^2PROM 使用更加方便, 可以直接在线擦除和写入。但读和写仍然是不对称的, 擦除和写入时, 需要对芯片中特定的寄存器写入特定的控制命令, 是一个编程的过程。

目前, 闪存的使用已经相当普遍。U 盘、TF 存储卡乃至作为 PC 硬盘使用的固态硬盘 (**S**olid-**s**tate **D**isk, SSD) 均属此类。闪存的主要问题是有擦写次数的限制, 存储单元反复擦写到一定次数以后将会永久失效。

CPU 访问外部存储器时, 需要通过 I/O 总线。外存不限于半导体存储设备。硬盘、光盘乃至更早期的磁带, 这些存储设备均属于外部存储器, 亦称为非易失性存储器。它们的特点是容量大, 单位成本低, 但是读写速度低, 通常用于构成计算机的文件系统 (filesystem)。

2.1.3 存储器的指标

存储器的容量和速度是我们最为关心的指标。存储容量指存储器芯片所能存储的二进制信息的多少。如某芯片有 2048 个存储单元, 每个单元存放 8 位二进制数, 则它的规格就是 2048×8 位, 通常又说成 2K 字节。尽管目前的微处理器普遍已进入 32 位甚至 64 位时代, 但存储器容量仍习惯按 8 位的单位计算, 并且多数计算机系统要求能够独立访问到 8 位字节。从另一方面说, 虽然存储器以字节为单位, 但存储器芯片内的存储单元并不必然是字节, 它可以是 4 位甚至 1 位。因此, 2048×8 位和 4096×4 位是两种不同规格的存储器, 它们在组织计算机存储系统时是不一样的。

存储器的速度用最大存取时间来衡量。存储器的存取时间定义为访问一次存储器所需的时间, 其上限值即为最大存取时间, 它反映了存储器的工作速度。这个指标的上下差别很大, 高速存储器的速度可以和 CPU 的速度处于同一数量级, 慢的可以达到几百个 ns, 当然, 其价格也相差悬殊。

此外, 当考虑一个实际系统的设计方案时, 存储器的可靠性、功耗、集成度以及封装也是必须考虑的。

2.1.4 存储器层次结构

计算机系统中的存储设备性能差异很大。速度越快的存储器, 价格越高。为了解决存储容量和价格的矛盾, 由此形成了计算机系统的存储器层次结构。计算机系统中, 根据存储器与 CPU 之间的亲疏关系, 一般有下面的层次:

寄存器 与 CPU 工作速度一致, 可以实现最快的访问。通常一个 CPU 内部有十几个到几十个寄存器。例如, Arm 处理器中有 37 个 32 位的寄存器。

高速缓存 由静态存储器构成, 与 CPU 工作速度相当。有些系统还会构建多级高速缓存 (cache)。

主存 由动态存储器构成, 速度比静态存储器慢很多, 但单位价格也低很多。计算机系统中通过高速缓存机制, 将主存与高速缓存进行映射, 让 CPU 在访问存储器时, 尽可能多地在静态存储器中操作。

辅助存储器 容量大, 价格低, 主要以外存的形式出现。在计算机存储器管理中作为虚拟地址 (virtual address) 空间与主存交换信息。

2.2 存储器系统

存储器是构成微机系统必不可少的部件, 它用来存放 CPU 所要执行的指令和各种数据。CPU 能直接获取的指令和数据必须在内存储器中。内存越多, 处理器就可以有更多的机会直接从 RAM 中访问指令和数据, 从而减少对硬盘的读写, 而读写硬盘的时间比读写 RAM 的时间要长得多。

2.2.1 存储矩阵

为了便于信息的读写, 将构成存储信息的基本存储单元配置成一定的阵列, 并进行地址编码, 由此形成了存储矩阵。每个具有唯一地址的单元中, 可存储一位或多位二进制数据。所以芯片的存储容量可以理解为芯片的存储单元数与每个单元存储位数的乘积。如某存储器芯片有 13 根地址线和 4 根数据线, 它的存储单元数为 2^{13}, 即 8K, 而每个单元中的

存储位数为 4, 所以它的容量表示为 8K×4bits。存储器对外的地址线数量和数据线宽度由芯片本身的结构决定。

2.2.2 存储器基本结构

半导体随机存储器一般由地址译码、存储矩阵、控制逻辑和输入输出控制电路等部分组成。

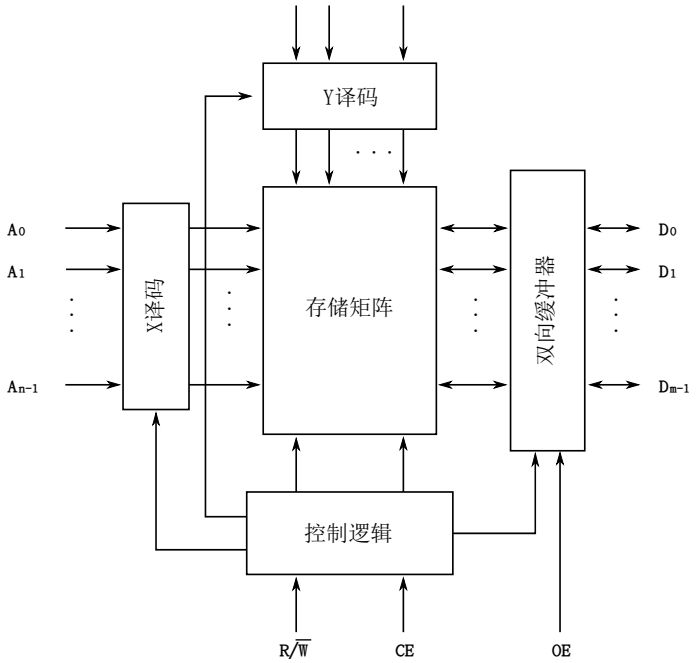


图 2.3: 随机存取存储器的结构框图

地址译码电路

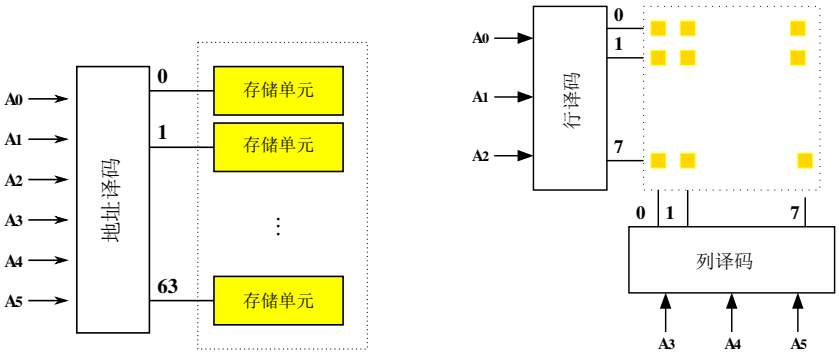
该部分接收 CPU 发出的地址信号, 产生地址译码, 以便选中存储矩阵中的某个存储单元。存储矩阵中基本存储电路的编址方式有单译码和双译码两种 (图2.4)。单译码方式较适用于小容量的存储器, 双译码方式比较适用于大容量的存储器。

读写控制逻辑

存储芯片的片选 (**C**hip **S**elector, CS) 有效时, 可对存储芯片进行读写操作; 无效时, 芯片数据线为高阻态, 与总线脱离。读写信号用于控制总线数据与存储器内部单元之间的数据流向。

输入/输出控制电路

半导体存储器的数据输入/输出控制电路多为三态双向缓冲器结构, 以便使系统中各存储器芯片的数据输入/输出端能方便地挂接到系统数据总线上。当对存储器芯片进行写入操作时, 片选信号及写信号有效, 数据从系统总线经三态双向缓冲器传送至存储器中相应的基本存储单元; 当存储器芯片进行读出操作时, 片选信号、输出有效及写开放信号无



(a) 单译码结构 (b) 双译码结构

图 2.4: 译码电路

效 (读/写控制信号为读态), 数据从存储矩阵中相应基本存储单元中读出, 经三态双向缓冲器传送至系统总线。

2.2.3 存储器扩展

计算机系统中的内存往往需要多块存储芯片进行扩展。存储器的位宽和容量在不满足需要的情况下, 要在字向或位向两个方面进行扩展。存储器扩展时需考虑地址线、数据线和控制线的连接问题。

位扩展

如要求设计组织 1024×8 位存储器。现有二片 1024×4 位存储器, 将一片中的四个数据位定义为 D0~D3, 另一片中的四个数据位定义成 D4~D7, 两片共同构成一个字节。

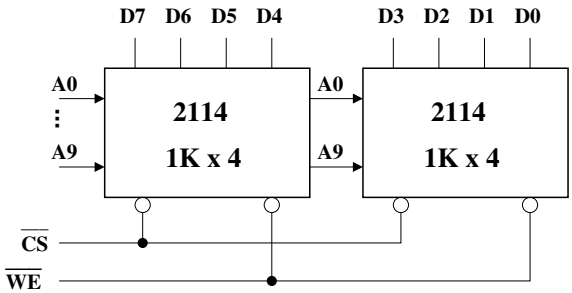


图 2.5: 位扩展法组成 8 位 RAM

芯片的每个单元的位数与存储器符合要求, 但容量不够, 需要在字的方向上扩充。图 2.6 给出用 16K×8 位芯片组成 64K×8 位的存储器框图。图中所有存储器的数据线并联, 并与数据总线相连。地址总线直接连接到各个芯片上, 片选/使能端则由地址译码器的输出单独控制。

字扩展

常常是位和容量都不能满足需要, 此时要在两个方向都进行扩展。

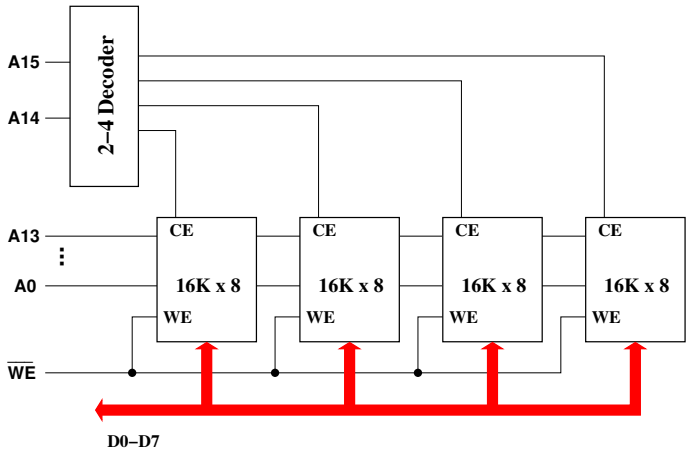


图 2.6: 字扩展法组成 64KB RAM

2.2.4 微处理器与存储器系统相联

由于静态存储器和动态存储器的不同特点, 他们与微处理器连接的方式也有所不同。静态随机存储器的连接方式比较简单: 低位地址线按要求和系统的地址总线连接, 读/写信号也直接来自控制总线, 片选信号 $\overline{\text{CS}}$ 由高位若干地址线通过译码产生。动态随机存储器除了和静态存储器相同的地址线、数据线、读写控制外, 还有行、列地址选择线 $\overline{\text{RAS}}$、$\overline{\text{CAS}}$。由于动态随机存储器的集成度远高于静态存储器, 为了缩小芯片的封装, 其地址线引脚输入分两次进行, 这是 DRAM 一个重要特性, 被称为地址重用。图2.7 对比了两款不同的存储器引脚特征。

动态存储器的存储单元被有规则地排成阵列。为了读到某个位置上的内容, 控制电路首先计算行号, 把这个行号放在 DRAM 的地址引脚上, 然后使行地址选择 ($\overline{\text{RAS}}$) 生效, 让 DRAM 读入行地址, DRAM 内部连通选对应行上的所有单元; 然后控制电路把需要读取的地址的列号送到 DRAM 地址线上, 再使列地址选择 ($\overline{\text{CAS}}$) 生效。DRAM 内部利用这个信号选中指定的输出单元。经过一个 $\overline{\text{CAS}}$ 访问时间的延迟, 这个单元的状态就会出现在 DRAM 的数据引脚上。

向 DRAM 写入数据的过程和读取数据类似, 也要经过两次寻址步骤。

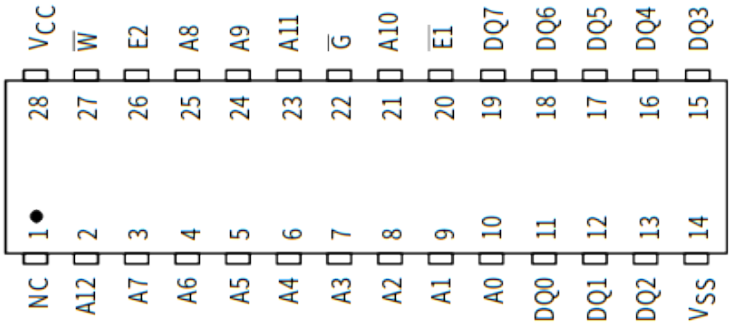
由于动态随机存储器的存储信息容易丢失, 在与 CPU 连接中, 还需要为其提供刷新电路。

大量的现代处理器具有 32 位或更宽的数据总线, 但仍然要求能够独立访问 8 位的存储单元, 于是地址总线中的最低位 (32 位系统的 A0 和 A1, 64 位系统中还包括 A2) 被单独分解成字节允许信号 BE (Byte Enable), 用来实现对字节访问的控制 (图2.8)。

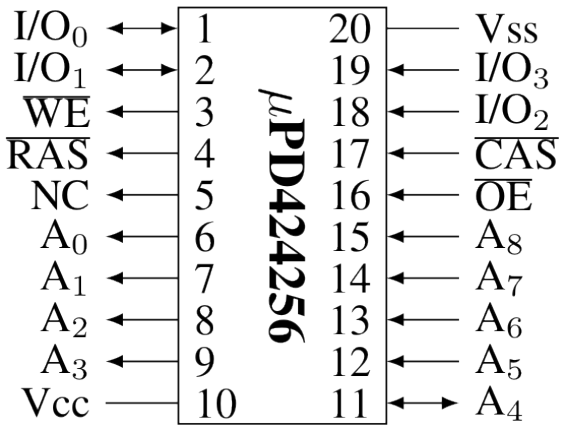
2.3 高速缓冲存储器

构成系统主存的大容量存储器是动态存储器, 速度比 CPU 慢很多, 而与 CPU 速度匹配的静态存储器价格却很高, 由此导致了性能和成本的矛盾。而计算机从内存中取指和取数是最主要的操作, 慢速的存储器严重限制了 CPU 性能的发挥, 影响了计算的运行速度并限制了计算机性能的进一步发展和提高; 另一方面, 由于速度与处理器同一数量级的高速存储器件的价格又十分昂贵, 不可能大规模的使用。

计算机系统采用高速缓存技术解决这一矛盾。其基本思想是, 在 CPU 和主存之间增加一层少量的静态存储器, 此静态存储器即为高速缓冲存储器 (cache)。使用高速缓存系



(a) 静态存储器 (8K×8bit)



(b) 动态存储器 (256K×4bit)

图 2.7: 静态存储器和动态存储器芯片引脚

统, CPU 可以减少访问主存储器的次数, 减少处理器的等待时间, 它对提高整个处理器的性能起到非常重要的作用。

对大量的典型程序的运行情况分析结果表明, 在一个较短的时间内, 由程序使用的地址往往集中在存储器逻辑地址空间的很小范围内。多数情况下, 指令是顺序执行的, 因此指令地址的分布是连续的, 再加上循环程序段和子程序段要重复执行多次, 因此对这些地址的访问就自然具有时间上集中分布的倾向, 这是时间的区域性 (temporal locality)。数据这种集中倾向不如指令明显, 但对数组的存储和访问, 以及工作单元的选择都可以使存储器地址相对集中。这种对局部范围的存储器地址访问频繁、而对此范围以外的地址则访问甚少的现象, 是空间的区域性 (spatial locality)。

根据程序访问的区域性原理 (locality of reference), 在主存和 CPU 之间设置 cache, 把正在执行的指令地址附近的一部分指令或数据从主存装入 cache 中, 供 CPU 在一段时间内频繁使用, 是完全可行的。图 2.9 是其中的一种实现方式。

2.3.1 cache 的工作原理

cache 位于主存和 CPU 之间。管理这两级存储器的部件为 cache 控制器。CPU 和主存之间的数据传输都必须经过 cache 控制器。cache 控制器将来自 CPU 的数据读写请求转向 cache 存储器。控制器先在自己的存储空间内查找是否有对应地址的映射、该数据是否有效。如果有效, 则直接使用该存储空间与 CPU 交换数据, 称为一次命中 (hit)。由于 cache 由静态存储器构成, 这个速度比访问主存 (动态存储器) 要快得多。如果数据无

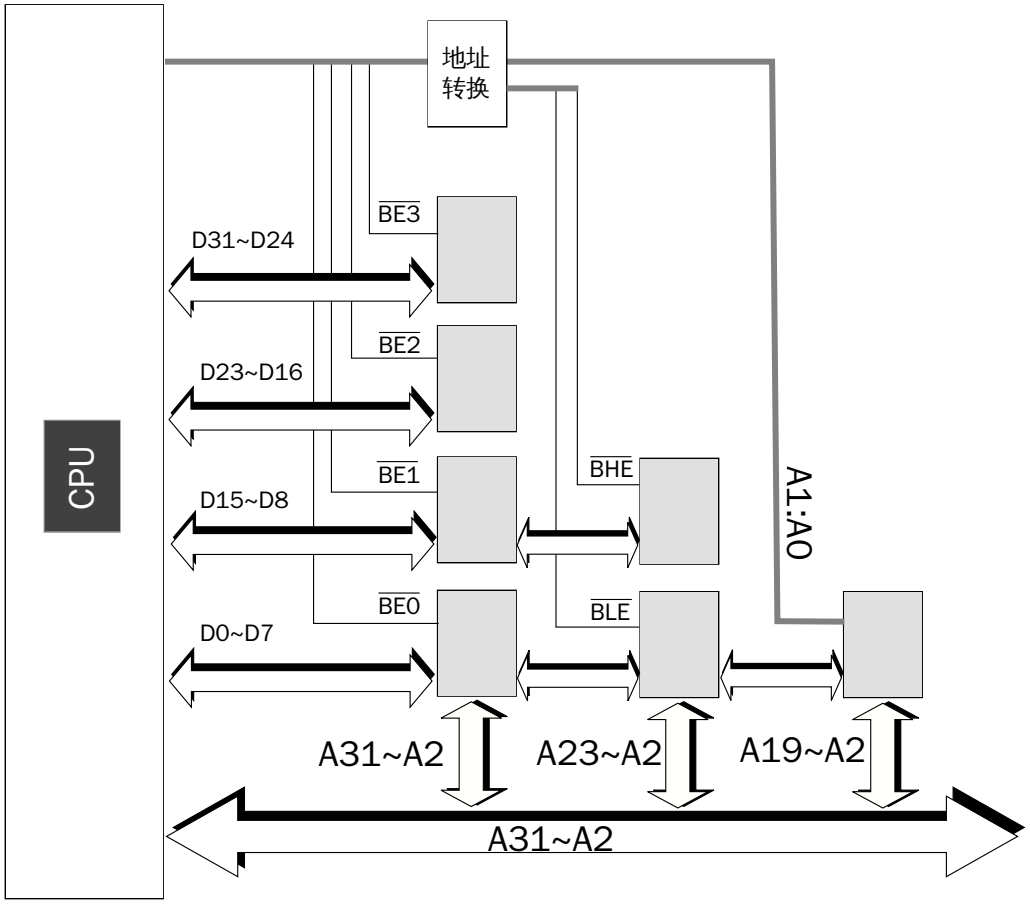


图 2.8: 32 位系统的存储器组织

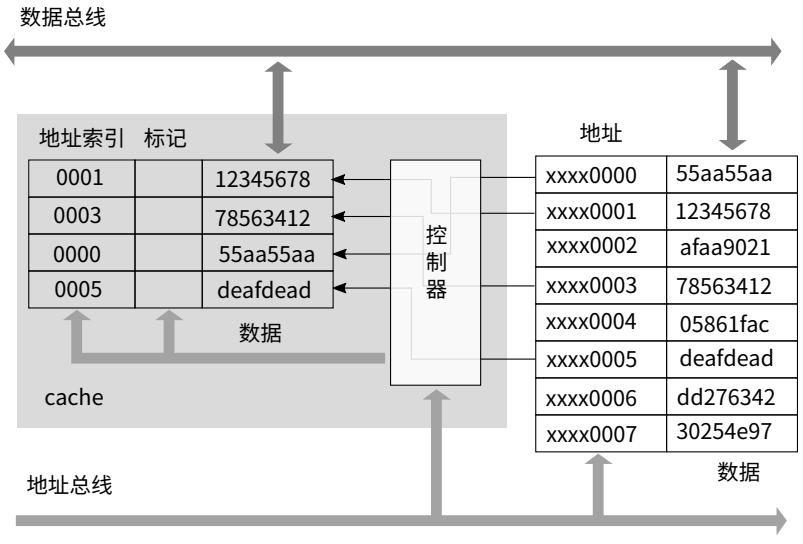


图 2.9: 高速缓存

效 (hit miss), CPU 必须在其机器周期中插入等待周期, 由 cache 控制器从主存中复制到 cache 存储单元, 并标记该单元对应的主存地址以及数据有效情况, 这一过程比较费时, 基本等效于对主存的访问。但在随后对该地址访问时就可以直接通过静态存储器了。大容量

的 cache 存储器与适当的调度策略相配合, 可以使 CPU 访问 cache 的命中率高达 90% 至 98%。只要程序运行遵循区域性原理, 一个设计合理的高速缓存系统可以用极少的成本代价换来系统性能的极大提高。从 CPU 的角度看, 这种 cache–主存存储体系的速度接近于 cache, 容量及存储单元价格则接近于主存, 从而缓解了速度与成本之间的矛盾。

cache 命中率 (hit rate) 是指命中 cache 的次数与访问主存次数的百分比。cache 的命中率受 cache 规模大小、cache 体系结构、所采用的算法以及运行的程序等诸多因素影响和制约。

Intel 公司于 1989 年推出的 80486 CPU 中集成了 8KB 的数据和指令共用的 cache, 1993 年 3 月推出的 Pentium CPU 中集成了 8KB 的数据 cache 和 8KB 的指令 cache, 与主机板上的静态存储器形成两级 cache 结构。多级 cache 系统中, CPU 首先会在微处理器内的 cache 查找数据, 如果不命中, 则在主机板上的二级 cache 中查找, 若数据在第二级 cache 中, cache 控制器在传输数据的同时, 修改微处理器内的 cache; 如果数据既不在微处理器内的 cache 也不在第二级 cache 中, cache 控制器则从主存中获取数据, 同时将数据提供给 CPU 并修改两级 cache。

2.3.2 cache 的组织方式

在主存–cache 存储体系中, 所有的程序和数据都在主存中, cache 存储器只是存放主存中的一部分程序块和数据块的副本, 这是一种以块为单位的存储方式。cache 和主存被分成块, 每块由多个字节组成。由上述程序的区域性原理可知, cache 中的程序块和数据块会使 CPU 要访问的内容, 在大多数情况下已经在 cache 存储器中, CPU 的读写操作主要在 CPU 和 cache 之间进行。

当 CPU 访问存储器时, 送出访问主存单元的地址, 由地址总线传送到 cache 控制器中的主存地址寄存器 MAR (**M**emory **A**ddress **R**egister), 主存–cache 地址转换机构从 MAR 获取地址并判断该单元内容是否已在 cache 中存有副本。如果副本已经在 cache 中, 则立即把访问地址变换成它在 cache 中的地址, 然后访问 cache 存储器; 如果 CPU 访问的内容不在 cache 中, 则转去直接访问主存, 并将包含该存储单元的一块信息 (包括该块数据的地址信息) 装入 cache; 若 cache 存储器已被装满, 则需在替换控制部件的控制下, 根据某种替换算法, 用此块信息替换掉 cache 中原来的某块信息。

用某种策略把主存地址对应到 cache 的存储单元地址, 称作地址映射 (mapping)。程序将主存地址变换为 cache 地址的过程叫做地址变换。布局规则决定着从主存储器映射到 cache 存储单元的操作。目前在计算机系统中, 主要的布局形式有全相联 (full associative)、直接映像 (direct mapping) 和组映像 (set associative) 等几种方式。

全相联

这种方式下, 主存中的每一个字块可映射到 cache 任何一个字块位置上。全相联的 cache 系统在任何时候决定将哪些存储块存放到 cache 时, 都能够最大限度地提供灵活性和适应性。这种方式可达到很高的 cache 命中率, 但实现较复杂。由于 cache 内 2^n 个数据块的全部地址之间不允许有单独联系现象存在, 所以 cache 内存放的必须是一个数据块的完整地址。当处理器需用数据时, cache 控制器就拿所需数据地址与 cache 内的 2^n 个地址逐个进行比较, 查找符合条件的数据。

全相联方法只有在 cache 中的块全部装满后才会出现块冲突, 这种方式的 cache 利用率最高。但全相联 cache 中, 块表查找的速度慢。由于 cache 速度要求高, 因此全部比较和替换策略都要用硬件实现, 这就带来了由于控制复杂、硬件实现起来比较困难的问题。而且地址表需要保存完整的地址信息, 成本也较高。

直接映像

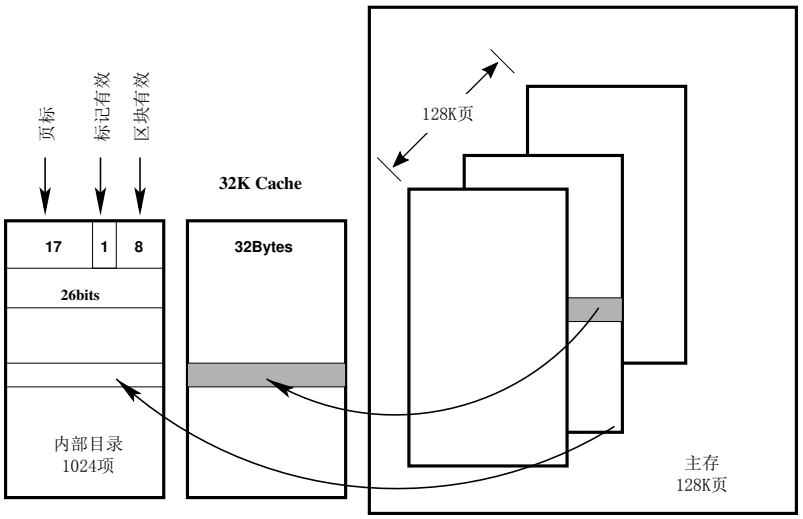


图 2.10: 直接映像方式 cache 系统

每个主存地址映像到 cache 中的一个指定地址的方式称为直接映像 (图2.10)。在直接映像方式下, 主存中存储单元的数据只可调入 cache 中的指定位置, 如果主存中另一个存储单元的数据也要调入该位置, 就会发生冲突。地址映像的方法一般是将主存块地址对 cache 的块号取模即可得到 cache 中的块地址, 这相当于将主存的空间按 cache 的尺寸分区, 每区内相同的块号映像到 cache 中相同的块位置。

直接映像的 cache 布局规则比较简单, 只需用一次块比较即可确定所需的字是否在 cache 中, 地址变换速度快, 不需要考虑替换策略问题。但使用直接映像规则时, cache 中的冲突概率较高。如果 CPU 频繁交替地在两个数据间访问, 而这两个数据又恰好映射到相同的 cache 块上, 就需要反复把这个两个地址的数据不断地写入与调出。这时, 即使 cache 中有其它空闲块, 也无法利用。

组相联映像

组相联映像方式是介于全相联映像和直接映像的一种中间方案 (图2.11)。组相联的 cache 配备两个或多个 cache 体, 它等价于若干个直接映像的 cache 以并行方式操作。它将存储空间分成若干组, 各组之间是直接映像, 而组内各块之间则是全相联映像。在组相联映像方式下, 主存中存储块的数据可调入 cache 中一个指定组内的任意块中。

全相联和直接映像是组映像的两个极端: 如果只有一组, 就变成了直接映像方式; 如果组的大小为整个 cache 的尺寸时就变成了全相联映像方式。组相联方法在判断块命中以及替换算法上都要比全相联方法简单, 拥有相同 cache 运行起来比直接映像 cache 更加有效, 其命中率也介于直接映像和全相联方法之间。

2.3.3 cache–主存的一致性

上面的讨论主要针对的是 CPU 对内存的读操作行为。当 CPU 向内存写数据时, 由于主存跟不上 CPU 的速度, 数据会先写入高速缓存, 由此造成了二者数据的不一致, 必须通过某种方法更新主存的信息。通常用通写法 (Write-through) 或回写法 (Write-back) 更新主存。前者的做法是, CPU 每次写入数据, cache 控制器都要把这个数据写入主存; 后者的

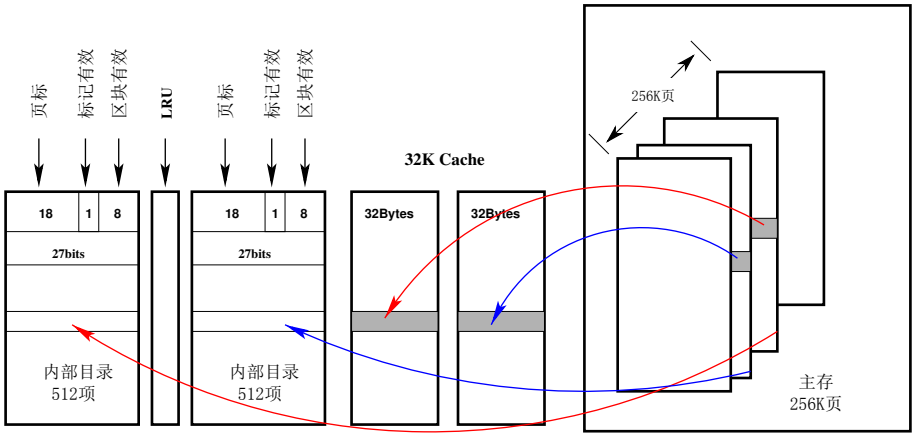


图 2.11: 两路组相联方式 cache 系统

做法是, cache 控制器仅在需要进行 cache 替换时才将数据写入主存。显然, 后者可以有效地减少总线访问。

还有一种造成 cache 和主存数据不一致的情况: 当系统中存在多个 CPU 时、或者存在其他可以访问存储器的设备时 (如 DMA), 主存中数据的变化并不能马上反映到每个 CPU 的局部高速缓存中。一般是通过清除有效标记位的方法解决这个问题: 当 cache 控制器监测到主存的某个地址被写入时, 将本块对应地址上的数据标记为无效, 下次 CPU 访问时就不会将错误的数据交给 CPU 了。

2.4 虚拟存储

“有再多的存储空间, 程序也会想尽办法将其占满”, 这是 IT 界的帕金森[1]定律。程序员通常希望系统能提供无限量的存储器, 而实际物理存储器总是有限的。存储器管理的任务就是要利用有限的物理地址空间满足各任务内存的需求。其中的一种做法是, 将需要使用的地址调入物理内存, 而将暂时不用的数据或程序保存到外存 (磁盘) 上, 这种技术被称作虚拟存储技术。根据对虚拟存储器的管理方式不同, 目前主要有分段 (segmentation) 和分页 (paging) 两种模型。[2]

2.4.1 分段存储管理

处理器执行程序时, 需要从存储器中读出指令、读取操作数、或者向存储器中写入运算结果。例如下面的两条 Arm 指令完成将寄存器 R3 存入内存 [8200] 单元的工作:

```
103ec:    e3a02a02    MOV     R2, #8192
103f0:    e5c23008    STRB    R3, [R2, #8]
```

在虚拟存储系统中, 所有对存储器的访问都是虚拟地址 (以上的指令地址 103ec、103f0 及数据地址 8200)。操作系统必须通过一系列的转换将它们转换到物理地址, 最终体现在地址总线信号上。

[1] Cryil Northcote Parkinson (1909.7.30–1993.4.9), 英国作家、历史学家。原文描述的是 “工作量会充满可用的工作时间” 这一现象。1958 年, 帕金森将这一现象扩充为一本专著, 形成帕金森定律。

[2] 也有的地方将两种模型结合而形成的段页式管理作为第三种模型。

分段管理的基本形式是, 根据程序的逻辑结构, 将不同的数据存放在不同的段中, 例如数据段、代码段、堆栈段等等。段的长度根据需要设定。为了说明逻辑段的不同属性, 系统在内存中为每一个段建立一个段属性表, 记录段的起点、长度、权限等信息。CPU 通过访问段表来实现逻辑地址到物理地址的转换。图 2.12 描述了一种可能的方案。

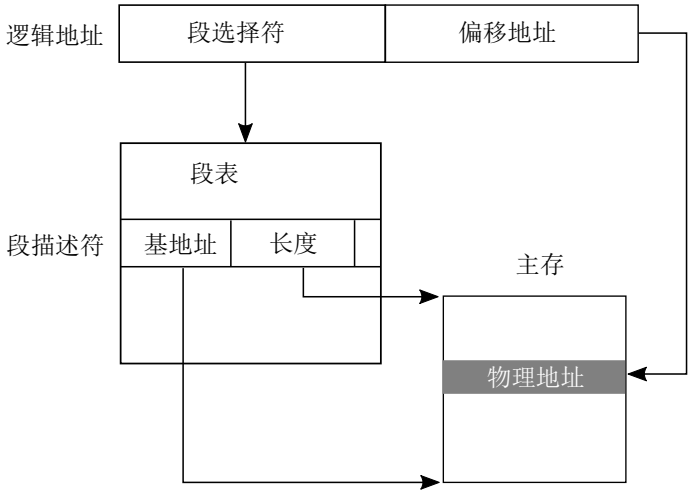


图 2.12: 分段管理方式的地址转换

程序访问的逻辑地址由段选择符 (segment selector) 和偏移地址两部分构成。段选择符用于选择段描述符。一组段描述符 (segment descriptors) 就构成了一个段表。一个段描述符描述了一个段的属性, 包括段的起始地址、长度、访问权限等等。在进行地址转换时, 操作系统将段的起始地址加上偏移地址, 如果这个地址不超出段的界限、且符合访问权限, 则这个地址就是逻辑地址对应的物理地址。

分段管理方式中, 段具有逻辑上的独立性, 方便程序的编译和设置保护权限, 也容易实现数据共享。但由于每个段的长短不一, 主存空间分配比较麻烦。

2.4.2 分页存储管理

分页存储管理中, 将内存 (虚拟内存与物理内存) 以 2 的整数次幂为单位, 划分成大小相等的块, 称为 “页” 或 “页面” (page frame)。在 32 位系统中, 一个页面的大小通常是 4KB, 每个页面对应一个页面号 PFN (**P**age **F**rame **N**umber)。一个虚拟地址由页面号和偏移地址组成, 例如上面的存储器地址 8200 (十六进制 0000 2008), 可以分解为页面号 2 (32 位地址中的高 20 位) 和偏移地址 8 (32 位地址中的低 12 位)。图 2.13 是采用页式存储器管理的抽象模型。

下一步, 处理器以虚拟页面号 (VPFN) 为索引, 在页表 (pagetable) 中检索该页所对应的物理页面。页表由操作系统内核在物理内存中建立。如果该页面未在页表中标记, 意味着这是一个无效的地址 (无物理单元与之对应), 处理器不能对该地址进行访问。此时处理器会触发一个缺页异常, 交由操作系统内核处理。由此, 内核获得了失效地址和页面访问出错的原因。

如果该地址不允许访问, 内核将终止该进程, 以保护系统中的其他进程不受影响。当你在运行程序时看到段错误 (或者 Segment Fault) 提示时, 就是发生了这样的事情。

如果该地址允许访问, 只是不在物理内存, 内核将在物理内存中查找一个空闲的页面, 将该页面从磁盘映像中读入内存, 映射到请求的虚拟页面, 并在页表中标记物理页面号 (PFN), 同时标记该页面有效, 最后返回处理器异常断点, 处理器得到了有效的页面, 程序

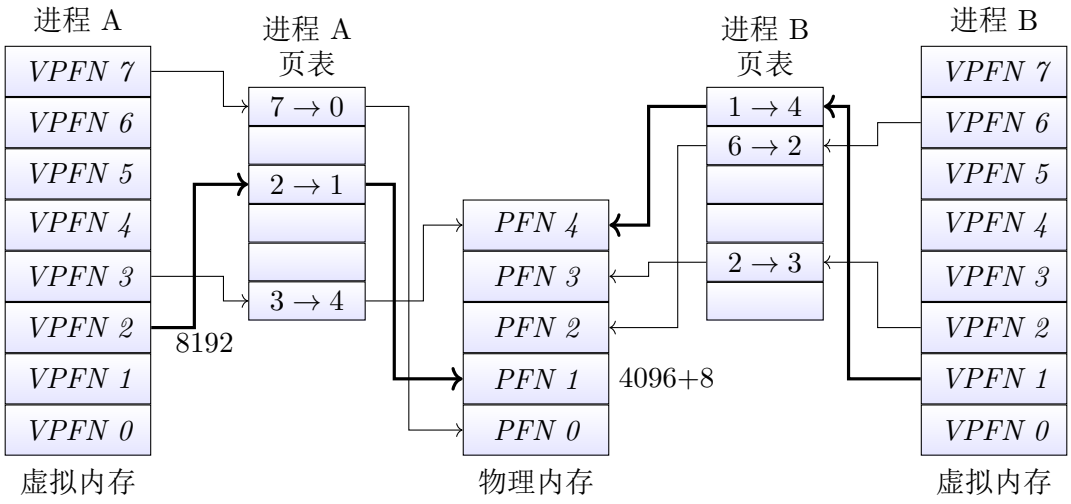


图 2.13: 虚拟存储管理的抽象模型

得以继续运行。

表 2.1: 32 位系统的页表入口格式

31–12	11–9	8	7	6	5	4	3	2	1	0
Page Physical Base Address	avl	G	PAT	D	A	PCD	PWT	U/S	R/W	P

表 2.1 是 32 位系统中一种典型的页表入口格式。除高 20 位作为物理地址索引外, 其余各位含义如下:

- P (**p**resent): 表示该页是否存在于物理内存。如果不存在, 访问该页将触发一次缺页异常。
- W/R: 读写标记, 表示该页是只读的还是允许写入的。
- U/S: 标记是用户页还是系统页 (**u**ser/**s**upervisor)。如果是系统页, 则该页只允许内核访问。
- A (**a**ccessed): 表示该页被访问过 (读或写)。
- D (**d**irty): 脏页表示该页已经被写过。未被写过的页面, 在交换时可以直接丢弃, 下次使用时重新从磁盘调取即可; 被写过的页, 在交换时不能简单地丢弃, 必须要保存到磁盘上, 等待下一次换入时才能有正确的数据。初始化时 A 和 D 位被软件清零, 处理器访问时将其置位。
- PCD (**P**age **c**ache **D**isable): 禁止缓存标记。当页表所关联的地址不是存储器而是 I/O 设备时, 必须禁止缓存, 否则读写该内存不能反映外设的真实情况。具有独立 I/O 地址空间的情况不考虑这一位。
- PWT (**P**age **W**rite-**T**hrough): 用于控制页面缓存方式: 通写 (Write-through) 或回写 (Write-back) 方式。

- PAT (**P**age **A**ttribute **T**able): 页属性。在一些处理器中, 用来表示页单位大小 (4KB 或 4MB)。
- G (**G**lobal): 用于标记此页是否为全局页表。
- avl (available): 供系统使用的一些标记位。

在图 2.13 中, 进程 A 的虚拟地址 8200 (0x2008) 通过查表, 得到 1 号物理页面。1 号物理页面的基地址是 4096, 在此基础上把虚拟地址低 12 位作为偏移量加上去, 即对应到物理地址 4104 (0x1008)。

2.4.3 页表

虚拟存储管理方式中, 页表负责将虚拟地址转换到物理地址。在具体实现时还需要考虑到, 由于使用地址高 20 位作为页表索引, 页表可能会非常大 ($2^{20} \times 4$ 字节, 每个页表项对应一个 32 位地址, 即 4 个字节)。

多级页表

为解决页表占用大量物理内存的问题, 操作系统引入了级页表 (multi level pagetable)。多级页表的基本思想是, 虽然进程的虚拟地址空间很大, 但并不会用到所有的虚拟地址, 因此也就没必要把所有的页表项都保留在内存中。

Intel 处理器的分页管理单元中, 32 位的虚拟地址被分成 3 个部分: 页目录 (10bit)、表目录 (10bit) 和偏移地址 (12bit)。最低 12 位的偏移地址恰好对应 4KB 页面的大小。页管理单元对虚拟地址的这种分解方法实际上就是多级页表的思想。

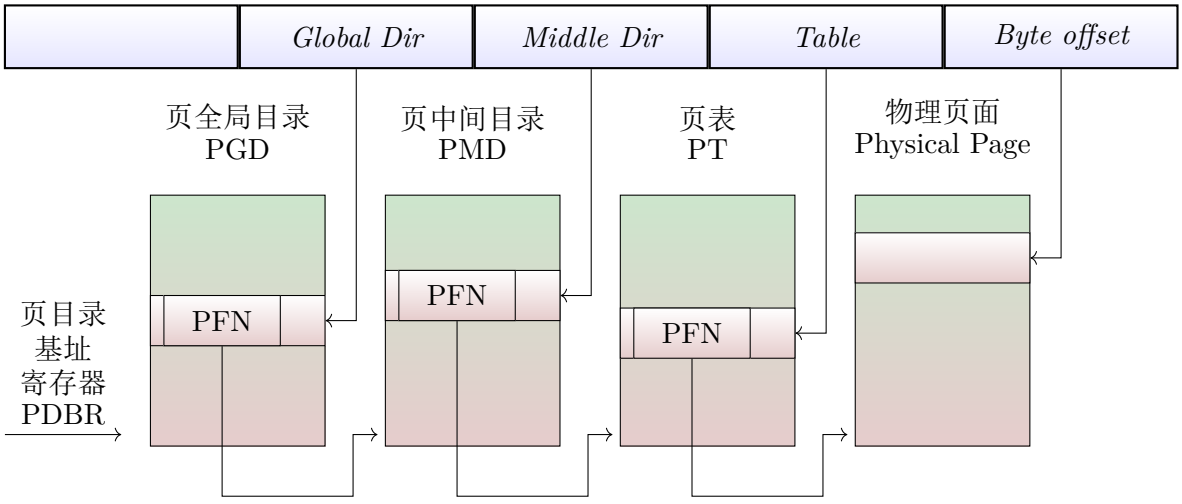


图 2.14: 多级页表

图 2.14 是多级页表查表示意图。Linux 内核设计了三级页表结构, 在这一结构中, 虚拟地址被分为全局页目录、中间页目录、页表目录和偏移地址四段。系统的页目录基址寄存器指向全局页表 (一级页表) 首地址, PGD 作为全局页表项索引, 查到页中间目录 (二级页表) 首地址, 再以中间页目录项作为索引在表中查到页表 (三级页表) 基地址, 通过页表定位到物理页面, 加上偏移地址, 最终得到物理地址。

传统的 Arm 32 位系统采用两级页表方式, 这种情况下, 只要将其中一级折叠起来就可以实现。如下是内核 4.19 版本 Arm 两级页表模式定义的常数, 中间级页表只有一项, 实际上就等效于两级页表: 一级页表 2048 项, 二级页表 512 项。

```
/* arch/arm/include/asm/pgtable-2level.h */
#define __PAGETABLE_PMD_FOLDED
...
#define PTRS_PER_PTE            512
#define PTRS_PER_PMD            1
#define PTRS_PER_PGD            2048
```

为了满足大容量存储空间的需求, Linux 加入了 LPAE (**L**arge **P**hysical **A**ddress **E**xtensions, 大容量物理地址扩展) 特性支持。LPAE 使用三级页表, 每级 512 个页表项 (每项 64bit 地址位)。限于 4GB 的地址范围, 全局页表目前只用了 4 项。如下是三级页表模式定义的常数:

```
/* arch/arm/include/asm/pgtable-3level.h */
...
#define PTRS_PER_PTE            512
#define PTRS_PER_PMD            512
#define PTRS_PER_PGD            4
```

转换旁视缓冲

虚拟存储的另一个问题是速度。在直接存储器管理方式下, 读取一条指令只需要访问一次存储器, 而通过页表查找指令再将指令读入 CPU 至少需要访问三次存储器。一个经典的解决方法是采用转换旁视缓冲 (**T**ranslation **L**ook-aside **B**uffer, TLB) 技术, 它利用程序的空间区域性原理: 程序在大量的时间里只集中在一小块区域操作 (运行程序或读写数据)。一些常用的页表入口被缓存在 TLB 中, TLB 通常是 CPU 存储器管理单元的一部分。当发出虚拟地址请求时, 处理器先尝试从 TLB 找到相匹配的入口, 如果找到, 则直接将虚拟地址转换成物理地址并对数据进行处理, 如果没有找到, 再通过操作系统的存储器管理部分进行转换。

交换

当一个进程需要把一个虚拟页面调入物理内存时, 如果系统中已没有空闲的物理页面, 操作系统必须丢弃其中的某些物理页面以换取空间, 此过程称为交换 (swap)。

如果被丢弃的页面自上次换入以来未被修改过, 则可以不需要保存, 直接用新的页面将它覆盖即可, 因为磁盘上有它的备份, 下次再使用时可以重新从磁盘中换入。但如果页面被修改过, 则操作系统必须将它保存在交换文件中, 以备下次访问时再次将其换入内存。由于需要通过硬盘访问, 换入换出的操作非常耗时, 操作系统必须设计一个高效的算法, 尽可能减少交换的频次。

一个最著名的换页策略是最近最少使用 (**L**east **R**ecently **U**sed, LRU) 算法。其基本思想是, 近期使用过的, 在不远的将来再次使用的可能性很大; 长期不用的, 以后很可能再也不用了。LRU 策略为系统中的每个页面设置一个年龄, 它随页面访问次数而变化, 长期不被访问的页面将逐渐老化。交换时, 老化的页面将被先行换出。

图 2.13 中, 在对进程 B 的虚拟页面 1 的映射过程中, 假设内核选择了物理页面 4 作为替换对象, 它要做的事情是: 根据进程 A 对应页面的入口情况决定是否将物理页面 4 的数据保存到磁盘, 再将页表项入口的 P (present) 位置为无效; 将进程 B 的虚拟页面 1 的页表入口填入对应的物理页面号, 并标记该页面有效。

虚拟存储技术和高速缓存技术在设计上有一些共同的特点, 它们都是用小容量的高速存储设备代替大容量低速廉价的存储设备, 都使用了存储器分块的方法, 在空间不足时都需要使用交换技术, 但二者的设计存在明显的不同:

- 目的不同。虚拟存储技术是为了扩大系统的地址空间, 高速缓存技术是为了提高对存储器的访问速度。
- 实现方法不同。虚拟存储的换页可以通过操作系统内核的软件方法实现, 而高速缓存的交换必须通过硬件实现。操作系统只在启动过程中对高速缓存控制器初始化, 决定映射方式和映射策略。此后软件不会再关心主存地址和 cache 地址的关系。高速缓存对软件是透明的, 而虚拟存储仅对操作系统的应用软件是透明的。

2.5 文件系统

文件系统是管理存储数据的软件, 它将存储在物理介质上的数据以某种方式组织起来, 使用文件和目录这样的逻辑概念代替物理存储设备使用的数据块的概念。不同的组织方式就形成了不同的文件系统。文件系统是操作系统核心模块的一部分, 多文件系统支持是 Linux 的一个显著特征。

2.5.1 硬盘分区

硬盘是计算机的常用外存储设备。出于数据安全、磁盘文件格式限制等多种因素的考虑, 常常在一块大容量的硬盘上划分出多个小块。

个人计算机的主板上装有一段固化的程序, 它保证系统上电、处理器复位后将首先得到执行。不同处理器的开始地址不同, 取决于处理器复位后程序计数器的初始值。对于 Intel X86 (或 AMD) 的个人计算机, 这段程序习惯上就叫作 BIOS, 它从内存地址 0x000FFFF0 处开始运行。

在系统启动阶段, BIOS 执行上电自检过程, 包括检查 CPU、存储器、I/O 接口等等, 并对这些设备进行必要的初始化。其中一个重要的步骤, 是根据主板上 CMOS [3] 的参数设置, 确定一个可引导操作系统的外部存储设备 (光盘、硬盘或 U 盘等等), 并将这个存储设备的第一段数据块装入内存并尝试运行。

2.5.2 硬盘的逻辑结构

主引导记录 (MBR, **m**aster **b**oot **r**ecord) 位于硬盘的第一物理扇区 (sector)。由于历史原因, 硬盘的一个扇区大小是 512 字节。它包含最多 446 字节的启动代码、4 个硬盘分区表 (partition table) 项 (每个表项 16 字节, 共 64 字节)、2 个签名字节 (0x55, 0xAA)。分区表的结构见表 2.2。

由于硬盘的主引导记录中只有 4 个分区表项, 因此一块硬盘最多只能划出 4 个分区。这样划出来后, 每个分区被称作主分区 (primary partition)。在分区内部的扇区是连续的, 但不要求分区与分区之间连续, 也不要求分区编号在物理扇区上有先后顺序关系。

如果需要更多的分区, 则可以把其中一个主分区设置为扩展分区 (extended partition), 由分区表项中的分区类型标识。而这个扩展分区又可视为一块逻辑硬盘, 再次对它进行分区划分。在扩展分区上划出的分区称为逻辑分区 (logical partition)。理论上, 逻辑分区的数量没有限制。

[3] 此处 CMOS 专指个人计算机主板上的一块存储设备, 它由 CMOS 电路构成。

偏移地址	字节数	内容
00	446	引导代码
446	16	第 1 分区项
462	16	第 2 分区项
478	16	第 3 分区项
494	16	第 4 分区项
510	2	签名 (0x55 0xAA)

图 2.15: 经典主引导记录布局

表 2.2: MBR 中的硬盘分区表项

标记	字节数	含义
活动	1	该分区是否为活动分区 (0x80 或 0x00)
起始地址	3	CHS 起始扇区地址 柱面 (Cylinder)10 位 磁头 (Header)8 位 扇区 (Sector)6 位
分区类型	1	该分区的类型 (文件系统格式)
结束地址	3	CHS 结束扇区地址 (结构与起始扇区地址相同)
LBA	4	起始扇区的逻辑块地址
扇区数	4	该分区的扇区数

随着硬盘容量的增加, MBR 结构限制了硬盘分区的能力。近年生产的计算机越来越多地采用 GPT (**G**UID **P**artition **T**able) 取代 MBR, 它使用 8 个字节的 LBA[4] 代替 MBR 中的 CHS[5]结构。MBR 允许 4 个主分区, 最大单个分区容量能达到 2TiB, 而 GPT 允许 128 个分区, 最大单个分区容量可到 8ZiB (1TiB=2^{40} 字节, 1ZiB=2^{70} 字节)。

2.5.3 Linux 文件系统类型

Linux 通过单一的树状结构访问所有的文件系统。文件系统通过挂载 (`mount`) 被安装到文件系统层次树中。安装的文件系统注册到虚拟文件系统 (**V**irtual **F**ile **S**ystem, VFS)。虚拟文件系统为用户层软件提供统一的系统调用 (system call) 接口。Linux 的虚拟文件系统支持三种类型的文件:

1. 基于磁盘、光盘及 U 盘等非易失性存储介质的文件系统, 这类文件系统为数众多, 包括传统的 FAT 系列、NTFS、Ext 系列、HFS、光盘 UDF、ISO9660、以及针对闪存的文件系统 JFFS2 (**J**ournal **F**lash **F**ile **S**ystem version 2)、YAFFS2 (**Y**et **A**nother **F**lash **F**ile **S**ystem, version 2)。由于闪存使用寿命有擦除次数的限制, 这类文件系统特别考虑了损耗平衡问题。

[4]**L**ogical **B**lock **A**ddress, 逻辑块地址

[5]**C**ylinder **H**eader **S**ector, 柱面-磁头-扇区

2. 通过网络协议实现的远程网络的文件系统, 如 NFS、SAMBA/CIFS、NCP、CODA 等等;
3. 特殊文件系统, 如通过内核实现的进程管理 PROCFS、系统管理 SYSFS 等。它们有文件系统之形, 但无文件之实, 也叫虚拟文件系统。为了与 VFS 相区别, 有的地方又称其为伪文件系统 (pseudo filesystem)。这类文件系统只在 Linux 系统内部使用。

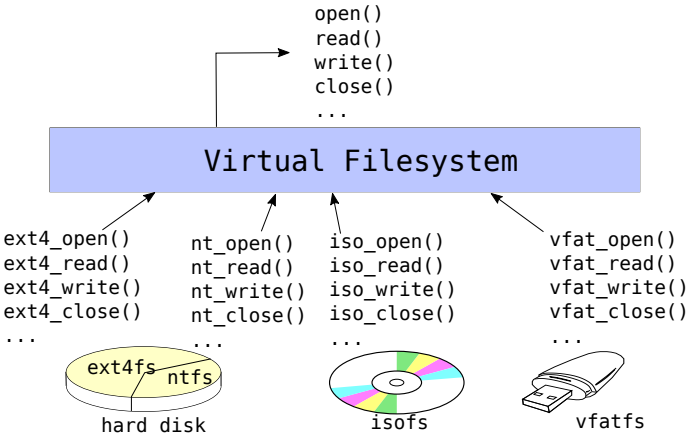


图 2.16: 虚拟文件系统的地位

不论文件系统属于什么类型, 挂载 (mount) 一个文件系统时都会将其加入到文件系统的目录树中。根文件系统 (root filesystem) 比较特殊, 它是在内核启动时挂载的。包含文件系统的设备就是块设备 (block devices)。Linux 文件系统认为这些块设备是简单的线性块集合, 它不需要了解数据块应该放在物理介质上的什么位置, 这些都是设备驱动的任务。设备驱动 (device drivers) 程序把对数据块的请求映射到正确的物理设备上 (磁盘、磁道、扇区等参数)。为了便于管理, 存储文件时, 文件系统以固定大小的块为单位进行分配。即使是一个字节大小的文件, 在 4096 字节单位块的文件系统上也要独占 1 个块 (4KB)。

2.5.4 FATFS

文件定位表 FAT (**F**ile **A**llocation **T**able) 文件系统由微软开发, 最早用在 DOS 操作系统。目前大多数主流操作系统都支持这种文件系统格式。FAT 将存储数据按簇 (cluster) 单元进行管理。若干连续的扇区构成一簇, 簇的大小随 FAT 类型及分区大小而变, 典型的簇大小介于 2KB 到 32KB 之间。表示簇的地址可以是 12 位、16 位或 32 位, 由此形成了 FAT12、FAT16、FAT32。

被格式化成 FAT 文件系统的分区包含下面四个主要部分:

- 保留扇区, 位于分区最开始的位置, 其中第一个扇区 (512 字节) 是引导扇区, 它包含文件系统的基本信息、系统启动代码。
- FAT 区, 这个区域包含两份文件定位表, 其中一份作为备份。文件定位表是分区信息的映射表, 标示簇的存储方式。
- 根目录区, 存储根目录文件和目录的信息的目录表。
- 数据区, 实际文件和目录数据的存储区域。

文件目录项由一个 32 字节的条目表示, 它记录了文件名 (文件名 8 字节、扩展名 3 字节, 俗称 “8.3 格式”)、文件属性、创建时间、首簇地址和文件/目录大小。存储在 FAT 分区

中的文件以簇为单位, 在磁盘上可能占有一个或多个簇, 这些簇按顺序形成一个单向链表。这些簇通常是零散地分布在磁盘上的, 组织这个链表的就是 FAT, 也就是 FAT 区的内容。

FAT 中的每个条目记录下面的六种信息之一:

- 0: 空闲簇;
- 1: 保留簇;
- 0xFF7: 坏簇。(FAT16 为 0xFFF7, FAT32 为 0x?FFFFFF7。以下表示方式类似);
- 0xFF0–0xFF6: 保留值;
- 0xFF8–0xFFF: 文件最后一个簇;
- 2–0xFEF: 链中下一个簇的地址。

FAT 文件系统的优点是实现简单, 在各种操作系统中都能够很好地兼容。缺点是当文件碎片化严重时影响读写性能 (主要指机械式磁盘, 因为机械式磁盘需要反复寻道), 并且文件和分区大小受限比较严重。FAT16 支持的最大分区是 2GB (簇大小 32MB、簇寻址 16 位)、最大文件 2GB, FAT32 理论上支持 8TB 的分区, 但最大文件仍不能超过 4GB。

2.5.5 Ext2FS

最初的 Linux 只支持 MINIX[6]文件系统。由于 MINIX 本身是为教学目的设计的, 有很大的局限, 它能够支持的最大空间只有 64MB, 文件名最长 14 个字符。1992 年 4 月, 第一个专为 Linux 设计的文件系统 ExtFS (**Ext**ended **F**ile **S**ystem, 扩展文件系统) 进入 0.96c 版的 Linux 内核。它支持的最大文件是 2GB, 最大文件名长度为 255 个字符。随后在此基础上又开发了功能较为完善的 Ext2FS 以及日志型文件系统 Ext3FS 和 Ext4FSExt4FS。Ext 系列的文件系统被大多数 Linux 发行版作为缺省的文件系统选项。这里以 Ext2FS 为例说明文件系统的组织原理。图 2.17 是某一分区上 Ext2FS 的布局。

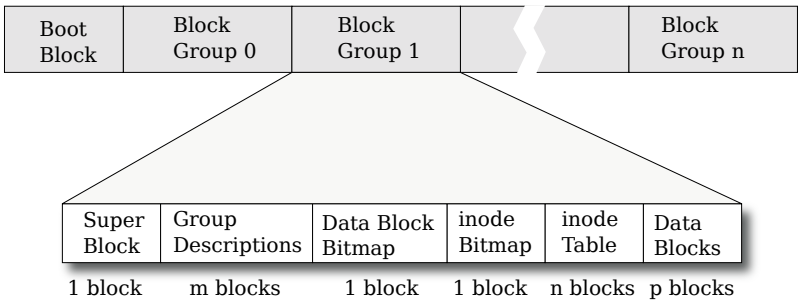


图 2.17: Ext2 分区布局

Ext2FS文件系统在格式化时, 将分区按 1024 字节或 2048 字节 (或其他 2 的整数次幂单位) 划分成大小相同的块 (Block), 不同分区上块的大小可以不同, 但在同一个 Ext2 文件系统上, 块的大小是相同的。若干块又被聚在一起形成一个组块 (Block Group), 每个组中块包含的块数也是固定的。

超级块 (Super Block) 中包含了描叙文件系统基本尺寸和形态的信息, 文件系统管理器利用它们来使用和维护文件系统, 第 0 组的超级块在挂载时由内核添加到 VFS 中, 其他组的超级块很少用到。为了防止文件系统被破坏, 第 1 组块包含了一个备份超级块, 用于在文件系统故障时通过 `fsck` 命令修复。

[6]MINIX 原是荷兰裔美籍计算机科学家泰能鲍姆 (Andrew Stuart Tanenbaum, 1944.3.16–) 为其操作系统教材而编写的类 UNIX 操作系统 (Mini-UNIX)。

组描述符 (Group Descriptors) 记录了组内数据块位图和 inode 位图的块号, 以及创建目录的数量。同超级块一样, 只有第 0 组的组描述符才会被内核用到。

块位图 (Block Bitmap) 和 inode 位图的每一位用来标记一个组块中块的占用/空闲情况。当块大小设置为 1KB 时, 1 个组就是 8K 个块, 在一个 32GB 的分区上最多有 4K 个组。

inode 是 Ext2 文件系统 (实际上也包括其他 inode 型文件系统) 最重要的元数据 (metadata) 信息, 文件的属性和数据入口都保存在 inode 中。每个 inode 的大小都是一样的 (128 字节或 256 字节), 并且也是在分区格式化时就已经规划好的。图 2.18 是 Ext2 文件系统的 inode 结构。

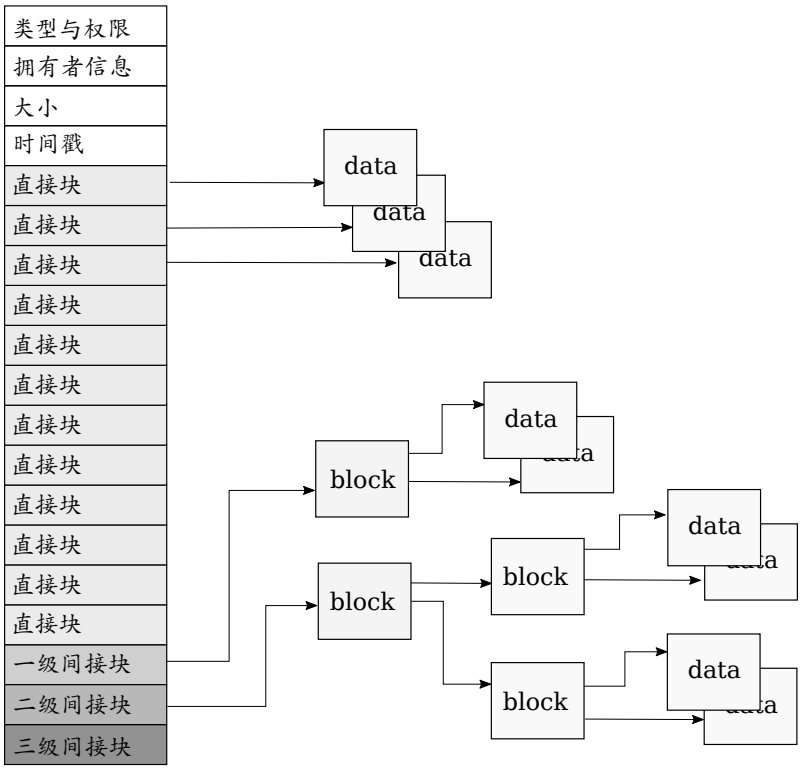


图 2.18: Ext2FS inode 结构

除了文件属性信息外, inode 通过直接块地址和多级间接地址标记数据所在的位置。仍以 1KB 大小的块为例: 每个直接块地址指向一个数据块, 当文件不超过 12KB 时, 这个文件的所有数据位置就可以从这 12 个直接块中找到。超过 12KB 的文件, 开始使用间接块查找数据。每个块地址 4 个字节, 一级间接块的 1KB 数据包含了 256 个块地址, 最大可以寻访 256×1KB 数据。当动用到三级间接块时, 可访问的最大文件超过 16GB ($256^3 \times 1$KB)。1KB 块的 Ext2 文件系统可支持最大的分区是 2TB。更大的文件或更大的分区, 可通过格式化时选择更大的块来实现。

当要读取文件时, 文件系统首先根据从超级块中找到的根目录的 inode, 去读取根目录的数据块。这个数据块存放着根目录一级的文件和目录名称及其对应的 inode。进而再从这些 inode 中逐级向下寻找, 直至找到文件的 inode, 从这个文件的 inode 中读取文件的数据。

由此我们知道, 目录也是文件, 也是通过 inode 表示的。不过目录的数据块中存放的不是通常意义下的数据, 而是该目录下的文件和子目录的名称以及对应的 inode。这种文件名和 inode 分离的方式, 使得构造硬链接 (hardlink) 非常容易: 只要将两个文件的 inode

设成同一个值就可以了, 此时两个文件指向同一个数据块。

虽然 Ext2FS 稳定可靠, 但缺乏日志功能, 在出现意外事故时数据很难恢复, 目前在主流发行版中已被 Ext4FS 取代。Ext4FS 是功能完备的日志型文件系统。

■ 本章练习

1. 简述几种半导体存储器的特点。
2. SRAM 和 DRAM 代表什么意思, 各有哪些优点和缺点?
3. 存储器的寻址范围是怎样确定的?
4. 设处理器对 DRAM 的访问时间是对 SRAM 访问时间的 10 倍, DRAM 的单位价格是 SRAM 的 1/10, 当 cache 容量达到主存的 1% 时, cache 命中率可以达到 80%。试计算此存储器系统的性能与极限性能 (全部采用 SRAM) 的差距是多少, 价格差距是多少。
5. 若存储空间的首地址为 0x1000, 写出存储器容量分别为 1K×8, 2K×8, 4K×8 和 8K×8 时对应的末地址。
6. 编译器为什么常常将存储器地址按 4 字节对齐设置?
7. 主存储器与高速缓冲存储器之间有哪些地址映像方式?
8. 微型计算机系统中, 使用的高速缓存大小范围通常是多少?
9. 有哪些措施保证主存与高速缓存的一致?
10. 设有一个 1024×1024 的二维数组, 顺序按行访问和按列访问的效率有什么不同?

第 3 章

LINUX 的使用

3.1 Linux 基本介绍

3.1.1 Linux 的诞生

1964 年, 由 MIT (**M**assachusetts **I**nstitute of **T**echnology) 牵头、通用电器 (GE, **G**eneral **E**lectric Company) 和美国电报电话公司 (AT&T, **A**merican **T**elephone and **T**elegraph Inc.) 下属的贝尔实验室参与, 开始计划一个宏大的项目: 在 GE 645 计算机上建立一个具备众多功能的信息应用工具, 能够支持上百个用户对大型计算机的交互式分时使用。它被命名为多功能信息计算服务系统 (**Mult**iplexed **I**nformation and **C**omputing **S**ervice, Multics)。遗憾的是, 由于项目的预期过于宏伟, 限于当时的计算机技术水平, 该项目历经磨难, 成为一个烂尾工程。

1969 年, 贝尔实验室从 Multics 项目中退出。其研究人员肯 · 汤普森[1] 等人带着从 Multics 项目中激发的灵感开始了新的创作, 并将自己的思想付诸于一台小型计算机 DEC PDP-7。相比 Multics 的设计目标, 这个操作系统相当简单, 它实现了一个文件系统、一个命令解释器和一些简单的文件工具, 是一个单任务系统。它被创造者们戏谑地冠名为 Unics (**Uni**plexed **I**nformation and **C**omputing **S**ervice)。后来, 又有人根据发音把它改成了 UNIX。

最初的 UNIX 是用汇编语言写成的, 应用程序则是用 B 语言写成。B 语言是一种解释性编程语言, 作为系统编程语言远远不够。直到布莱恩 · 科尼翰 [2] 和丹尼斯 · 里奇[3] 发明了 C 语言, 这个情况才开始发生变化。

1973 年, 汤普森和里奇用 C 语言重写了整个 UNIX 系统。从此, UNIX 操作系统开始了令人瞩目的发展。由于当时针对 AT&T 一项法律判决的限制 [4], AT&T 被禁止进入计算机相关的商业领域, 所以, UNIX 非但不能成为商品出售, 而且贝尔实验室还必须将非电话业务的技术许可给任何提出要求的人。由此, UNIX 的源代码被散发到各个大学和研究机构, 一方面使得科研人员能够根据需要改进系统, 或者将其移植到其他的硬件环境中去, 另一方面培养了大量懂得 UNIX 使用和编程的学生, 这使得 UNIX 的普及更为广泛。在

[1] Kenneth Lane Thompson(1943.2.4–), 1966 年加州大学伯克利分校电子工程与计算机科学硕士, B 语言的发明人, 1983 年图灵奖获得者, 长期供职于贝尔实验室。2006 年进入谷歌, 在此期间同他人合作发明了 Go 语言。

[2] Brian Wilson Kernighan (1942.1.1–), 生于多伦多, 加拿大计算机科学家, 1964 年多伦多大学工程物理学士, 1969 年普林斯顿大学电子工程博士。C 语言的发明人之一, K&R C 中的字母 K。他也是有记载的著名的 “Hello world” 程序的 C 语言样板原创者。

[3] Dennis MacAlistair Ritchie (1941.9.9–2011.10.12), 生于纽约, 美国计算机科学家, 毕业于哈佛大学, C 语言的发明人之一,K&R C 中的字母 R。因开发 UNIX 操作系统于 1983 年和 Ken Thompson 共同获得图灵奖。

[4] 1956 年, AT&T 与美国司法部达成和解协议, 承诺不从事电话和政府项目之外的业务, 从而结束了长达 7 年的反托拉斯案诉讼。

这期间, 加州大学伯克利分校计算机系统研究小组 CSRG (**C**omputer **S**ystems **R**esearch **G**roup) 是一个最重要的学术热点。他们在 1974 年就开始了对 UNIX 的研究, 他们的研究成果就反映在他们使用的 UNIX 中。他们对 UNIX 做了相当多的改进, 增加了很多当时非常先进的特性, 大部分原有的源代码都被重新写过。他们的创意和代码不断反馈到贝尔实验室。很多其他 UNIX 使用者都希望能得到 CSRG 改进版的 UNIX 系统。因此 CSRG 的研究人员把他们的 UNIX 做成一个完整的发行版 (distribution) 对外发布, 这个发行版的名字就叫 BSD (**B**erkeley **S**oftware **D**istribution) UNIX。

至此, UNIX 就有了两个分支: 一个是来自 AT&T 的 UNIX, 一个是 BSD 的 UNIX 发行版。现代 UNIX 的版本大部分都是这两个发行版的衍生产品。AT&T 的 UNIX 也被称作 System, 并用罗马数字 III、IV、V 标记版本。(事实上, System IV 从未出现过)

芬兰赫尔辛基大学计算机科学专业的学生李纳斯 · 托瓦兹 [5] 在学习计算机操作系统课程中, 研究了泰能鲍姆教授编写的教材《操作系统: 设计与实现》。在此之前他已购买了一台 386 的个人计算机, 因此得以将学习的 MINIX 代码在个人计算机上实施。MINIX 是泰能鲍姆为教学目的编写的一个类 UNIX 操作系统—mini-UNIX。1991 年 9 月, 李纳斯把他初步完成的操作系统上传到大学的 FTP (**F**ile **T**ransfer **P**rotocol) 服务器。这标志着 Linux 操作系统的诞生。

Linux 以其可靠性和稳定性, 迅速进入服务器领域。全世界超级计算机统计网站上 https://www.top500.org 记载, 1998 年 11 月, 性能最强的前 500 台超级计算机 99% 以上都使用 UNIX, 只有 1 台在使用 Linux, 次年 6 月, 这个数字迅速上升到 17 台, 到 2018 年 6 月, 该榜单已全部被 Linux 操作系统占据。

在移动设备上, Linux 很早就进入了市场竞争, 早期的个人数字助理 (PDA, **P**ersonal **D**igital **A**ssistant)、手机都有它的身影。自从谷歌发布 Android 系统以后, 除苹果公司以外的几乎所有的移动设备生产厂商都开始采用这种基于 Linux 内核的操作系统, 以打造手机和平板电脑。据统计, 到 2018 年 8 月, Android 系统的手机市场占有率 72.9%, 平板电脑几乎与苹果的 iOS 平分天下。

3.1.2 Linux 操作系统的特点

Linux 是一种可以在 PC 上运行的类 UNIX 操作系统, 它与其他商业性的操作系统最大的不同点在于它的开放性。Linux 的源代码是完全公开的, 用户可以在网上自由下载、复制和使用。Linux 之所以能在短短十几年的时间得到迅猛的发展, 与 Linux 所具有的良好特性是分不开的。Linux 操作系统具有以下主要特点:

- 可靠性、稳定性

  Linux 运行于保护模式, 内核态和用户态地址空间分离, 对不同用户读、写权限控制, 带保护的子系统及核心授权, 多种安全技术措施共同保障系统安全。再加上良好的用户使用习惯, 使得一个系统能够长期稳定地运行。

  良好的用户使用习惯是一个重要因素。Linux 操作系统对用户权限做了合理的设计, 普通用户使用计算机时, 无需使用系统管理员权限就可以完成绝大部分的日常工作。错误的操作也不会对系统造成损害。在这种情况下, 即使恶意软件入侵系统, 也只能以普通用户的身份运行, 不会危及整个系统, 病毒也不具备传播能力。破坏这个规则, 系统安全性仍然是一个问题。Android 系统 root 后[6]遭受恶意软件侵害的例子并不鲜见。

[5] Linus Benedict Torvalds (1969.12.28–), 生于芬兰赫尔辛基, 软件工程师, 1996 年赫尔辛基大学计算机科学硕士学位, 2010 年入籍美国。

[6] root 是类 UNIX 操作系统的超级用户名, 此处指利用 Android 系统的漏洞获得 root 权限的一种行为。

Linux 的内核的源代码是公开的, 几乎所有应用软件的源代码也都是公开的, 不存在暗箱。只有少数发行版的极少数软件是闭源的, 并且也不是必须的选择。这种情况下, 很难有恶意代码的容身之地。世界各地的软件工程师和 Linux 用户都在热心地为开源社区提出建议。一旦出现 bug, 可以很快得到修正。稳定、可靠的 Linux 操作系统成为绝大多数服务器的首选。

- 良好的可移植性

  Linux 软件开发遵循 POSIX 标准, 在不同平台之间不经修改或只需要很少的修改就可以直接使用。Linux 内核支持包括 Intel、Arm、PowerPC 等在内的数十种处理器架构和上百种硬件平台, 小到掌上电脑、可穿戴设备, 大到超级计算机、集群计算都可以看到 Linux 的身影。

- 设备独立性

  Linux 系统中, 有 “一切皆是文件” 之说, 所有设备也统一被当作文件看待。操作系统核心为每个设备提供了统一的接口调用。应用程序可以像使用文件一样, 操纵、使用这些设备, 而不必了解设备的具体细节。在每次调用设备提供服务时, 内核都以相同的方式来处理它们。设备独立性的关键在于内核的适应能力。它带来的好处是, 用户程序和物理设备无关, 系统变更外设时程序不必修改, 提高了外围设备分配的灵活性, 能更有效地利用外设资源。

- 多种人机交互界面

  Linux 从初期单调的字符界面, 发展到如今丰富的图形化人机接口和功能强大的命名行方式并存, 可以满足不同系统资源的需求。传统的命令行界面利用 shell 强大的编程能力, 为用户扩充系统功能提供了更高效便捷的手段。Linux 的图形桌面有多种选择, 可以充分体现用户的个性化设置。

- 多用户、多任务支持

  UNIX 设计之初就是为了满足多人使用计算机的需求。Linux 继承了 UNIX 操作系统的这一特性, 它天生是一个多用户操作系统。除了像其他运行在个人计算机的操作系统一样, 可以为每个用户分配独立的系统资源以外, 还可以让多人同时使用一台计算机。这一点, 与专为个人计算机设计的操作系统不一样。

  多任务是现代操作系统的一个重要特性, 它是指计算机同时执行多个程序, 并且各个程序的运行互相独立。Linux 系统调度为每一个任务独立分配处理器资源和存储空间, 相互之间不会受到干扰。某个任务的失败一般不会影响到其他任务。这为系统的可靠性提供了保障。

- 完善的网络功能

  Linux 操作系统具备 UNIX 的全部功能, 包括 TCP/IP 网络协议的完备实现, 同时也支持完整的 TCP/IP 客户与服务器功能, 具有强大的网络通信能力。Linux 还具有开放性, 可以支持各种类型的软件和硬件, 同时具备先进的内存管理机制, 能更加有效地利用计算机资源。

- 多种文件系统支持

  Linux 通过虚拟文件系统层实现对不同文件系统的支持, 几乎可以识别目前所有已知的磁盘分区格式, 软件不经修改就可以对不同分区的文件进行读写操作。

- 便捷的开发和维护手段

  Linux 各发行版提供了多种编程语言开发工具, 此外还大量地使用了脚本语言 (script language), 除了方便编程以外, 还为系统的可维护性建立了基础。Linux 的各种服务都

是通过脚本程序维护的, 服务器的功能也是通过脚本文件配置的, 包括系统的启动过程, 也都是通过脚本程序完成。管理员可以使用任何自己熟悉的文本编辑工具管理计算机。

从发展历史来看, Linux 是从成熟的 UNIX 操作系统发展而来的, 技术上具备 UNIX 系统的几乎所有优点。并且 Linux 是开源的、可以自由传播的操作系统。

3.1.3 Linux 的发行版

我们平常所说的 Linux 可能有两个意思, 一是指 Linux 内核, 二是指以 Linux 内核为核心外加大量的 GNU 软件构建的一整套操作系统。为了区分, 后者常常又以发行版名称来指代。

Linux 内核遵循 GPL 开源版权协议, 同时全世界成千上万的软件开发人员为开源世界贡献了无以计数的应用软件, 这些软件足以打造一个功能强大的操作系统。任何人, 只要遵守版权协议, 无需付费, 都可以根据自己的设计方案构建一个完整的操作系统, 由此就出现了众多的 Linux 发行版。这种类型的发行版, 软件界习惯称 GNU/Linux, 它表示基于 Linux 内核的、以 GNU 软件为基础构成的操作系统, 以区别于虽是 Linux 内核、但应用层不是 (或不主要是)GNU 的软件的系统, 如 Android, 或者虽是 GNU 软件, 但不是基于 Linux 内核的操作系统, 如 GNU/Hurd。

目前比较普遍使用的 Linux 发行版有 Debian、Ubuntu、Fedora、Gentoo 等。不同的发行版各有不同风格, 但功能差别并不明显。对使用者来说, 主要是使用习惯上的差异。而对管理员来说, 服务器和安装包管理软件会有一些差别。

3.2 命令行工作方式

命令行 (**C**ommand **L**ine **I**nterface, CLI) 工作方式是类 UNIX 系统的特色, 也是一种非常重要的工作方式。它占用资源少, 效率高, 快捷方便, 能够实现对计算机系统的完全控制。图形界面工具对发行版依赖较多, 而命令行工具基本与发行版无关。很多嵌入式 Linux 系统由于资源所限, 必须使用命令行方式交互。在图形化界面日益丰富的今天, 命令行方式非但没有消亡, 反而越来越受到开发人员的重视。“图形界面方式让容易的事变得容易, 而命令行方式让困难的事变得可能。”[7]

3.2.1 命令行的特点

Linux 操作系统提供两种典型的操作界面：命令行界面和图形用户界面 (**G**raphical **U**ser **I**nterface, GUI)。支持命令行工作方式的环境, 我们把它叫作 shell。shell 是命令行解释器, 它是键盘与操作系统进行交互的程序。类 UNIX 系统中曾经使用过很多种 shell。目前 Linux 系统中使用最普及的是 bash (**B**ourne **A**gain **SH**ell)。[8]

3.2.2 两种字符界面

命令行界面又称字符界面。广义的字符界面还包括图形界面中的终端 (terminal) 环境。Linux 桌面系统中, 有两种使用命令行的方式: 在图形桌面菜单选项中选择 “终

[7]源自《Linux 命令行》的作者 William Shotts 的引述:“Graphical user interfaces make easy tasks easy, while command line interfaces make difficult tasks possible.”。原始出处已不可考。

[8]Bourne Shell 由 Bell 实验室研究人员 Stephen Richard Bourne 开发, 作为 1979 年发布的第 7 版 UNIX 默认的命令行解释器。bash 是 GNU 项目的 shell, 开发者是美国软件工程师 Brain J. Fox。

端”(Terminal), 会创建一个终端窗口, 又叫终端模拟器。Linux 允许在图形桌面上打开多个终端窗口。Ubuntu 的默认终端是 gnome-terminal, 多个终端窗口也可以以标签的方式合并在一个窗口中, 通过组合键 CTRL+PgUp/PgDn 在终端之间切换。这个窗口虽然是图形化界面的一部分, 在窗口管理器下工作, 但窗口内只能以字符为操作单位, 用户必须通过键盘与系统进行对话, 鼠标在多数情况下无法发挥作用。不懂得命令的用法, 就不能有效地在终端中使用计算机。

第二种是纯字符界面。UNIX 系统最初的设计只有字符界面, 这是真正的终端 (不是模拟器), 它只需要一个字符显示器和一个键盘, 每个终端通过调制/解调器连接到主机, 多个 UNIX 用户通过各自的界面同时使用一台计算机。图形终端是在 UNIX 系统发明后很晚才出现的。Linux 系统的字符界面总是可用的——即使在图形界面发生故障或是崩溃时。在个人计算机系统上, 如果使用 lightdm 作为显示管理器, 纯字符界面可以通过控制键 CTRL+ALT+Fn (n 从 1~6) 进行切换。默认的系统安装下, 这六个都是字符界面。从 CTRL+ALT+F7 之后则都是图形界面。通常桌面系统只启动了一个图形界面, 因此 CTRL+ALT+F8 之后的切换是不起作用的。

获得终端控制权后, 也可以用命令 `chvt`切换终端。

对于开发者来说, 纯字符界面比窗口终端更具实用价值。它无需启动图形界面, 不需要图形显示器, 占用资源极少, 硬件开销极低, 特别是通过网络访问时, 对双方的计算机配置要求都很低。纯字符界面不能使用鼠标。

3.2.3 命令行的格式

命令行中, 一条完整的命令格式是:

```
$ command [options] [arguments]
```

命令行中的选项 `options` 告诉命令以何种方式执行, 通常前面有 “-” 或 “--”。前者又被称作 “短选项”, 属于 UNIX 风格, 使用很少的字符 (通常是一个字母或数字) 表示, 后者又被称作 “长选项”, 属于 GNU 风格, 用完整的单词或词组表示。多个选项有时候可以合并在同一个 “-” 后。少数情况下, 选项字母前面没有 “-” (属于 BSD 风格。相同字母的选项, 有 “-” 的和没有 “-” 的功能可能有差别)。

参数 `arguments` 通常是命令的操作对象。有些命令可以没有选项和参数, 有的命令可能有不止一个选项或参数, 有的选项要求后面紧跟它的参数。

命令、选项和参数之间用空格分开。命令行中, 所有与 Linux 系统操作相关的字母都是大小写敏感的[9]。

命令行这种带有选项和参数的特点使得它在运行程序时具有很高的具有灵活性。所有在图形界面下通过鼠标点击启动的程序, 都可以通过命令行输入命令的方式启动, 反之则不必然。例如使用火狐浏览器可以用下面的命令直接打开指定网页:

```
$ firefox www.google.com
```

而使用鼠标则缺乏这种灵活的手段。

[9]有些文件系统 (如 FAT 系列的文件系统 FAT16、FAT32、VFAT) 不区分文件名的大小写字母。一个被格式化成 FAT 的 U 盘, 访问这个盘上的文件时, 文件名就不必区分大小写。这属于文件系统的设计问题, 与 Linux 的命令行没有直接关系。

3.2.4 快捷键和符号

在命令行中运行的程序, 有时候我们想让它提前终止, 或者因为屏幕显示内容过多, 想让它暂停, 可以通过一些快捷键进行操作。表 3.1 罗列了命令行操作中常用的快捷键, 方便用户对程序的控制。

表 3.1: 命令行使用的快捷键

快捷键	功能
CTRL+C	终止当前任务 (发送 TSTP 信号)
CTRL+\	终止当前任务 (发送 QUIT 信号)
CTRL+L	清屏, 相当于命令 clear
CTRL+S	暂停终端打印输出
CTRL+Q	继续终端打印输出
CTRL+Z	进程挂起 (见3.2.8节)
CTRL+D	退出当前终端 (相当于 exit 命令)

终端支持大多数命令和参数的自动补全功能。对于较长的命令或参数, 无需挨个将字母敲完, 只要输入前几个字母, 在适当的时候使用制表符键, 命令或参数就会自动补齐。“适当的时候” 是指已敲出的字母在当前候选命令或参数中只有一个。例如, 设置网络地址的命令 `ifconfig`, 敲完前三个字母 “ifc” 接着敲一个 TAB, 如果系统中没有其他以这三个字母开头的命令, 完整的`ifconfig` 命令就出来了。

使用自动补全功能的好处不仅仅是快捷方便, 更重要的是不容易出错。

3.2.5 目录

在任何一个环境中生存, 首先要熟悉这个环境。进入命令行方式, 直接面对的就是文件系统环境。本节讨论对目录 (directory) 和文件的基本访问方法。

Linux 系统中, 所有文件和目录都被组织成一个树形结构, 即使不同的物理存储设备, 包括同一硬盘的不同分区、不同的外挂硬盘、U 盘、光盘, 以及网盘, 也都在这一棵树上。`cd` (**c**hange **d**irectory) 是 shell 的内部命令, 表示改变工作目录, 两个点 “..” 表示上一级目录。`cd` 通常带有一个参数或不带参数。不带参数时, 表示回到个人用户的主目录 (环境变量 `HOME` 指向的目录)。参数表示转移到的目的地目录。“~” 也表示用户的主目录, 它在不允许参数空缺时使用。例如将文件`file.txt` 复制到主目录:

```
$ cp file.txt ~                    # cp 命令的第二个参数不能空缺
```

目录有两种表示形式, 分别叫做绝对目录和相对目录。考虑到目录和文件两种情形, 统称为绝对路径和相对路径。

绝对路径 从上到下所有目录依次串在一起, 一直到最末端的目录或文件, 每一层目录或文件之间用 “/” 分隔, 最上层的根目录 (树根) 用 “/” 表示。

相对路径 以当前位置为起点, 向上或向下走到枝杈上。与绝对路径不同的是, 相对路径的起点不是 “/”。

图 3.1 中, 从用户 harry 的主目录走到 /usr, 下面两条命令的参数分别是绝对目录和相对目录形式:

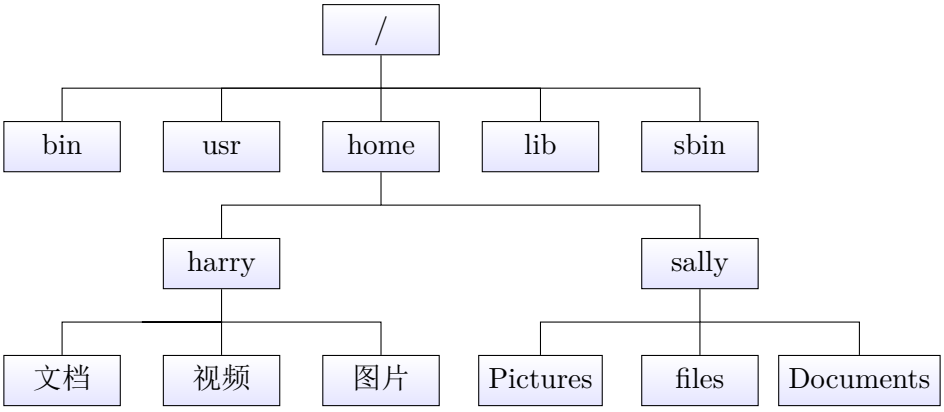


图 3.1: 与用户 harry 有关系的目录

```
$ cd /usr        # 绝对目录
$ cd ../../usr   # 上一级、再上一级、左转、进下一级usr
```

创建和删除目录

将一些相关的文件专门放在一个目录里, 可以让工作显得更有条理, 便于维护。例如, Ubuntu 系统会给每个用户创建文档、图片、视频... 等等目录。用户自己还可以根据自己的需要创建美食、旅游、股票... 等目录。创建目录的命令是 `mkdir` (**m**a**k**e **dir**ectories)。使用`mkdir` 可以一次创建多个目录, 只要将多个目录名作为命令的参数即可。如果创建的是多层目录, 但上级目录还不存在, 可以用选项`-p` 或`--parents`, 表示连同父目录一同创建:

```
$ mkdir -p foods travel/nanjing/friends travel/beijing stocks
```

一个空目录可以用 `rmdir` (**r**e**m**ove **dir**ectories) 删除。非空目录, 即使这个目录里只有一个文件长度为 0 的空文件, 也不能用 `rmdir` 删除。

当前目录

想知道自己现在在哪个目录下工作, 只要看看提示符就可以了 (系统默认设置的提示符格式)。但如果是程序想知道当前的工作环境呢? 这时可以用命令 `pwd` (**p**resent **w**orking **d**irectory) 输出当前工作目录:

```
$ cd ~/travel/nanjing/friends
$ pwd
/home/harry/travel/nanjing/friends
```

目录层次结构

Linux 系统的目录结构遵循文件系统层次结构标准 (**F**ilesystem **H**ierarchy **S**tandard, FHS)。该标准由 Linux 基金会维护, 它规定了操作系统中的目录结构及它们的内容。以下是构成 Linux 主要目录的内容。

/ 根目录 (root directory), 所有目录或文件的起点。一些嵌入式 Linux 会将初始化脚本 init 放在根目录, 或者将 /sbin/init 链接到根目录的 linuxrc。

/bin 系统基本命令, 这些命令是二进制 (**bin**ary) 可执行程序, 目录由此得名。单用户使用这些命令启动或者修复系统。系统正常运行时也为普通用户提供系统的基本操作。

/boot 存放系统引导 (Bootloader) 文件。该目录应该仅保存在系统引导进程 (process) 中使用的文件, 如 Bootloader 配置文件、内核镜像文件以及初始化 RAM Disk (initrd)。Linux 的初始化进程 (init) 及所需的配置文件应该在 /sbin 和 /etc 目录。

/dev 设备文件目录。该目录下包含系统已驱动的所有设备相关的设备文件 (字符设备和块设备), 它们可能是通过 `mknod` 命令创建的, 也可能是由内核的 DEVTMPFS 文件系统机制自动生成的。

/etc 存放主机的全局配置文件。一些规模较大的软件包会在该目录下、或者在二级目录 /usr 下建立自己的子目录, 用于管理自身的配置。/et c 目录下一般不存放二进制可执行文件。

该目录最初被 UNIX 命名时意为附属目录 (**et c**etera), 存放那些不明确属于哪个专属目录的文件。[10]

/etc 目录下还可能有一些分支:

- /etc/opt /opt 下附加包的配置文件目录。
- /etc/sgml 处理 SGML 软件的配置文件目录。
- /etc/X11 X-Window 系统的配置文件。
- /etc/xml 处理 XML 软件的配置文件目录。

/home 用户目录。用于存放除超级用户以外的个人用户的数据。个人用户的主目录在这个目录下。

/lib 用于支持 /bin 和 /sbin 程序的基础库文件存放目录。它有几个变体: /lib32、/lib64 等等, 针对不同的二进制格式。

/media 移动存储设备的挂载点, 包括光盘、U 盘。

/mnt 临时挂载的文件系统会连接到这个目录下。

/opt 存放可选的 (**opt**ional) 应用软件, 它们通常是在系统软件仓库之外的软件。

/proc 该目录下的文件提供系统的所有进程 (**proc**ess) 信息, 这些信息是在系统运行中自动产生, 并且是在不断地变化的。这样的功能, 在内核中通过 PROCFS 伪文件系统实现。

/root 超级用户的主目录。与普通用户不同, 超级用户的主目录不在 /home 中而在根文件系统中。由于 /home 通常是另一个分区的挂载点, root 的主目录位置保证了即使其他分区出问题时, root 也能正常工作。

/run 存放系统自启动以来的运行时可变数据, 如登录的用户、守护进程等等。这些数据对系统来说不是必须的, 通常在这个目录上挂载临时文件系统 TMPFS。

/sbin 系统级基本二进制命令 (**s**ystem **bin**ary) 存放目录, 普通用户不能使用这些命令修改系统设置。

[10]在 GNOME 邮件列表上, 有人将其解释为可编辑配置文本 (**E**ditable **T**ext **C**onfiguration) 或扩展工具箱 (**E**xtended **T**ool **C**hest)。

/srv 系统服务目录, 例如 FTP 服务器的上载/下载文件区、NFS 服务器的目录等等。

/sys 伪文件系统 SYSFS 的挂载点, 提供内核运行时信息, 包括设备、驱动程序和内核特性等等。

/tmp 临时文件目录。

/usr 二级目录结构, UNIX 系统资源 (**U**NIX **S**ystem **R**esources)。绝大多数应用软件都以这个目录为起点。这是 Linux 桌面系统中最大的目录。有的系统将它专门做成一个分区, 可以让不同的版本、不同的系统挂载。

/usr 目录下面还有明细的分工:

- /usr/bin 提供给所有用户使用的应用软件的二进制可执行程序目录。
- /usr/include C 语言的标准 include 文件。扩展开发包的 include 文件也会在此目录下独立创建一个子目录。
- /usr/lib 这个目录存放了 /usr/bin 和 /usr/sbin 二进制可执行程序使用的库文件 (共享库), 也包括开发软件包需要的静态库 (staticall linked library)。相应的变体 (如/usr/lib32) 是针对不同二进制格式的库。
- /usr/local 三级目录结构, 该目录下的程序或文件与主机系统可以没有直接关系, 通常也包含与一级目录结构或二级目录结构相似的目录。
- /usr/sbin 给管理员使用的二进制应用程序的目录, 例如守护进程、网络服务等等。
- /usr/share 存放与系统架构无关的应用程序数据, 如文档、手册等。
- /usr/src 源代码目录, 主要是 Linux 内核及其头文件。
- /usr/X11R6 X-Window 系统。

/var 存放变化的文件。这些文件在系统正常工作时会不断地变化, 例如日志文件等等。

FHS 标准最初的名称是文件系统标准 (Filesystem Standard), 1997 年发布 2.0 版时使用现名, 最新的 3.0 版于 2015 年 5 月 18 日发布。Linux 系统原则上遵循这一目录标准, 但也有发行版在一些细节上有变化。

3.2.6 文件属性

列文件清单

有关文件的操作, 最基本也是最常用的命令就是列表文件清单 `ls`。我们在个人主目录下不带选项不带参数地执行这条命令:

```
$ ls
examples.desktop  stocks   公共的   视频   文档   音乐
foods             travel   模板     图片   下载   桌面
```

如果终端支持彩色显示, 可以看到用不同颜色显示的不同文件 [11]: 蓝色表示目录文件, 红色表示压缩包文件, 绿色表示可执行文件, 粉红色表示媒体文件 (图像、声音、视频等), 浅蓝色表示链文件, 白色表示没有特别属性的文件...。颜色是表示文件属性最简洁明快的方式, 但不够详细。

[11] ls 命令本身并不直接显示颜色, 但通常它是加了选项 --color=auto 后的别名。试试在终端上执行 type ls 看看结果?

文件的完整属性

`ls` 一个较常用的选项 “-l” (long) 可以列出文件的详细信息, 一个不太常用的选项 “-i” 用于显示文件的索引结点 inode 号:

```
$ ls -li
6817091 -rw-r--r-- 1 harry harry 8980 9月  13 17:09 examples.desktop
6817035 drwxrwxr-x 2 harry harry 4096 9月  19 12:09 foods
6817045 drwxrwxr-x 2 harry harry 4096 9月  19 12:09 stocks
6817029 drwxrwxr-x 4 harry harry 4096 9月  19 12:09 travel
6817066 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 公共的
6817090 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 模板
6817074 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 视频
6817033 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 图片
6817047 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 文档
6817024 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 下载
6817059 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 音乐
6817083 drwxr-xr-x 2 harry harry 4096 9月  13 17:19 桌面
```

清单中的每一行列出了一个文件的完整属性, 它包括 8 个字段: inode、文件类型和操作权限、链接数、所属用户, 所属组别、文件大小、最后修改日期 (日期和时间)、文件名。

- inode 记录了文件在文件系统上的入口地址。
- 文件类型和操作权限由 10 个字符组成 (对应 16 位二进制), 其中文件类型占该字段的第一个字符 (16 位二进制的高 4 位), 用字母或 “-” 表示。表3.2 列举了表示文件类型的字母和符号。

表 3.2: 文件类型的表示

符号	意义	常见来源
-	普通文件	由编辑工具创建或应用软件生成
b	块设备文件	由命令 `mknod` 创建
c	字符设备文件	由命令 `mknod` 创建
d	目录	由命令 `mkdir` 创建
l	符号链接	由命令 `ln` 选项 `-s` 创建
p	管道文件	由命令 `mkfifo` 或 `mknod` 创建
s	套接字文件	由系统调用 `bind()` 创建

- 操作权限按用户本人、组别和其他人分成三部分, 每一部分中由读允许、写允许和运行允许组成, 分别用字母 r (**r**ead)、w (**w**rite)、x (e**x**ecute) 表示, 一共 9 个字符, 如果某个权限不具备, 则将相应位置上的字母用 “-” 代替。

  普通文件的 “x” 标记表示它是一个可执行程序。不符合可执行文件格式的, 即使将该属性位强制修改为 “x” 也不能正确执行。目录文件的可执行标记 “x” 表示允许访问 (进入)。其他类型的文件, 可执行属性没有意义。

- 普通文件的硬链接数在使用命令 `ln` 时会自动增加, 目录文件的链接数由一级子目录数决定, 包括目录 "." 和 ".."。
- 对于设备文件, 文件大小字段显示的是主设备号和次设备号。主设备号用于内核识别设备驱动, 次设备号用于驱动程序识别子设备。

改变文件的属性

改变文件属性主要涉及到改变权限和属主。改变文件属性的命令要求用户有对该文件的操作权限, 改变属主时还需要有对目标属主的操作权限。

改变权限 使用 `chmod` (**ch**ange **mod**e) 可根据需要改变文件的读写属性, 一般格式是:

```
$ chmod mode file
或
$ chmod [ugoa]+/-/=[rwxsXt] file
```

`mode` 一般按 "rwx" 的八进制格式表示, 例如 "0644" 即表示文件拥有者可读写、同组和其他人可读的不可执行文件。下面的命令将文件 examples.desktop 的属性改成 "`-rw-------`":

```
$ chmod 600 examples.desktop
```

`chmod` 命令的第二种形式中, u、g、o、a 表示针对用户本人 (**u**ser)、用户组 (**g**roup)、其他人 (**o**thers, 不包括 user 和 group)、所有人 (**a**ll), `+/-/=` 表示为它们添加、去除或者赋予某个权限。例如, 给文件 file 加上 SUID、去掉同组人和其他人的读权限, 使用下面的命令:

```
$ chmod u+s,go-r file
```

使用`chmod` 统一改变一个目录下的所有文件和子目录的权限, 需要使用递归选项 "-R"。

改变用户和组别 `chown` (**ch**ange **own**er) 和 `chgrp` (**ch**ange **gr**ou**p**) 用于重新指定文件所属用户和组别。`chown` 命令中可以同时包含组别:

```
# chown root:harry examples.desktop
```

根据分组策略原则, 改变组别需要超级用户权限。

改变文件的时间 文件系统中记录了文件的三个时间: 最后访问时间、最后修改时间和最后状态改变时间。命令`stat` 可以显示文件的这三个时间信息:

```
$ stat examples.desktop
  文件：examples.desktop
  大小：8980           块：24          IO 块：4096    普通文件
设备：805h/2053d    Inode：16678661     硬链接：1
权限：(0644/-rw-r--r--)  Uid：( 1000/    harry)
Gid：( 1000/    harry)
最近访问：2018-12-24 21:14:50.680213529 +0800
```

```
最近更改：2018-03-16 14:55:44.624701535 +0800
最近改动：2018-03-16 14:55:44.624701535 +0800
创建时间：-
```

使用不带选项的 touch命令时, 会用当前时间覆盖文件的时间戳; 选项 “-t” 用于将特定的时间赋给文件。单独改变最后修改时间或最后访问时间可使用选项 “-m” (**m**odification) 或 “-a” (**a**ccess)。touch 也常常用来创建空文件。

复制文件

cp (**cop**y) 命令将一份文件复制为两份。单个文件复制时, 命令格式是

```
$ cp [选项] 源文件 目标文件
```

如果目标文件已存在, 复制时会将目标文件覆盖, 用复制的源文件替换。为避免误操作, 复制时可以加一个选项 “-i” (**i**nteractive), 表示交互操作方式: 当目标文件存在时, 会提示用户确认覆盖或者放弃。

多个文件复制时, 最后一个参数必须是目录, 复制的结果, 是将前面所列的文件复制到目标目录下。如果包含了目录复制, 应给出选项 “-r” (**r**ecursive), 表示递归操作, 否则目录和目录里的内容都将被跳过。

对文件名具有某种特征的一组文件进行操作时, 为避免输入烦琐和遗漏, Linux 系统使用一种被叫做通配符 (wildcard) 的符号系统来代替。通配符是规则表达式 (regular expression) 在 shell 环境下的延伸。表 3.3 是在文件操作中常用的通配符。

表 3.3: 常用的文件操作通配符

符号	含义	举例
?	匹配任意一个字符	“???” (三个字符的文件名)
*	匹配任意个字符 (. 起头的除外)	“*.c” (以.c 为后缀的文件名)
[ ]	匹配列表中的字符	“*[Aa]” (以字母 A 或 a 结尾的文件名)
		“[A-Z]” (任意大写字母开头的文件)
^、!	不包含	“[^0-9]* (不以数字开头的文件名)
{ }	匹配括号中的列表	“*.c,[a-z]*” (所有 .c 文件和小写字母开头的文件)

例如, 要将 foods 目录下的所有文件和目录 (不包括 foods 本身) 复制到 travel 目录下, 执行的操作应该是:

```
$ cp -ri foods/* travel
```

对于软链接 (softlink) 文件, 复制时可能会有不同的要求: 按文件类型复制或按文件内容复制。选项 “-a” “-P”、“-d” 在复制过程中会保持文件类型属性。

cp 命令仅能复制普通文件和目录文件, 复制其他类型文件时可能得不到预期的结果。例如复制管道文件或者设备文件, 实际上是从管道或设备中读取数据, 再将数据写入目标文件, 也就是说只是创建了一个普通文件, 而不是创建了一个新的管道 (pipe) 文件或者设备文件。并且管道或设备可能是阻塞 (blocked) 的, 复制过程能否进行, 取决于读数据端的阻塞情况。此外, 跨分区复制时, 还要考虑文件系统对文件属性的支持。例如 FAT 文件系

统不支持 inode, 作为目标文件载体时不能创建链接, 作为源文件载体时也不能传递操作权限信息。

移动文件

mv (**mo**ve) 命令有两种功能: 它可以将一个文件搬到另一个文件上, 即文件改名; 或将若干文件搬到一个目录里, 即搬家。与cp 命令类似, 为避免覆盖已有的文件, 选项 “-i” 用于交互提示操作。

删除文件

rm (**rem**ove) 是删除文件的命令。它和桌面环境中将文件移动到回收站的操作结果不同: 回收站只是一个临时存放文件的目录, 移动到那里的文件并不释放硬盘存储空间, 仅当在回收站清除文件后, 该文件才永久消失; 而rm 则是将文件直接永久删除。使用选项 “-i”, 会在每个文件删除之前给出一个提示, 用户以 “y” 加以确认, 或其他任何输入跳过这个删除操作。

删除一个只读文件也会收到警示, 此时回答 “y” 确认, 将执行删除操作, 其他回答都会忽略这个删除命令。

```
$ ls -l file?
-rw-rw-r-- 1 harry harry 0 9月  20 08:44 file1
-r--r--r-- 1 harry harry 0 9月  20 08:44 file2
$ rm file?
rm: 是否删除有写保护的普通空文件 'file2'？ y
```

如果想在操作过程中避免烦琐的确认, 确定删除指定文件, 可使用选项 “-f” 或 “--force”。

选项 “-d” 可以用于删除空目录。“-r” 选项执行目录的递归操作, 即逐层从下到上删除目录和文件。

在终端中, 使用rm 删除的文件不能用正常手段恢复。交互提示选项 “-i” 可以让用户的操作更慎重一些, 但仍然需要用户细心、耐心地操作, 特别是在使用通配符 “*” 操作文件时更应谨慎。

查找文件

locate 根据文件名特征查找文件最快捷的命令是locate, 它从数据库搜索文件名关键字, 迅速地打印出结果。由于依赖数据库, 可以发挥速度的优势, 但同样也是因为数据库, 如果数据库没得到及时更新, 查找文件就会出错。系统一般会设定在夜晚更新数据库, 以免打扰使用者的正常工作。

例如, 查找文件名中包含 “locate” 的文件:

```
$ locate locate
/usr/bin/fallocate
/usr/bin/locate
/usr/bin/mlocate
/usr/bin/msd-locate-pointer
/usr/bin/updatedb.mlocate
```

```
/usr/include/c++/7/bits/allocated_ptr.h
...
```

whereis 如果查找的是可执行程序或者手册页 (manual page), `whereis` 命令也可以很快地找到。它在环境变量 PATH 和 MANPATH 指定的范围内查找。由于范围很小, 查找也很快:

```
$ whereis locate
locate: /usr/bin/locate /usr/share/man/man1/locate.1.gz
```

find 桌面系统的文件管理工具提供的查找手段局限比较大, 一般支持按文件名、日期、大小进行搜索, 更多的文件特性在桌面文件管理器中也不方便体现。前面提到的 `locate` 和 `whereis` 只能根据文件名的线索查找, `find` 命令可以以更加灵活的方式查找文件。表 3.4 列出了部分 `find` 较常用的选项。

表 3.4: find 的部分选项

选项	功能
-maxdepth levels	最大查找目录深度
-mindepth levels	最小查找目录深度
-amin n	最后访问时间在 n 分钟之前的
-anewer file	在文件 file 修改之后访问过的文件
-atime n	最后访问时间在 n*24 小时之前的
-cmin n	文件状态 n 分钟之前变化的
-cnewer file	在文件 file 状态变化之后, 状态发生了变化的文件
-ctime n	文件状态在 n*24 小时之前变化的
-executable	具有可执行属性的文件和目录
-readable	具有读权限的文件
-writeable	具有写权限的文件
-perm mode	权限位, 可以用八进制数字表示, 也可以用 [ugoa] 形式
-gid n	根据文件的组 ID 查找
-group gname	根据组名查找
-uid n	根据用户 ID 查找
-user uname	根据用户名查找
-type c	根据文件类型 (表3.2) 查找, 普通文件类型是 f
-newer file	在文件 file 之后修改过的文件
-samefile file	与文件 file 有相同 inode 的文件
-size n	按文件大小查找, 可使用 k、M、G 等字母单位
-inum n	按 inode 查找文件 (用-samefile 更方便)
-name pattern	按文件名查找, 可以用通配符

... 接上页

选项	功能
-delete	删除找到的文件
-exec cmd	对找到的文件用命令 cmd 处理

查找文件选项中用到的数字 n, 可以用前缀 +/- 表示大于或者小于。

下面的命令, 删除用户 harry 目录下所有大于 20MB 的文件:

```
$ find ~ -size +20M -delete
```

下面的命令, 将当前目录下所有 “.c” 和 “.h” 文件移到 “文档” 目录下 [12]:

```
$ find . -name "*.[ch]" -exec mv {} ~/文档 \;
```

此命令需要做一些解释: 查找命令作用范围是从当前目录 (“.”) 向下, 不限深度; 查找文件名的通配方式 “*.[ch]” 见表 3.3; 选项 “-exec” 表示对找到的文件执行文件移动命令mv, 每一个文件对应一个 “{ }”, 将它移动到用户主目录下的 “文档” 目录; mv 命令会执行多次, 同一命令行的命令用分号 “;” 分隔, 为避免 “;” 被 shell 理解为 find 命令的分隔, 用转义符 “\” 转义。

按多个逻辑关系查找文件时, find 支持逻辑运算操作。表3.5是逻辑操作的简单说明。

表 3.5: 条件查找的逻辑操作

操作符	功能
-and (简写 -a)	操作符两边的条件都成立
-or (简写 -o)	操作符两边条件至少有一个成立
-not (简写 !)	操作符后面的条件反转

例如, 下面的命令查找根目录一层权限为-rw-------的文件或者权限不是drwx------的目录:

```
$ find / -maxdepth 1 \( -type f -perm 600 \) -o \( -type d ! -perm 700 \)
```

在有逻辑运算时, 默认的顺序是从左到右。括号 “( )” 组团的运算用于提高优先级。由于括号在命令行中有特定的含义, 这里用 \将其转义。括号和选项之间要留有空格。

文件比较

比较两个或两组文件之间的差异, 一般可以使用 diff (**diff**erence) 命令。因为是程序操作, 比使用肉眼观察要可靠得多。

- 人工阅读文档时, 有些符号在有些字体显示下容易混淆, 如数字 “0” 和字母 “O”、数字 “1” 和字母 “l”, 有的符号甚至干脆无法分辨, 如空格和制表符。而diff 可以准确地区分不同的符号, 毕竟它是机器, 和人的行为方式完全不同。

[12]本示例仅用于说明 find 的用法。作为批量移动文件的实际操作是有缺陷的, 它没有考虑不同子目录中的同名文件在移动过程中的覆盖问题。

- 在另一些场合下, 差异并没有实际意义, 例如文档中连续的多个空格或空行、制表符和空格；一些程序语言对大小写字母也不介意。此时, diff 的不同选项可以让比较的结果忽略这些差异, 使问题更加集中。

```
Twinkle, twinkle , little star,
   How I wonder what you are.
  Up above the
world so high,
   Like a diamond in the sky.
```

(a) star1.txt

```
Twinkle, twinkle, little star,
How  I  wonder
what you are.

Up  above  the
world so high,
Like  a  diamond  in
the sky.
```

(b) star2.txt

图 3.2: 用于文件比较命令的两段短文

图 3.2 是两篇看上去不一样、但内容完全相同的文档。使用不带选项的 diff 会发现每行都有不同:

```
$ diff star1.txt star2.txt
1,4c1,5
< Twinkle, twinkle , little star,
<    How I wonder what you are.
<   Up above the  world so high,
<    Like a diamond in the sky.
---
> Twinkle, twinkle, little star,
> How  I  wonder   what you are.
>
> Up  above  the  world so high,
> Like  a  diamond  in  the sky.
```

而使用 “-B” (忽略空行) 和 “-w” (忽略空格) 选项后, 两个文件的比较结果完全相同。

使用 diff 进行文件或目录比较的命令格式是:

```
$ diff [options] FILE1 FILE2
```

FILE1 和FILE2 可以是文件或目录。在进行目录比较时, |diff| 将两个目录下的同名文件进行比较。表 3.6 列出了diff 的常用选项。

表 3.6: diff 的常用选项

选项	功能
-q	给出相同或不同的简短结论
-c,-C NUM	分别输出差异处前后 NUM 行内容 (缺省是 3)
-u,-U NUM	合并输出差异处前后 NUM 行内容 (缺省是 3)
-y	并排两列输出
-t	输出时将制表符扩展成空格
--tabsize	制表符对应的空格数 (缺省是 8)
-r	目录递归 (用于目录比较)
-N	将空缺文件作为空文件处理 (用于目录比较)
-x PATTERN	排除文件名匹配 PATTERN 的文件 (用于目录比较)
-X FILE	排除文件名匹配文件 FILE 中的文件 (用于目录比较)
-i	忽略字母大小写
-E	忽略制表符扩展
-Z	忽略尾部空格
-b	忽略多个连续空格的差异
-B	忽略空行
-w	忽略所有空格
-a	将文件作为文本文件处理 (比较二进制文件)
--color[=WHEN]	彩色输出 (never、always 或者 auto)

`diff` 还有两个长选项: –ignore-file-name-case 和 –no-ignore-file-name-case。缺省方式, `diff` 对文件名字母大小写敏感。有些文件系统不区分文件名大小写, 这两个选项可以决定是否忽略文件名大小写的差别。

`diff` 在比较二进制文件时, 通常是直接给出两个文件相同或者不同的结论, 而不指出具体差别。当使用选项 “-a” 时, 它也完全可以给出准确的结果, 但二进制的比较结果对修改文件没有太大帮助, 如果要修改, 只能是全文替换。因此它一般不用于比较两个二进制文件。

`diff` 在实际使用时, 比较结果通常会输出到一个文件, 这个文件又叫补丁文件。Linux 系统提供了专门的工具 `patch`, 用补丁文件升级软件。除了作为普通的文件比较以外, 它也是数据库备份、系统升级、版本控制的重要基础工具。

文件处理

grep 如果需要在一组文件中查找特定的内容, `grep` (**g**lobal **r**egular **e**xpression **p**rint) 是一个非常有用的工具。缺省方式下, 它直接打印满足匹配条件的文件名 (如果以通配符作为文件名参数的话) 和特征文本所在行:

```
$ grep 'harry' /etc/passwd
```

```
harry:x:1000:1000:Harry Potter,,,:/home/harry:/bin/bash
```

在彩色终端上, grep 通过 “--color” 选项支持匹配文字彩色显示, 很多系统已经将它默认加上这个选项做了别名。

流编辑 除了使用文件编辑器直接编辑文件以外, Linux 提供了一些程序化的编辑工具, 它们无需人工操作, 而是以程序的方式编辑文件。sed (**s**tream **ed**itor) 是其中最常用的一个, 在完成差不多功能的情况下, 它比 awk 和 Perl 要小得多。流编辑器还有一个文本编辑器无可替代的功能: 它可以编辑 shell 变量。

表 3.7: sed 的常用选项

选项	功能
-e script	执行脚本 script
-f script-file	执行脚本文件 script-file 中的命令
-i[suffix]	直接在文件内编辑 (无此选项时, 默认输出到标准输出设备)。
	如提供 suffix, 则以 suffix 作为备份文件的后缀
-n	安静模式, 通常与 p 命令结合, 仅打印特征行信息
-E, -r	使用扩展型规则表达式

表 3.7 列出了使用 sed 时的一些常用选项。缺省方式下, 编辑命令脚本写在命令行的引号中, 无需选项 “-e” 指定, 选项明确时, 编辑命令也可以不用引号。

作为流编辑器, sed 每次只从文件读入一行, 对该行进行指定的处理、输出, 接着读入下一行, 整个文件像流水一样被逐行处理然后逐行输出。处理时, 把当前处理的行存储在临时缓冲区中, 该缓冲区称为 “模式空间”; 处理完成后, 把缓冲区的内容送往 “保留空间”, 接着处理下一行, 直到整个文件处理完毕。

表 3.8是 sed 常用的编辑命令, 它们与编辑器 vim 使用的编辑命令有很大的重叠。sed 的编辑命令支持规则表达式。

表 3.8: sed 的常用编辑命令

命令	功能	
{ }	命令块	
:	标号	
b	跳转	
t	条件跳转	
T	条件跳转	
=	打印当前行号	
#	注释	
q	退出	
p	打印	打印当前模式空间

... 接上页

命令	功能	
P	打印	打印当前模式空间, 直到第一个换行位置
n,N	下一行	将下一行送入模式空间
h	空间复制	模式空间复制/追加到保留空间
g	空间复制	保留空间复制/追加到模式空间
l	清晰打印	明示不可见字符 (换行符\r, 行尾 $, 等)
c	行替换	1,10cnew text, 用新的字符串替换模式空间
s	串替换	1,10s/abc/ABC/, 用 ABC 替换 abc
d	删除行	/this/d, 删除带有 “this” 的行
D	删除行	删除模式空间, 直到第一个新行位置
i	前插入行	5iNew Line, 在第 5 行前面插入新行
a	后插入行	5aNew Line, 在第 5 行之后插入新行
r	插入文件	5r hello.h, 在第 5 行之后插入文件 hello.h
w	写文件	3,8w newfile, 将第 3–8 行写入文件 newfile
x	交换	模式空间与保留空间互换
y	映射转换	y/source/dest

编辑文件时, 模式空间可以由行号或行号范围指定, 也可以用`/pattern/` 匹配方式指定。例如, 下面的命令打印文件 hello.c 中含有 `#include <...>` 的行:

```
$ sed -n '/#include <\(.\+\)>/p' hello.c
```

下面的命令将文件 hello.c 中的小写字母移位, 即 a→b,b→c,...,z→a, 起到一个简单的加密作用:

```
$ sed 'y/abcdefghijklmnopqrstuvwxyz/bcdefghijklmnopqrstuvwxyza/' hello.c
```

使用`sed` 一次执行多个编辑项时, `sed` 命令允许重复使用 “-e” 选项, 或者用分号 (;) 分隔各个编辑项。由于分号在命令行中有特殊用途, 因此后一种方法应将编辑命令置于引号中。

`sed` 处理后默认输出到标准输出设备。通常有两种方式保存处理结果: 输出重定向或者使用 “-i” 选项直接编辑修改文件。

压缩和解压

Linux 系统主要使用三种开源压缩算法, 它们被列在表 3.9 中以便对比。

文件压缩后, 通常还需要一个规范的格式进行包装 (有的压缩命令允许以裸数据保存), 表 3.9 的格式一栏所列的文件后缀就是 Linux 系统习惯的包装命名方式。缺省压缩方式, 输出文件名就是在原文件后面加上格式后缀。

每个压缩命令都可以有一个 0 到 9 之间的数字作为选项, 用来指定压缩率指标, 数字越大, 压缩率越高, 同时意味着算法耗时也更多。

```
$ gzip -9 file              # 将 file 以最大压缩率压缩成 file.gz
$ gunzip file.gz            # 将 file.gz 解压
```

表 3.9: 常见压缩格式

格式	算法	压缩命令	解压命令	压缩效率	算法复杂度
gz	DEFLATE	gzip	gunzip	低	低
bz2	Burrows-Wheeler	bzip2	bunzip2	中	高
xz	LZMA2	xz	unxz	高	中
lzma	LZMA	lzma	unlzma	高	中

本节涉及到的压缩算法都属于无损压缩, 压缩后的文件可以节省磁盘占用空间, 通过相应的解压算法可以完美地恢复原来的数据。在图像、音视频中采用的压缩算法常常是有损压缩, 它利用人的感知特性, 有目标地去除一些非敏感特征, 从而可以在保持信息基本完整的情况下获得更大的压缩率。例如, 将一幅 BMP 格式的图像转成 PNG 格式时, 使用的是 DEFLATE 算法, 与 gz 的压缩效率相当, 属于无损压缩; 而转成 JPG 图像格式时, 可以借牺牲图像的部分信息以换取更高的压缩率, 但无法再还原成与原图一样的图像。

打包和拆包

经常我们需要将一组文件打包合并成一个文件, 便于携带和传输。为了减小体积, 还要进行压缩。有一些命令专门完成这项工作。`tar` (**t**ape **ar**chive) 是一个常用的工具。它可以通过不同的选项选择压缩方式, 打包和压缩一次完成。

`tar` 命令的传统方式选项前面没有 "-"。选项必须包含 A、c、d、r、t、u、x 中的一个。例如, 将目录 "桌面" 下的所有文件以 bzip2 格式压缩并打包成文件 desktop.tar.bz2:

```
$ tar cjf  desktop.tar.bz2 桌面/
```

拆包时, `tar` 能自动识别压缩文件的格式, 因此不需要用参数指定。错误的参数反而会导致命令执行失败。下面是解压命令的例子:

```
$ tar xvf desktop.tar.bz2 -C 下载
```

上面的命令表示在 "下载" 目录下解包文件 desktop.tar.bz2, 解包过程中会显示压缩包里的文件名。

`zip` 是 Linux 系统中不太常用的一种压缩包格式, 它使用 `zip/unzip` 一对命令。例如, 下面的命令将目录 travel 打包后生成一个 travel.zip 文件, 选项 "-r" 表示子目录递归:

```
$ zip -r travel.zip travel
```

zip3.0 版之后具有压缩分包功能。下面的例子将 foo 目录下的文件以 2MB 为单位生成若干压缩包, 文件名后缀依次是 .z01、 .z02、...、 .zip:

```
$ zip -s 2m -r split.zip foo
```

经 `zip` 压缩打包的文件, 可使用 `unzip` 列表或解压拆包。常用的选项有: "-l"—列表; "-x"—解压; "-d"—指定解压目录; "-t"—检查校验。上面生成的 travel.zip 经下面的命令处理后, 会将文件在 "桌面" 目录里展开:

```
$ unzip travel.zip -d 桌面
```

3.2.7 手册页

大多数 Linux 命令的用法都被写成了标准的帮助文档, 这种帮助文档被编辑成手册页 (**man**ual page)。手册页比命令本身的 “--help” 选项打印的信息更全面。例如, 如果不知道 `diff` 命令怎么用, 只需要`man diff`, 就可以查到详细的使用说明:

```
DIFF(1)                          User Commands
DIFF(1)

NAME
       diff - compare files line by line

SYNOPSIS
       diff [OPTION]... FILES

DESCRIPTION
       Compare FILES line by line.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       --normal
              output a normal diff (the default)

       -q, --brief
              report only when files differ
...
```

这套说明书共有 8 卷, 按表 3.10 的内容分类。

表 3.10: 帮助文档说明书分类

卷号	内容
1	可执行程序或 shell 命令
2	编程中使用的系统调用函数
3	编程中使用的库函数
4	设备文件 (通常位于 /dev 目录)
5	文件格式和规范 (通常是各种命令用到的配置文件)
6	游戏
7	杂项 (包括宏包和规范, 如 man(7), groff(7))
8	系统管理命令 (通常只提供给 root 用户)

每篇帮助文档包括概述 (SYNOPSIS)、详述 (DESCRIPTION)、选项 (OPTIONS)、遵循标准 (CONFIRMING TO)、作者 (AUTHORS)、参见 (SEE ALSO) 等小节, 很多文档中包含了命令的使用范例。函数帮助中 (卷 2 和卷 3) 有函数格式原型、参数和返回值的说明, 有些还给出了完整的使用例子。这给 C 语言编程带来了很大方便, 著名的文本编辑器 `vim` 可以直接在编辑界面中通过快捷键打开所关注的函数的帮助文档。

有的命令可能会同时属于说明书的不同卷, 例如 `kill`, 既是一条命令, 也是一个系统调用函数。查阅文档时, 首先会在第 1 卷查到它是一个命令, 帮助文档最后会显示:

```
SEE ALSO
       kill(2), killall(1), nice(1), pkill(1), renice(1), signal(7), skill(1)
```

`kill(2)` 表示它还存在于说明书第 2 卷。打开靠后的卷的帮助, 需要在 `man` 命令的参数名后面加上卷号的后缀:

```
$ man kill.2
```

有时候, 用户想做一件事情, 但不知道准确的命令, 只是大致知道一些关键词, 此时可以使用 "-k" 选项让 `man` 在说明书中模糊查找, 例如:

```
$ man -k printf
asprintf (3)         - print to allocated string
dprintf (3)          - formatted output conversion
fprintf (3)          - formatted output conversion
fwprintf (3)         - formatted wide-character output conversion
printf (1)           - format and print data
printf (3)           - formatted output conversion
snprintf (3)         - formatted output conversion
sprintf (3)          - formatted output conversion
swprintf (3)         - formatted wide-character output conversion
vasprintf (3)        - print to allocated string
vdprintf (3)         - formatted output conversion
vfprintf (3)         - formatted output conversion
vfwprintf (3)        - formatted wide-character output conversion
vprintf (3)          - formatted output conversion
vsnprintf (3)        - formatted output conversion
vsprintf (3)         - formatted output conversion
vswprintf (3)        - formatted wide-character output conversion
vwprintf (3)         - formatted wide-character output conversion
wprintf (3)          - formatted wide-character output conversion
```

根据查到的列表清单, 用户可以根据这个线索再继续使用 `man` 详细查阅。`man -k` 也可以用另一个命令 `apropos` 代替。

手册页存放在系统 /usr/share/man 目录 (三级目录结构是 /usr/local/share/man), 以子目录 man1~man8 分卷管理。每个文件就是一个 `troff` (**t**ypesetter **roff**, 一种格式化文档) 格式的帮助文档。为了节省空间, 一些文件会采用压缩格式存放, `man` 命令会自动解压。

3.2.8 进程管理

进程状态

显示进程状态的命令是 ps (**p**rocess **s**tatus)。选项 “ax” 列出当前系统中所有进程的状态:

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     1:45 /sbin/init splash
    2 ?        S      0:00 [kthreadd]
    4 ?        S<     0:00 [kworker/0:0H]
  ...
20755 ?        Ssl    0:00 /usr/lib/evince/evinced
21326 ?        Ss     0:00 /lib/systemd/systemd --user
21327 ?        S      0:00 (sd-pam)
23376 pts/0    Sl     0:47 evince LinuxBasis.pdf
23885 pts/2    S      0:01 bash
25034 pts/0    S+     3:20 vim text/commandline.tex
  ...
```

列表的第一栏表示进程号 (**P**rocess **ID**)。Linux 系统在创建进程时会为每个进程开一个进程槽, 并用一个正整数作为进程号的 ID, 作为管理的入口。进程号从小到大顺序安排。1 号进程是系统的初始化进程。进程号达到最大值后会绕回来再从小的空闲数字中挑选。第二栏表示进程启动的终端 (TTY 源自早期的电传打字机 **t**ele**ty**pewriter), 初始化程序和图形桌面启动的程序没有 TTY。STAT 表示进程当前的状态。一个进程可能处于四种状态之一: 运行、睡眠 (等待)、停止、僵尸 (死亡), 有些状态还可以进一步细分。表 3.11 是用于表示进程状态字母的含义。TIME 一栏是进程消耗 CPU 的累积时间。COMMAND 列出了进程的完整命令, 有的命令很长, 超出了打印的篇幅, 上面的清单做了删减。

有些进程之间会有父子继承关系, `pstree` 可以打印进程的关系图。

还有一个比较常用的显示进程状态的命令 `top`, 它以一定时间间隔动态显示系统的进程状态, 并按指定参数对这些进程进行排序显示。

改变进程状态

用户可以使用 kill、pkill 或 killall 命令终止一个进程。`kill` 使用进程号作为操作对象, `pkill` 或 `killall` 使用进程名 (程序名) 作为操作对象:

```
$ kill 23376                # 以进程号为命令参数
$ pkill evince              # 以程序名为命令参数
```

`kill` 命令并不像它的名字那样凶险。它本意是向进程发送信号, 而信号是进程间通信的一种机制。但缺省的方式是发送 SIGTERM, 它一般导致进程终止。表 3.12 是用选项 “-L” 列出的常用信号值, 它们与定义在 /usr/include/signal.h 的符号有可能不一样。信号值或者信号名都可以作为 `kill` 的选项, 表示向指定进程发送指定信号。

有的进程会忽略 SIGTERM 信号, 这时如果想结束那个进程就必须用 `kill -9 PID`。信号并非都是导致进程停止的, 例如下面的命令就可以让一个进程暂停、继续:

表 3.11: 进程状态 STAT

字母	含义
D	不可中断睡眠
S	可中断睡眠
R	运行状态
T	被任务控制信号停止
t	被 debugger 停止
Z	僵尸
<	高优先级 (优先级不能被其他用户改变)
N	低优先级 (优先级可以被其他用户改变)
L	内存中有页面被锁定
s	一组会话的组头
l	多线程的进程
+	在前台进程组中

表 3.12: kill 命令常用信号描述

信号名	信号值	说明
SIGHUP	1	挂起
SIGINT	2	中断 (终端 CTRL+C)
SIGQUIT	3	进程退出
SIGABRT	6	异常中止
SIGKILL	9	杀死进程 (不可阻塞)
SIGUSR1	10	用户定义的信号 1
SIGSEGV	11	段错误 (存储器访问非法)
SIGUSR2	12	用户定义的信号 2
SIGPIPE	13	管道错误
SIGALRM	14	闹钟
SIGTERM	15	进程中止
SIGCHLD	17	子进程状态改变
SIGCONT	18	进程继续
SIGSTOP	19	进程停止 (不可阻塞)

```
$ kill -STOP 23376
$ kill -CONT 23376
```

同样的程序在系统中可以创建多个进程, 例如用 gnome-terminal 打开多个终端, 多个用户可能同时都在使用 gnome-terminal。以进程名为控制对象的命令 `pkill` 或 `killall`

会将信号发送到多个进程。它可以通过特定的选项对进程进行筛选, 包括对指定用户、指定组、在指定时间之前或之后的进程操作。

前台与后台

在终端启动图形界面程序时, 为了避免对当前终端的占用, 习惯上在命令行的尾部加一个 “&” 符号, 该程序即以后台 (background) 方式运行, 终端仍可以继续其他的命令操作:

```
$ gedit hello.c &
```

shell 的两个内部命令 `fg`、`bg`用于命令的前、后台切换。上面这个后台命令可以用`fg` 将其拖回前台。

当一个终端运行了多个后台命令时, 另一个 shell 内部命令 `jobs` 可以列出当前的后台命令和序号。序号可作为`fg` 的参数, 用于对指定命令的操作。

一个终端启动的进程接收到停止信号 (SIGSTOP) 时, 或在终端用快捷键 CTRL+Z 操作时, 该进程进入停止状态, 也会将终端让出。这时候在这个终端上用 `fg` 或 `bg` 可以继续刚才的进程。

3.2.9 I/O 重定向与管道

I/O 重定向

标准 I/O 设备的重定向 (redirection) 是 Linux 文件操作的一种特性。在应用层面, 所有操作都将落实到文件读写上。文件是信息的一种载体, 重定向意味着改变信息的来源和去向。

标准设备文件

表 3.13: 标准 I/O 文件

设备	文件描述符 (值)	FILE 文件指针
标准输入	STDIN_FILENO (0)	stdin
标准输出	STDOUT_FILENO (1)	stdout
标准错误输出	STDERR_FILENO (2)	stderr

重定向的基础是标准 I/O 设备。所有进程都会默认打开三个文件: 标准输入 (standard input, 0 号设备)、标准输出 (standard output, 1 号设备) 和标准错误输出 (standard error output, 2 号设备)。使用终端时, 键盘是标准输入, 显示器承担标准输出和标准错误输出两项功能。它们对应的文件描述符 (file descriptor) 和流文件结构指针见表 3.13。标准设备的文件描述符定义在 unistd.h, FILE 文件指针结构定义在 stdio.h。

标准输出和标准错误输出

键盘作为标准输入设备比较容易理解。一个程序把显示器作为输出设备, 可能出于两种需求: 一、输出程序正常的运行结果; 二、输出程序的运行状态。为了区分这两种性质

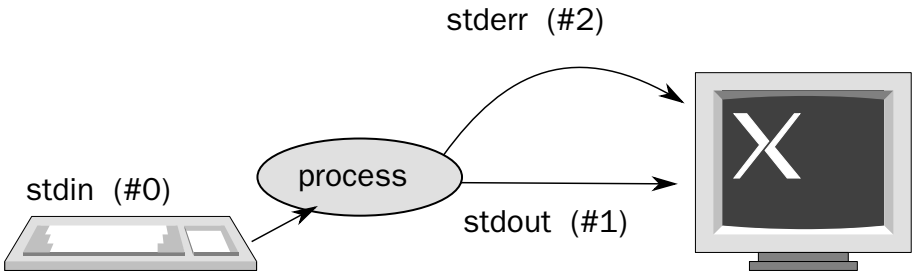


图 3.3: 与进程相关的标准 I/O 设备

不同的信息, 系统将前者送往标准输出设备, 后者送往标准错误设备 (建议不要单纯从字面上理解 “错误” 一词)。

下面是以普通用户身份执行 `ls` 命令的两个不同结果:

```
$ ls /home
harry sally
$ ls /root
ls: 无法打开目录 '/root/': 权限不够
```

上面两句看似都是打印在显示器上, 但实际上是有差别的, 前者来自标准输出, 后者来自标准错误输出。清单3.1给出了一个使用标准设备的简单例子。

清单 3.1: stdio.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[])
{
    char msg[256];

    fgets(msg, 256, stdin);

    if (strlen(msg) <= 1) {
        fprintf(stderr, "No input message\n");
        return EXIT_FAILURE;
    }
    fprintf(stderr, "Input %ld chars\n", strlen(msg));
    fprintf(stdout, "Input message: %s", msg);

    return EXIT_SUCCESS;
}
```

程序运行时从标准输入设备读取一行信息。如果有信息输入, 则将输入字符数输出到标准错误设备, 并将信息连同提示打印在标准输出设备, 如果是空行, 则向标准错误输出设备打印提示文字。下面是程序运行的结果:

```
$ ./stdio
hello, world        # 这一行从键盘输入
Input 13 chars
Input message: hello, world
$ ./stdio > out
hello, world        # 这一行从键盘输入
Input 13 chars
$ cat out
hello, world
```

上面的例子中, `> out` 是标准输出重定向操作, 它将原应送往标准输出设备的信息转送到文件 “out” 中。

I/O 重定向

表3.14是重定向操作符。

表 3.14: 重定向操作符

操作符	含义
> 文件	输出重定向
>> 文件	追加式输出重定向
< 文件	输入重定向
<< 分界符	用分界符控制输入重定向的结束
<<< string	重定向输入字符串

“>” 输出重定向的文件如果不存在, 则创建它, 如果存在, 则会把原来的内容覆盖。追加式输出重定向 “>>” 的定向文件如果存在, 会把新的内容追加到该文件之后, 不会覆盖原来的内容。

仍然以程序3.1为例, 使用输入重定向:

```
$ ./stdio < stdio.c
Input 19 chars
Input message: #include <stdio.h>
```

它将文件 stdio.c 的第一行作为输入, 替代标准输入设备 (键盘) 输入的内容。

输入重定向 “<<” 使用指定的分界符结束输入:

```
$ cat > grape.txt << EOF
> 吃葡萄不吐葡萄皮
> 不吃葡萄倒吐葡萄皮
```

```
> EOF
$
```

上面的命令同时使用了输出重定向, 操作结束后新建的文件 grape.txt 中包含了这个绕口令的内容。命令行环境下, 人工以键盘作为标准输入设备进行输入操作时, 也可以用 CTRL+D 组合键结束输入。但在程序自动运行过程中, 人工无法进行键盘干预, 这种重定向方式就可以派上用场了。

输入重定向 “<” 和 “<<<” 不同, 前者输入重定向来自文件, 后者输入重定向直接是字符串, 下面的例子应可以帮助理解：

```
$ cat <<< stdio.c
stdio.c
```

由于标准输出存在两个设备, 通过文件描述符加以区分: “1>” 是标准输出重定向, “2>” 是标准错误输出重定向 (中间不留空格, 否则数字会被理解为命令的参数)。如果是标准输出重定向, 数字 “1” 可以省略。

仍然用清单 3.1 的程序例子:

```
$ ./stdio < stdio.c > out.txt 2> err.txt
$ cat out.txt
Input message: #include <stdio.h>
$ cat err.txt
Input 19 chars
```

两个标准输出设备之间也可以互相重定向: “2>&1” 将 2 号设备重定向到 1 号设备, “1>&2” 将 1 号设备重定向到 2 号设备 (中间不留空格, 否则 & 会被理解为后台操作符)。下面的命令从文件 stdio.c 输入, 将标准输出结果送往 out.txt, 且由于标准错误输出也被重定向到标准输出, 因此文件 out.txt 最后保存的结果是二者的叠加:

```
$ ./stdio < stdio.c > out.txt 2>&1
```

使用重定向技术时, /dev/null 是一个非常有用的设备, 它像一个能吞噬一切的无底洞: 当一个程序的输出是我们不想要的内容时, 我们不希望它干扰屏幕的显示, 就可以把它重定向到这个设备上:

```
$ cat /etc/* > etc-files 2>/dev/null
```

上面的命令将 /etc 目录下所有文件打印到 etc-files 文件中, 而目录或无访问权限的错误提示被丢弃到 /dev/null 里。当然也可以把无用的信息定向到一个废弃的文件中, 不过磁盘文件操作比较慢, 而且占用空间。

重定向只适用于标准 I/O 设备, 两个普通文件不能直接进行重定向操作。Linux 的系统命令多数都是以标准 I/O 设备为操作对象的, 由此使得命令行中各种命令组合相当灵活。

管道

命令行中, 管道 (pipe) 是连接一组先后执行的命令输入/输出数据的通道, 这里的命令又被称作过滤器 (filter), 连接过滤器的操作符是 |。下面的例子有助于理解这一概念:

```
$ cat /etc/passwd | head -n 15 | tail -n 5
```

表 3.15: 常用命令一览

命令	功能	命令	功能
cat	(串联) 打印文件	cd	切换工作目录
chmod	修改文件属性	cp	复制文件或目录
df	显示文件系统占用率	du	显示磁盘用量
echo	显示文字信息	find	查找文件
grep	搜索匹配文本	kill	改变进程状态
ln	创建链接	ls	文件清单列表
mkdir	创建目录	mv	移动 (重命名) 文件或目录
more	分屏浏览文件内容	ps	显示进程
rm	删除文件	rmdir	删除目录
sed	流编辑	tar	打包压缩/解压

cat 命令将文件 /etc/passwd 的内容送到标准输出设备, 用 head 命令过滤出前 15 行, 继续送到标准输出设备, 再用命令 tail 过滤出后 5 行, 最后仍在标准输出设备上输出, 最后的结果就是打印文件 /etc/passwd 的第 11 行到第 15 行 [13]。

注意管道与重定向的不同: 重定向连接的是命令和文件, 管道连接的是一个命令的标准输出和另一个命令的标准输入。上面的管道操作命令, 与下面的重定向操作等效, 但显然效率要高很多:

```
$ cat /etc/passwd > file1
$ head -n 15 < file1 > file2
$ tail -n 5 < file2
$ rm file1 file2
```

3.2.10 Linux 常用命令小结

以上介绍了一些常用的 Linux 命令。虽然命令行方式需要记忆很多操作命令, 但它对系统资源要求极低, 并且在很多场合有图形方式难以企及的高效。表 3.15 列出了使用频度最高的 20 条命令, 仅供参考。

在通用计算机上, 命令行不是使用计算机的必须手段, 它只是开发人员或者系统管理人员提高效率的一种选择。面向大众的软件, 在条件允许的情况下能够提供友好的交互界面, 才是开发人员努力的目标。

3.3 Linux 环境的软件开发

相比于消费类的计算机产品, 安装 Linux 操作系统的个人计算机更适合当作开发工具使用——尽管 Linux 的桌面环境完全满足一般的消费型用户的需求。Linux 的软件仓库中包含众多的编程语言开发工具, 开发人员可以根据自己的需要选择安装其中的一部分。

[13]此功能还可以用 sed 实现: `sed -n 11,15p /etc/passwd`

虽然 Linux 支持多种编程语言, C 语言仍是 Linux 系统最主要的编程语言之一, 它也是目前处理工程类问题最普适的编程语言。Linux 内核中, 除了极少量的初始代码是与处理器架构相关的汇编语言 (assembly language) 以外, 其他的代码全部都是由 C 语言完成, 应用软件也是以 C 语言为主。

3.3.1 编译工具

高级语言需要通过专门的软件, 将其转换成机器语言, 才能被计算机执行。这个转换软件统称编译工具 (compiler)。将高级语言编译成机器语言, 大致需要经过预处理、编译、汇编和链接四个过程。

预处理 (preprocessing) 在这一过程中, 将 `#include` 的文件嵌入、进行宏替换、以及其他宏语句的处理。

编译 (compiling) 将预处理后生成的文件进行语法分析、翻译, 转换成汇编语言。如果有文件输出, 生成后缀为 ".s" 的汇编语言文件。

汇编 (assmbling) 将汇编语言翻译成机器语言, 生成目标文件, 通常文件名后缀是 ".o"。目标文件中是可执行的代码, 但不具备可执行文件的格式, 因为操作系统不知道如何将它装入内存, 也不知道如何给它分配地址。

链接 (linking) 根据要求将若干目标文件组装在一起, 填入正确的外部地址, 再加上工具链中的启动文件, 共同组合成一个可执行文件。

正常开发软件过程中, 如果不是有意为之, 生成的中间文件可能不会保留, 预处理的输出、汇编语言文件和目标文件.o 属于这类情况。并非所有软件的开发过程都要经过上面四个步骤, 例如开发静态库只需要前三步, 最后用库管理工具做个包装就行了; 如果本身就是汇编语言源程序, 汇编转换工作也是不必要的。

目前, GCC 是 Linux 系统编译工具的首选[14]。GCC 最初是 C 语言的编译器, 目前已支持包括 C++、FORTRAN、JAVA 等在内的多种编程语言。当我们提到 GCC 时, 狭义的是指 gcc 一条命令, 而广义的则是指一组编译工具的集合, 它包含下面几部分内容:

1. 预处理器 cpp、C 编译器 gcc、C++ 编译器 g++, FORTRAN 编译器 gfortran, 等等。
2. 二进制代码处理工具: 汇编器 as、链接器 ld、库管理工具 ar、ranlib、代码转换 objdump 等等。编译过程中如果有链接需求, gcc 或 g++ 会自动调用链接器 ld。通常 C++ 的源程序用 g++ 链接, 因为默认的 gcc 不链接 C++ 库 stdc++。
3. 调试器 gdb, 用于诊断程序的错误。
4. C 语言标准库 glibc, 由若干库文件和启动文件组成。
5. 应用程序头文件。

gcc 和 g++ 的帮助文档 (manual page) 有两万多行, 涉及的选项有数百个, 其中大多数选项对于普通开发者来说都不会直接用到。除了在解释 C++ 程序时需要使用 g++ 命令外, 二者的选项大部分是重合的。表 3.16 列出了 gcc 最常用的一些选项。

[14] 另一款著名的开源编译器是 LLVM, 它最初的名称来自 **L**ow **L**evel **V**irtual **M**achine。LLVM 目前的开发与通常认为的 "虚拟机" 概念关系不大。为避免歧义, 已不再引用这个全称。与 LLVM 配合的前端是 Clang。

表 3.16: gcc 的常用选项

选项	说明
-o file	输出文件
-c	编译, 生成 .o 目标文件
-E	预处理
-S	输出汇编语言程序 .s
-g	为 gdb 生成源码级调试信息
-s	去除可执行程序中的所有符号表和定位信息
-p, -pg	增加生成供 gprof 剖析的代码
-DNAME	定义宏 NAME=1
-UNAME	取消 NAME 的宏定义
-I path	指定额外的头文件搜索路径 path
-L path	指定额外的库文件搜索路径 path
-l library	链接指定的库, 库名不含前缀 lib 和后缀 .a/.so
-O	优化选项。从-O0 到 -O3, 数值越大, 优化程度越高 (程序运行越快)。缺省方式是-O0 (编译最少耗时)。-Os 以优化代码大小为目标。
-shared	生成共享目标文件 (或称动态库)
-static	使用静态链接方式, 禁用共享连接
-pthread	链接 POSIX 线程库, 等效于 -lpthread
-fPIC	生成位置独立代码。一些共享库需要此选项
-Wa,options	options 是传递给汇编器的选项, 多个选项之间用逗号分隔
-Wl,options	options 是传递给链接器的选项, 多个选项之间用逗号分隔
-W	打印警告信息
-w	不打印任何警告信息

3.3.2 软件开发过程

下面通过一个例子展示使用 GCC 进行软件开发的过程。

源程序的准备

编写一个计算斐波那契序列[15]的程序。要求在命令行中输入序列项数, 程序打印计算结果。

通常一个项目会由若干个源文件组成。功能相近的函数以及相关的数据结构放在一个文件或一组文件里, 形成一个模块。这种模块化的设计方式可以给软件带来更好的可维护

[15] 形如 1,1,2,3,5,8,13... 这样的序列被称作斐波那契序列, 从第三项开始, 每一项是前两项之和。这个序列最早被意大利数学家列奥那多 · 斐波那契 (1175 年–1250 年) 研究, 由此得名。

性, 也更有利于协同开发。清单 3.2–3.4 是完成本任务的三个文件, 采用递归算法。算法直接来自公式

$$F(n) = \begin{cases} 1 & \text{when } n < 2 \\ F(n-1) + F(n-2) & \text{when } n \geq 2 \end{cases}$$

算法复杂度是 $O(2^n)$。

清单 3.2: 计算斐波那契序列主程序 main.c

```
#include <stdlib.h>
#include <stdio.h>
#include "fibo.h"

int main (int argc, char *argv[])
{
    long ret;
    int n;

    if (argc < 2) {
        perror("missing index.\n");
        return -1;
    }
    n = atoi(argv[1]);
    ret = fibo(n);
    printf("Fibo(%d) is %ld\n", n, ret);

    return EXIT_SUCCESS;
}
```

清单 3.3: 计算斐波那契序列子程序 fibo.c

```
long fibo(int n)
{
    if (n < 2)
        return 1L;
    else
        return fibo(n-1) + fibo(n-2);
}
```

清单 3.4: 计算斐波那契序列头文件 fibo.h

```
#ifndef _FIBO_H
#define _FIBO_H
```

```
long fibo(int n);
#endif
```

对于编写源程序的几点说明:

1. **#include** 可以用<...> 和 "..." 两种形式包含 .h 文件。前者包含的 .h 文件在 gcc 的搜索目录里 (通常是 /usr/include 或是 -I 选项指定的目录, 不含这些目录的子目录); 后者包含的 .h 文件先在当前目录下查找, 如果当前目录下没有, 再到 gcc 的搜索目录里查找。建议根据规则使用不同的包含形式, 因为 gcc 可以使用特定的选项区分这两种情形, 并指导 GNU Make 建立依赖关系。
2. .h 文件中通常只包含常量、结构体、宏、函数原型这些声明, 过程代码不建议写在 .h 文件里。gcc 命令行的参数中不应该出现 .h 文件。 .h 文件里一般也不直接定义变量。如果有全局变量, 可在 .c 文件中定义, 并在 .h 文件中使用 **extern** 声明。
3. 为避免 .h 文件重复包含, 常常使用清单 3.4 这样的宏保护结构定义一个专属符号, 或者使用预处理命令 **#pragma once**, 不过后者需要编译器支持。
4. 程序结构尽可能简单明确, 函数调用层次或循环层次不宜过多, 不要追求编程技巧。语义不明的代码容易给编译器带来困惑, 也容易出现错误。如下面的代码就是语义不明确的:

```
    if (a)
        if (b)
            foo ();
        else
            bar ();
```

else 可以对应第一个 **if**, 也可以对应第二个 **if**。这种情况, 最好用 **{ }** 将语句块明确下来。

3.3.3 编译和运行

编译过程如下:

```
$ gcc -c main.c
$ gcc -c fibo.c
$ gcc -o fibo main.o fibo.o
```

gcc 的 “-c” 选项仅将源程序编译成目标程序而不继续生成可执行程序, 默认生成的文件名后缀用 .o 替换。不能一次性将源程序编译成可执行程序时, 一般需要使用 “-c” 选项。不使用 “-E”、“-S”、“-c” 选项时, gcc 会调用链接器尝试生成可执行程序, “-o” 用于指明生成的文件名。不使用 “-o” 指定时, 生成的可执行文件被命名为 a.out, 但这种方式一般不采用, 因为文件名缺乏个性, 不具备识别意义。

运行:

```
$ ./fibo 20
Fibo(20) is 10946
```

对于简单的项目, 可以将多个源程序一次性编译成可执行程序, 而不需要生成中间的目标文件。对于分阶段编译的目标文件, gcc 会自动识别合法文件格式, 并进行正确处理。如:

```
$ gcc -o fibo main.c fibo.c
```

或

```
$ gcc -o fibo main.c fibo.o
```

将模块编译成库

库可以实现功能共享, 简化程序设计。Linux 系统中有静态库 (statically linked library) 和动态库 (dynamically linked library) 两种形式。在 Linux 环境中, 动态库也称共享库 (shared library)。

静态库 静态库由一组目标文件 (.o) 使用 `ar` 命令打包而成, 上面生成的 fibo.o 就可以作为库的一部分, 使用 `ar` 创建 libalgorithm 库:

```
$ ar -cq libalgorithm.a fibo.o
```

Linux 系统的库文件名前缀是 lib, 静态库的后缀是 .a。多个目标文件可以使用 `ar` 的 “-a”、“-b”、“-q”、“-r” 等选项向归档文件 (库文件) libalgorithm.a 中添加。

只需要了解函数的调用接口, 明确函数的格式 (.h 文件中的原型声明), 主程序就可以很方便地直接使用库的功能, 而不必重复编写函数。编译命令如下:

```
$ gcc -o fibo main.o -lalgorithm  -L .
```

gcc 选项 “-L” 指明额外的库搜索路径 (“.” 表示当前目录),“-l” 指明库名。以上编译出的可执行程序 fibo 运行时不再依赖 algorithm 库。使用 nm 可以看到, 函数 fibo() 已经存在于可执行程序 fibo 中:

```
$ nm fibo|grep fibo
000000000000068a T fibo
```

动态库 动态库通过 gcc 的选项 -shared 生成。用于链接的动态库文件名前缀是 lib, 后缀是 .so。习惯上, 用于运行时的动态库后缀还要加上版本号。编译生成动态库的命令如下:

```
$ gcc -c -fPIC fibo.c
$ gcc -shared -o libalgorithm.so.0.0 \
    -Wl,-soname,libalgorithm.so.0 fibo.o
```

此处用到了 gcc 的 “-Wl” 选项, 将 “-soname,libalgorithm.so.0” 传递给链接器。上面的命令生成库文件 libalgorithm.so.0.0, 还需要手工创建链接库和运行时 (run-time) 库:

```
$ ln -s libalgorithm.so.0.0 libalgorithm.so
$ ln -s libalgorithm.so.0.0 libalgorithm.so.0
```

libalgorithm.so 用于编译时的链接 (gcc 的 “-l” 选项), 编译应用程序的命令与使用静态库的编译方式完全一样。在同时存在静态库和动态库的情况下, gcc 优先选择链接动态库。由 “-soname” 指定的 libalgorithm.so.0 是运行时库。与静态链接的程序不同在于, 动态链接的代码不在可执行程序中。这一点使用 nm 命令可以看出:

```
$ nm fibo |grep fibo
                 U fibo
```

动态链接的程序在运行时仍然依赖动态库。未使用 “-soname” 命名时, 运行时库就是链接库。缺少 (或者没找到) 动态库的程序, 会出现类似下面的错误提示:

```
$ ./fibo 20
./fibo: error while loading shared libraries: libalgorithm.so.0:
cannot open shared object file: No such file or directory
```

动态链接的程序通过下面几种方式找到运行时库:

- 在系统默认的库搜索路径中查找运行时库。默认路径有 /lib、/usr/lib (不包括这些目录的子目录), 64 位平台上还包括 /lib64 和 /usr/lib64。
- 缓存文件 /etc/ld.so.cache。该文件来自 /etc/ld.so.conf 中的目录列表, 通过 `ldconfig`命令生成。
- 环境变量 `LD_LIBRARY_PATH` 包含的目录列表。

动态链接的程序文件比静态链接的小, 因为实现功能的代码不在可执行程序中; 动态链接库中的函数可以为多个动态链接的程序共享, 这正是共享库这一名称的来由。这种共享特性可以大量节省磁盘占用空间。此外, 动态链接的程序还很容易升级: 动态链接库升级后, 只要函数接口不变, 上层应用不需要做任何改动, 甚至不需要重新编译, 就可以直接使用新的库。

以 3.3.2 节程序为例。我们发现 fibo() 函数的效率太低, 对此重新设计了优化的算法, 见清单 3.5。该算法复杂度是 $O(n)$。

清单 3.5: 计算斐波那契序列子程序 fibo.c(升级版)

```
#include <stdlib.h>

long fibo(int n)
{
    int i;
    long *result;

    result = malloc(sizeof(long)*n);
    result[0] = 1;
    result[1] = 1;
    for(i = 2; i <= n; i++)
        result[i] = result[i-1] + result[i-2];
    return result[n];
}
```

升级动态库:

```
$ gcc -shared -o libalgorithm.so.0.1 -shared -fPIC \
  -Wl,-soname,libalgorithm.so.0 fibo.c
```

将新版本的动态库链接到运行时库 libalgorithm.so.0:

```
$ ln -sf libalgorithm.so.0.1 libalgorithm.so.0
```

无须重新编译 fibo, 再次运行程序可以发现, 程序效率有了明显的提高。

3.3.4 软件调试

软件开发过程中, 需要对软件进行各种测试。一旦发现故障, 就需要有一个有效的诊断工具。GDB 就是一个功能强大的软件调试器。

为了能够进行高效的调试, gcc 编译时应使用选项 “-g”, 这样生成的代码包含了源码调试信息, 调试时可以很方便地与源程序对照。在缺乏源代码对照的情况下, 只能以汇编语言方式调试。

启动 gdb

作为一个例子, 我们有意在清单 3.5 中设计一个错误, 将第 13 行改成:

```
    return result[i];
```

运行时发现结果与预期不符: 要么结果不对, 要么内存分配错误。为了诊断错误, 在编译时加上选项 “-g” 重新编译程序 (包括动态库), 然后使用下面的命令进入 gdb 环境:

```
$ gdb -q --args fibo 3
Reading symbols from fibo...done.
(gdb)
```

缺省方式下, 调试一个正在运行的程序, gdb 的第一个参数是待调试的程序, 第二个参数是与之相关的进程 ID; 使用选项 “-p” 指定进程 ID 时, 程序名可以省去。假设正在运行的程序 program 的进程号是 17245, 下面两个方式都是可行的:

```
$ gdb program 17245
或
$ gdb -p 17245
```

如果当前目录下确有一个名为 “17245” 的文件, 以该文件作为 gdb 的参数时, 为避免歧义, 可在文件名前加上目录名, 明确其路径属性, 即 “./17245”。

当被调试的程序带有选项和参数时, gdb 选项 “--args” 告诉 gdb, 被调试的程序同参数一起被带入 GDB 环境。

gdb 环境中, 使用 file 命令可以指定待调试的程序。

运行程序

gdb 完全基于键盘交互方式, 为了便于操作, 所有的命令都可以简化, 即, 在不产生歧义的前提下, 仅使用命令开头的少数几个字母。大量的常用命令被简化为单键操作, 例如, “h” 即 “**h**elp”, “b” 即 “**b**reak” , “s” 即 “**s**tep” (尽管 s 开头的命令不止一个, 但单步命令使用频率极高, 故而 gdb 内部以 “s” 表示单步)。不输入任何命令, 直接敲回车, 等效于重复执行上一条命令。在不确定完整命令的情况下, 可以使用<TAB> 键的命令补全功能。

gdb 环境内包含了多级帮助菜单。缺省的 help 命令打印出第一级帮助 (中文为编者标注):

```
(gdb) h
List of classes of commands:

aliases       -- 命令的别名
```

```
breakpoints  -- 断点相关 (设置断点、观察变量等)
data         -- 数据相关 (打印数据、数据转储等)
files        -- 文件相关 (指定调试文件、列清单、改变目录等)
internals    -- gdb 内部维护命令
obscure      -- 高级特性
running      -- 运行相关 (运行、单步、停止等)
stack        -- 检查堆栈
status       -- 查看状态
support      -- 便捷操作
tracepoints  -- 在不停止程序的情况下跟踪执行
user-defined -- 用户定义的命令
```

假如不知道一项功能准确的命令是什么, 可以用 apropos 对命令进行模糊搜索。

如果已经将调试程序作为 gdb 的参数带入, 可以用 run 或 start 命令启动程序。run 命令在碰到最近的一个断点处暂停, 或碰到程序故障时暂停, 如果既无断点又无异常, 则完整地执行一遍程序。start 命令运行到主函数 main() 的入口处暂停。进入 GDB 环境没有指定选项和参数时, 此时可作为 run 或 start 的参数指定。

```
(gdb) r 4
Starting program: /home/harry/gdb/fibo 4
Fibo(4) is 134529                              # 错误的结果
[Inferior 1 (process 5231) exited normally]
(gdb)
```

需要注意的是, 如果是远程调试, 运行命令 run 已经在服务器启动时发出了, 因此调试端不需要再执行 run 命令, 而是应该执行 continue 命令。

调试功能

使用源码级调试, 源文件 (包括生成调用的动态库的源文件) 应与被调试的可执行程序在同一目录, 否则应使用 dir 命令添加源程序搜索路径。

list 命令打印源文件清单, 缺省方式是打印连续的 10 行。连续打印的行数由 gdb 内部参数 listsize 决定, 可以通过 set 命令改变 listsize。连续的 list 命令从当前行开始继续打印 10 行, 与其他命令的重复方式不同。这种设计使连续打印程序清单的操作更为便捷 (只需要连续回车即可)。list 的参数可以使用下面几种格式:

```
(gdb) l 8                # 打印当前文件第8行前后的10行
(gdb) l fibo.c:5,10      # 打印指定文件的行号范围
(gdb) l fibo             # 打印函数 fibo 附近的10行
```

与动态库相关的源文件信息要等到程序开始运行时才能生效 (run 命令或 start 命令)。

调试时, 我们需要在程序的特定位置设置一些断点, 以便让程序暂停, 观察可疑的参数和变量, 或者改变参数和变量, 再继续运行。设置断点的命令是 break。常用的设置断点格式有下面几种情况:

```
(gdb) b fibo.c:8           # 指定文件的行号,缺省时表示当前文件
```

```
(gdb) b fibo                  # 函数入口或语句标号作为断点
(gdb) b fibo.c:fibo:label     # 以函数内标号作为断点
(gdb) b *main+10              # 断点设在main+10字节处 (适用于汇编语言)
```

设置断点后, 使用 info break 观察断点执行情况, delete breakpoints n 删除指定编号的某个断点。不指定编号时, 清除所有断点。clear 命令清除指定行的断点。

假设我们把断点设置在 fibo.c:8, 开始运行:

```
(gdb) start 4
Temporary breakpoint 1 at 0x7e9: file main.c, line 10.
Starting program: /home/harry/gdb/fibo 4

Temporary breakpoint 1, main (argc=2, argv=0x7fffffffde48) at main.c:10
10    if (argc < 2) {
(gdb) b fibo.c:8
Breakpoint 2 at 0x7ffff7bd3615: file fibo.c, line 8.
(gdb) c
Continuing.

Breakpoint 2, fibo (n=4) at fibo.c:8
8    result = malloc(sizeof(long)*n);
(gdb)
```

每次开始运行或者继续运行, 程序会停在最近的一个断点处。继续运行程序, 除了使用 c 命令, 还有 n(**n**ext) 和 s(**s**tep) 两条单步命令。二者的差别是, 当碰到函数时, s 将进入函数内单步执行, 而 n 将函数当作一条指令一次走完。

在当前的停留位置, 如果我们对函数 `malloc()` 的具体实现过程不关心, 则应使用 n:

```
(gdb) n 3                             # 执行 next 命令3次
11          for(i = 2; i <= n; i++)
```

继续下面几步, 这次把断点设在 12 行。重复的循环操作比较耗时, 可以为断点增加测试条件, 让程序跳过我们不关心的步骤:

```
(gdb) b 12 if i==4
(gdb) c
Breakpoint 3, fibo (n=4) at fibo.c:12
12              result[i] = result[i-1] + result[i-2];
```

display 命令可以将待显示的变量加入显示列表, 每次停在断点处打印这个列表。delete display n 删除指定列表项, undisplay 删除列表表达式。

断点停留位置的指令尚未执行, 因此还要再执行一次单步命令才能得到需要的结果。

程序停在 13 行时, 使用 p (**p**rint) 命令打印我们关心的变量:

```
(gdb) p result[4]
$1 = 5
```

```
(gdb) p result[i]
$2 = 134529
(gdb) p i
$3 = 5
```

至此, bug 基本已定位: 原因是函数 `fibo()` 返回值下标 i 已在循环中增加, 而这不是我们希望的情况。只要将变量 i 改成 n, 就可以得到正确的返回结果:

```
(gdb) set variable i=4
(gdb) c
Continuing.
Fibo(4) is 5
[Inferior 1 (process 25832) exited normally]
```

打印多个数组元素使用 `*array@len` 或 `(type[len])array` 作为地址参数, 命令格式为 print/FMT ADDRESS, 其中 FMT 有如下格式:

t 打印二进制整数形式。[16]

o 打印八进制 (**o**ctal) 整数形式。

d 按十进制 (**d**ecimal) 带符号整数形式打印。

u 按无符号 (**u**nsigned) 十进制整数形式打印。

x 按十六进制 (he**x**adecimal) 整数形式打印。

z 按十六进制打印, 但在左边用 0 补足到变量类型的长度。

f 按浮点数 (**f**loating point) 格式打印。

c 按字节打印字符 (**c**har), 不可打印字符的符号使用 `\nnn` 格式。

s 打印字符串 (**s**tring)

a 打印地址 (**a**ddress), 包括十六进制地址值和程序内部标号的偏移值。

r 打印裸数据 (**r**aw)。

```
(gdb) p/x *result@10
$4 = {0x1, 0x1, 0x2, 0x3, 0x5, 0x20d81, 0x0, 0x0, 0x0, 0x0}
(gdb) p/d (long[8])*result
$5 = {1, 1, 2, 3, 5, 134529, 0, 0}
```

gdb 常用命令

gdb 功能非常强大, 针对多线程、后台进程都有比较完备的调试手段。表 3.17 是最常用的一组命令。

[16]表示二进制 binary 的首字母已经用于 break, 它的使用频度要高得多。这里的 t 取自 two 的首字母。

表 3.17: gdb 常用命令

命令	功能
break	设置断点
clear	删除所在行的断点
continue	继续执行正在调试的程序
display	设置程序停止时显示的表达式/变量
file	装载待调试的可执行文件
info	显示文件、函数、变量、断点等信息
list	显示源代码清单
next	执行下一行源代码, 遇有子程序时不进入
print	显示表达式/变量的值
quit	退出 gdb
run	运行程序
set	参数设置
start	运行被调试程序到主程序入口
step	单步执行下一行源代码, 遇有子程序时进入

汇编语言调试命令

如果 gcc 编译时没有 "-g" 选项, 或者后期经过 `strip` 命令处理, 可执行程序中不再带有源码信息。这种情况下, 调试难度大大增加, 不过gdb 仍有操作空间。

列汇编语言清单使用内存检查命令 "x" (e**x**amine):

```
(gdb) x/5i main
    0x55555555471a <main>:        push   %rbp
    0x55555555471b <main+1>:      mov    %rsp,%rbp
 => 0x55555555471e <main+4>:      sub    $0x20,%rsp
    0x555555554722 <main+8>:      mov    %edi,-0x14(%rbp)
    0x555555554725 <main+11>:     mov    %rsi,-0x20(%rbp)
```

"5i" 表示 5 条指令 (**i**nstruction), 箭头所指处为当前断点。需要说明: 对于汇编语言源程序带有 "-g" 选项编译生成的可执行程序, 调试时仍可以使用 list 命令列清单。

汇编语言调试, 面对的主要是寄存器和存储器操作。显示寄存器的命令是 info registers 和 info all-registers, 前者不包括浮点寄存器和矢量寄存器。对单个寄存器的修改和显示同对普通变量的操作方法类似:

```
(gdb) set $rax=12345678
(gdb) p/x $rax
$1 = 0xbc614e
```

由于失去了行号信息, 断点只能以指令标号地址的形式设置, 且 X86 指令的长短不一, 错误的断点设置会导致调试故障。

3.4 GNU Make

一个大型项目包含许多源程序文件。在项目开发过程中, 程序员需要不厌其烦地反复输入编译命令。在 3.3.2 节的例子中, 为了避免重复操作, 可以将编译过程写进下面的脚本文件中:

```
#!/bin/sh
gcc -c main.c
gcc -c fibo.c
gcc -o fibo main.o fibo.o
```

并为它加上可执行属性。之后可以用一条简单的命令 `./build.sh` 代替三行编译命令。

这种做法缺乏一定的灵活性。当在项目中增减文件时需要修改编译脚本, 无形中增加了一项工作, 而且不能节省编译的时间。例如, Linux 内核有上万个源代码文件, 完整地编译一遍需要二、三十分钟 (取决于计算机的性能)。如果修改了其中的部分文件, 再次编译时, 并不需要将所有文件都重新编译一次, 只需要编译修改过的文件, 以及与修改过的文件存在依赖关系的文件。这样就可以大量节省编译时间, 提高开发效率。

这样的软件开发方式可通过 GNU Make实现。

3.4.1 Makefile 基本结构

仍以 3.3.2 节的程序为例。为了编译这个项目, 我们在该目录下另写一个文件 Makefile:

清单 3.6: 文件 Makefile

```
fibo: fibo.o main.o
        gcc -o fibo main.o fibo.o

fibo.o: fibo.c
        gcc -c fibo.c

main.o: main.c fibo.h
        gcc -c main.c

clean:
        rm -f fibo main.o fibo.o
```

使用 `make` 执行编译命令, 结果如下:

```
$ make
gcc -c main.c
gcc -c fibo.c
gcc -o fibo main.o fibo.o
```

当修改了其中的一个源文件后, 重新编译, 可看到下面的结果:

```
$ touch main.c        # 修改main.c的时间戳
```

```
$ make
gcc -c main.c
gcc -o fibo main.o fibo.o
```

注意到, 第二次执行 make 命令时, 文件 fibo.o 不会重新编译。因为即使重新编译, 新生成的 fibo.o 也与原来的完全一样。如果再次执行 make, 我们看到:

```
$ make
make: 'fibo' is up to date.
```

它告诉我们, 用于生成最终文件 fibo 的所有依赖文件都是旧的, 所有编译和链接命令都不需要再做。

GNU Make 根据一个脚本文件, 按文件的时间戳建立依赖关系, 并根据给定的规则生成目标文件。默认情况下, GNUMakefile 是 GNU Make 的第一顺位脚本, 接下来依次是 makefile 和 Makefile。Linux 系统习惯使用 Makefile, 据说是因为 Linux 系统中的文件习惯以小写字母命名, 首字母大写的文件名在众多文件中比较容易找到。如果不接受默认, make 使用选项 "-f" 指定脚本文件。

Makefile 文件中描述了文件的生成规则。一个 Makefile 文件中可以有多项规则, 每项规则由目标、依赖文件和动作三部分组成。目标和依赖文件之间用冒号 ":" 分开, 多个依赖文件之间用空格分隔。如果依赖文件比较多, 可以多行书写, 使用 "\" 作为换行符。

目标可以是由动作生成的文件, 也可以是一个单纯的字符串标号, 如清单 3.6 中的 "clean", 这样的目标被称为伪目标 (pseudo target)。GNU Make 也允许多目标。使用同一个规则描述多个目标的依赖关系比较复杂, 而且容易产生规则定义不明, 应尽量避免这样的形式。

目标行下面的命令被称为动作。Makefile 默认动作前面必须是制表符, 不能用多个空格代替。如果不喜欢制表符引导动作的格式, 可以重新定义变量.RECIPEPREFIX。一般情况下, 达成一个目标使用一个动作; 如果有多个动作, 可以按动作的先后顺序写在依赖关系的下面。这种做法本身没有问题, 只是有可能导致多余的重复动作。

make 有选项很多, 常用的有下面几个:

-f file 指定文件 file 作为 make 的脚本文件

-C dir 进入目录 dir, 执行该目录下的 GNU Make 脚本。在一个大型项目中, 各个模块被组织在不同的子目录中, 每个目录都可能有一个 Makefile, 用于指导一个模块的编译过程。

-j jobs 指定可并行工作的数量, 在多核处理器中, 此选项可以大大加快编译速度。

-p 打印规则和变量。

缺省参数情况下, make 会执行第一个目标的动作。因此, 在编写 Makefile 时通常都会把实现项目的最终目标 (又叫终极目标) 的规则写在最前面, 这样可以省去在命令行中为 make 指定参数。如果明确指定了参数, GNU Make 就以该参数为目标。例如, 基于清单 3.6, 下面的命令仅生成 fibo.o:

```
$ make fibo.o
```

除了终极目标所在的规则以外, 其它规则在 Makefile 文件中的顺序无关紧要。

3.4.2 GNU Make 变量

清单 3.6 的 Makefile 只是机械地重复了键盘命令, 对每个 ".o" 文件都要手工建立一个规则。随着源文件数量的增加, 编辑 Makefile 也成了一项额外的负担。GNU Make 内建的变量和规则可以帮我们简化 Makefile 的编写。

GNU Make 有三类变量: 预定义变量 (内部变量)、自动化变量和定义变量。

GNU Make 内部定义了一些变量, 表 3.18 是比较常用的部分。内部变量可以使用`make`的选项 "-p" 打印出来。如果存在内部变量, 建议尽量使用它们, 这可以使 Makefile 更加规范。此外, 所有环境变量都作为 GNU Make 的预定义变量。

表 3.18: GNU Make 的主要预定义变量

预定义变量	含义	默认值
AR	归档维护程序名	ar
AS	汇编程序名	as
CC	C 编译器名	cc
CXX	C++ 编译器名	g++
CPP	C 预编译器名	$(CC) -E
FC	FORTRAN 编译器名	f77
LEX	Lex 到 C 语言转换器	lex
PC	Pascal 语言编译器	pc
RM	删除	rm -f
YACC	Yacc 的 C 解析器	yacc
YACCR	Yacc 的 Ratfor[17] 解析器	yacc -r
TEX	tex 编译器 (生成.dvi)	tex
以下是命令的选项/参数		
ARFLAGS	归档维护程序的选项	rv
ASFLAGS	汇编程序的选项	(空)
CFLAGS	C 编译器的选项	(空)
LDFLAGS	链接器 (如 ld) 的选项	(空)
CPPFLAGS	C 预编译的选项	(空)
CXXFLAGS	C++ 编译器的选项	(空)
FFLAGS	FORTRAN 编译器的选项	(空)
LFLAGS	Lex 解析器选项	(空)
PFLAGS	Pascal 语言编译器选项	(空)
YFLAGS	Yacc 解析器选项	(空)

[17] **rat**ional **for**tran, FORTRAN 语言的一个分支, 作者 Brian Kernighan

第二类变量形式是自动化变量, 它们是随上下文关系发生变化的一类变量, 在 Makefile 中有非常重要的作用。表 3.19 中列出了主要的自动化变量。

表 3.19: GNU Make 的自动化变量

符号	含义
$@	规则中的目标文件名
$<	规则的第一个依赖文件名
$^	规则的所有依赖文件列表 (不包括重复的文件名), 以空格分隔
$+	和 $^ 类似, 但保留重复出现的文件
$?	所有比目标文件更新的依赖文件列表, 以空格分隔
$%	当目标是静态库时, 表示库的一个成员名
$*	模式规则中的主干, 即 “%” 所代表的部分, 一般表示不包含扩展名的文件名

第三种形式是用户定义的变量, 它可以随 `make` 命令导入, 也可以在 Makefile 文件中定义。

利用 GNU Make 提供的变量资源, 编译一个 C 语言文件的规则可写成:

```
main.o: main.c fibo.h
    $(CC) -c $<
```

引用变量方法是在变量前面加 “$” (表 3.19 的自动化变量已经有了 $, 不需额外再加)。如果变量名由多个字母组成, 需要用括号 “( )” 或 “{ }” 把变量名括起来, 否则会以第一个字母作为变量名。变量 CC 是 C 语言编译器命令, 默认是`cc` (UNIX 系统的编译器名)。出于兼容性考虑, Ubuntu 系统中已将命令 `cc` 链接到 `gcc`。如果要明确编译器命令, 可以在 Makefile 文件中第一个引用之前定义它:

```
CC  = gcc
```

或者随`make` 命令定义:

```
$ CC=gcc make
```

变量的定义, 一方面避免重复, 同时也便于替换, 例如在针对 Arm 平台的编译项目, 只需要修改 Makefile 一处, 将变量 CC 重新定义成 `arm-linux-gcc` 即可, 其他地方不需要做任何改动。

3.5 源代码移植

Linux 世界拥有大量的开源软件, 为各种应用提供了相当大的便利条件。这些软件不仅可以免费获得源代码, 自由地修改源代码; 只要遵守软件的版权协议, 甚至不排斥商业用途。

3.5.1 获取源码

在互联网上, 目前获取开源软件源代码的集中途径主要有各发行版源码仓库、各版本控制系统 (**C**ontrol **V**ersion **S**ystem, CVS) 托管服务器 (github、gitlab 等等)、sourceforge.net。

通过发行版源码库

Ubuntu 默认安装时, 源代码目录是不开放的。如果想通过 Ubuntu 仓库下载源码, 应在其包管理器的仓库源列表文件 /etc/apt/sources.list 中, 去掉 deb-src 前面的注释, 再执行更新命令, 就可以直接从发行版仓库中下载源代码了:

```
# apt update
```

假设要下载一个名为 hello 的源码包, 使用下面的命令 (下载源码不需要超级用户权限):

```
$ apt source hello
```

命令正确执行后, 会获得一个 hello 的压缩包, 以及该压缩包解压的目录。

Ubuntu 源码仓库使用 HTTP 协议, 因此也可以直接使用浏览器通过网页功能下载:
http://cn.archive.ubuntu.com/ubuntu/pool/main/h/hello/
其他 Linux 的各大发行版都有类似的 HTTP 服务器, 下载方式大同小异。

通过 git 托管服务器

目前最著名的 GIT 托管服务器是 https://www.github.com 和 https://www.gitlab.com, 它们都提供了网页访问途径。可以在它们的网页上搜索需要的软件, 进入下载页面, 通过浏览器下载软件的压缩包。压缩包只包含特定的版本, 它通常比 `git clone` 下载来的数据少得多。如果没有进一步开发要求, 下载压缩包更合算一些。

除了以上途径, 网上也分散着大量的开源软件。例如, 通过 Ubuntu 软件仓库服务器获得的 hello 也可以在 GNU 的官网 https://www.gnu.org中找到。需要注意的是, 发行版源码仓库提供的软件通常是较新的稳定版, git (也包括其他版本控制系统) 服务器可以找到最新的版本, 但可能未经严格的测试, 而通过搜索引擎找到的软件可能比较老旧 (与搜索引擎本身和搜索技巧等因素有关)。另外, 使用时应注意阅读软件的版权协议。

3.5.2 源码结构

以下以 hello 源码为例, 介绍它的移植过程。

首先, 将压缩包解压。文件 `hello_2.10.orig.tar.gz` 是从 Ubuntu 软件仓库得到的源码压缩包, 解压命令是:

```
$ tar xf hello_2.10.orig.tar.gz
```

然后进入这个解压目录 hello-2.10, 注意这个目录下的一些关键文件。下面是大多数 GNU 开源软件具有的文件和目录:

```
|-- AUTHORS          作者名单
|-- autogen.sh       自动配置的脚本程序
|-- ChangeLog        软件修改、升级日志文件
|-- configure.ac     自动配置的脚本文件
|-- README           关于软件的说明文档
|-- COPYING          版权协议文件
```

```
|-- configure           用于配置编译环境的可执行文件
|-- Makefile            GNU Make 脚本
|-- GNUmakefile         GNU Make 脚本
|-- CMakeLists.txt      用于 cmake 配置编译环境的脚本
|-- INSTALL             安装说明文档
|-- src/                包含源程序的目录
|-- include/            软件自身的头文件目录
|-- man/                包含帮助手册的目录
|-- doc/ 或 docs/       文档目录
`-- po/                 用于多语言支持的字符串转换文档的目录
```

想了解软件的功能、用法, 应该阅读 README 以及 man 和 doc 或 docs 目录中的文档; 如果想继续开发/修改软件, 重点关心 src、include 目录里的内容; 如果要编译/移植, INSTALL 文件中一般会有编译/安装的说明; 另外再请阅读一下版权协议 COPYING, 看看是否符合软件的授权要求。除此以外, 几个直接与编译相关的文件是 autogen.sh、Makefile/GNUmakefile、CMakeLists.txt 和 configure, 多数软件包里常常会有其中的几个。

3.5.3 配置编译环境

如果源码目录下已经有了 Makefile, 可以尝试执行 `make` 命令。不过最好还是根据自己的环境做一些设置。Linux 操作系统应用软件的一个很大特点是以模块化形式层层堆叠, 软件之间的依赖关系很常见。直接使用软件包提供的 Makefile 未必能满足编译条件。实际上大多数源码都不直接提供 Makefile, 而是通过配置工具在当前环境下生成 Makefile。

生成 Makefile 可以按以下步骤实现:

1. 如果存在 configure 文件, 则可以运行:

   ```
   $ ./configure
   ```

   它会检查当前的开发环境: 针对什么运行平台, 编译器是否合适, 开发软件是否齐全, 依赖的软件是否已经具备, 等等。如果条件都已满足, 再根据当前开发环境生成 Makefile。生成的 Makefile 也许不止一个, 各个子目录中可能都会有, 不过在编译时只需要在主目录上执行 `make`, 各子目录 Makefile 的调用会由主 Makefile 自行解决。

   configure 有很多选项, 可以控制生成不一样的 Makefile, 例如选项编译静态库还是动态库, 在可以依赖多个软件时选择依赖哪个, 安装时安装目录的指定等等。使用 `./configure --help` 可以详细查看各选项的说明。
2. 如果没有 configure, 但存在 configure.ac, 可以使用 autoconf 工具生成 configure:

   ```
   $ autoreconf -ivf
   ```

3. 另一个生成 configure 的途径是通过 autogen.sh:

   ```
   $ sh autogen.sh
   ```

4. 如果存在 CMakeLists.txt, 则可以用 `cmake` 工具生成 Makefile:

   ```
   $ cmake .
   ```

   注意上面命令最后的一个点 (.), 它表示以当前目录中的 CMakeLists.txt 为脚本。

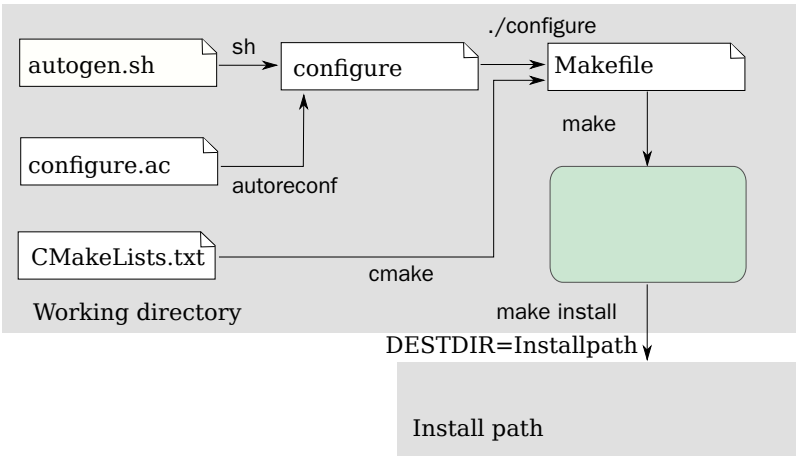


图 3.4: 编译和安装过程

3.5.4 编译与安装

以上配置完成后, 生成 Makefile, 接下来就是使用 GNU Make 工具编译和安装了。原则上只需要执行下面两条简单的命令 (也可以并在一行, 两条命令中间用 `&&` 分隔):

```
$ make
$ make install
```

`make` 命令根据 Makefile 的指示完成软件的编译工作, `make install` 将编译的结果复制到安装目录 (此过程并非简单地复制, 可能还包含一些后期处理, 如 `strip` 符号等)。缺省的配置方式下, 安装目录的起点是 /usr/local, 该目录对普通用户没有写入权限。如果一定要安装到这个目录, 需要超级用户权限。安装到这个目录的好处是, 多数源码在编译配置过程中都已经将 /usr/local/lib、/usr/local/include 路径加在 gcc 的编译选项中, 多层编译时依赖库的设置比较简单; 环境变量 (environment variables) `PATH` 包含 /usr/local/bin 这个目录, 可以直接运行安装在这个目录里的程序。

如果不想接受缺省的安装目录设置, 有两个方法改变它:

1. 在 configure 命令中用选项 --prefix 指定安装目录:

   ```
   $ ./configure --prefix=/opt/devel
   ```

2. 在 make install 命令中用变量 DESTDIR 指定一个有写入权限的安装目录:

   ```
   $ make install DESTDIR=/opt/devel
   ```

假设我们已经将 hello 这个软件安装到 /opt/devel 目录, 进入 /opt/devel/usr/bin 应可以看到一个可执行文件 hello, 此时可以尝试着运行它了:

```
$ ./hello
Hello, world!
```

3.6 内核重构

Linux 操作系统内核是遵循 GPL 版权协议的开源软件。当已有的二进制代码不能满足需要的时候, 还可以直接从源码编译一个完整的内核。

3.6.1 为什么要编译内核

大多数 Linux 桌面发行版的内核满足一般用户的普遍需求, 要求用户自己编译内核的情况很少, 除非用户想尝试最新的内核版本, 因为 Linux 官方发行版内核通常不是最新的; 或者需要运行特殊的软件, 而这个软件对内核版本或者内核特性有要求。

但在嵌入式平台, 需要编译内核的情况很普遍。嵌入式系统通常会有对系统资源的限制条件, 不同平台之间的差异也很大, 很难像 PC 使用的操作系统那样设置成统一的参数。

编译内核的起点是获得内核源码。由 Linux 内核核心团队维护的内核源码可以在 https://www.kernel.org 下载并解压。该网站同时也提供了版本控制系统 git 的下载方式。此外, 一些嵌入式平台有自己的源码分支, 更具可配置性。例如, 以 TI AM335X 处理器为核心的各种嵌入式开发板 (Beagle Board), 其内核维护网址在 https://github.com/beagleboard/linux, 其中包含了对 Beagle Bone Black 的支持。本节就以它为起点, 介绍内核的源码结构和编译过程。Beagle Board 的核心处理器与 PC 的架构不同, 除了编译器的差异以外, 其他过程与针对 PC 的内核编译基本相同。下面的命令将源码克隆到本地:

```
$ git clone https://github.com/beagleboard/linux
```

下面针对内核的讨论, 以克隆目录 linux 为起点。

3.6.2 内核源码结构

内核源码树结构中包含下面的目录和文件:

arch 该目录下包含与体系架构相关的内核代码。每种体系架构在此目录下单独占一个子目录。子目录中的 configs 目录包含支持该架构的不同机型默认的配置文件。建议编译内核从该配置文件开始, 这样可以大大减少之后繁琐的配置。例如, 文件 arch/arm/configs/bb.org_defconfig 是为 Beagleboard 预制的内核配置。

block 该目录包含块设备层核心代码。与块设备驱动相关的代码在 drivers/block 目录中。存储设备相关的代码分布在 drivers 目录中的具体驱动程序里。

COPYING 版权协议文本。

crypto 该目录包含数据压缩/解压和加密/解密核心代码。内核和初始化 RAMDisk 镜像文件都可能采用压缩格式, 以减少存储空间占用。

Documentation 该目录包含内核源代码各个模块的说明文档。

drivers 设备驱动目录, 其下的每一个目录是一类设备的驱动程序。

firmware 该目录下包含与一些设备相关的固件。

fs 文件系统目录。每一个具体文件系统的实现代在此目录下单独构成一个子目录, 最终汇聚到虚拟文件系统层。

include 内核的头文件目录。它也是模块包含头文件的起点。与体系结构相关的头文件分布在 arch 各个子目录中。

init 内核的主文件代码。

ipc 该目录包含进程间通信的核心代码 (消息、信号量、共享内存)。

Kbuild 编译内核的脚本文件。

Kconfig 配置内核的脚本文件, 各级子目录中都有一个 Kconfig, 它们用来帮助形成内核配置界面的菜单。

kernel 内核的核心代码。此目录下的文件实现大多数 Linux 的内核函数。

lib 该目录包含内核使用到的库 (编解码、校验)。

mm 该目录包含所有独立于 CPU 体系结构的内存管理代码, 如页式存储管理、内存的分配和释放等。与体系结构相关的代码分布在 arch 的各个子目录中, 例如与 arm 相关的存储管理代码在 arch/arm/mm 中。

MAINTAINERS 该文件是内核维护者的名单。

Makefile 编译内核的 GNU Make 脚本文件。主目录下的 Makefile 用于控制整个系统的配置和编译。各级子目录中的 Makefile 用于描述依赖关系。

net 该目录包含以太网、无线、蓝牙、红外等多种网络通信协议的核心代码。与网络设备相关的驱动程序代码包含在 drivers/net 目录中。

README 内核的说明文档, 其中有关于内核配置和编译的简单说明。

samples 该目录包含一些模块的源码样例, 它们通常不编译进内核。

scripts 该命令包含配置和编译内核的脚本程序, 是编译内核的辅助工具。脚本程序根据源码每个目录下的 Makefile 和 Kconfig 形成配置界面, 帮助用户配置和编译内核。

security 该目录包含与内核安全性相关的代码。

sound 该目录包含 Linux 声音系统的核心 ALSA (**A**dvanced **L**inux **S**ound **A**rchitecture) 和设备驱动程序。

一个隐藏文件 .config 记录了需要编译的内核的所有配置选项。配置内核的过程就是编辑这个文件的过程。理论上说, 你甚至可以直接使用文本编辑工具编辑配置选项。不过由于内核的功能太多, 很少人直接手工编辑这个文件, 而是通过内核专门提供的可视化配置界面编辑这个文件。

3.6.3 配置和编译内核

Beagle Board 的处理器采用 Arm 指令集架构, 需要安装针对 Arm 的交叉编译工具:

```
# apt install g++-arm-linux-gnueabihf
```

根据依赖关系, 上述命令会同时安装交叉编译工具中的 C 编译器、链接器和 glibc 库。

内核使用 GNU Make 控制配置和编译过程。通常有下面几种命令方式打开配置界面:

1. ARCH=arm make menuconfig

   这是基于 ncurses 库的字符配置界面。Makefile 根据变量 ARCH 选择编译何种架构, 缺省的方式是主机的架构 X86。通过键盘将光标移动到选项位置, 用 “y”、“n”、“m” 决定该选项编入内核、不编入内核或者编译成独立的模块。编入内核意味着内核具备该项功能; 编译成独立模块将在安装模块时生成以 .ko 为后缀的文件。编译成模块的, 内核启动后可通过模块加载或卸载命令动态调整内核的该项功能。
2. ARCH=arm make xconfig

   xconfig 打开基于 Qt 库的图形配置界面 (图3.6), 用鼠标勾选内核的选项。

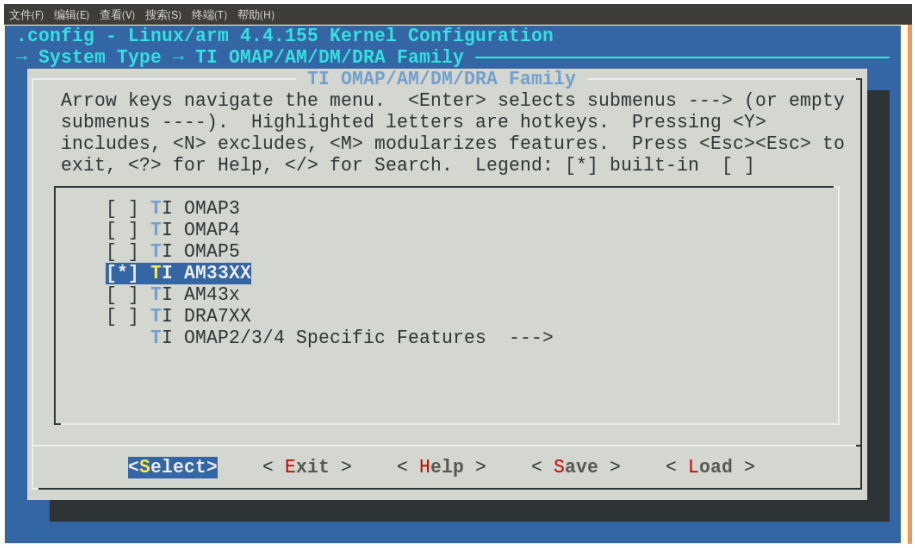


图 3.5: menuconfig 界面

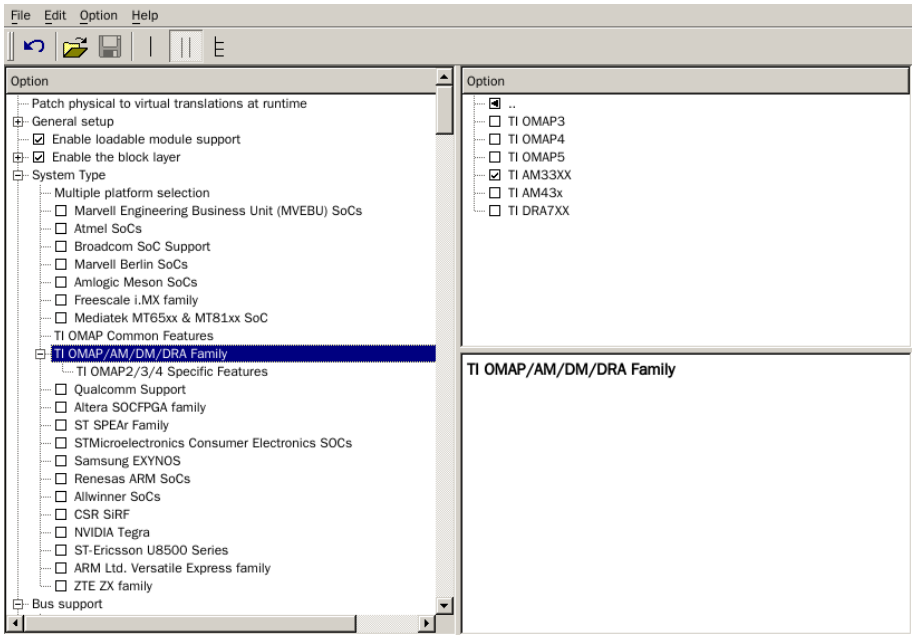


图 3.6: xconfig 界面

3. ARCH=arm make gconfig

   gconfig 打开基于 GTK+ 库的图形配置界面 (图3.7)。操作方法与 xconfig 完全相同, 只是界面风格不一样。

以上配置的结果最终会保存在一个名为 .config 的文件中。

内核配置选项繁多。4.4 版本的内核, .config 文件近 5 千行。逐项配置的工作量巨大。开发人员为一些通用平台预制了基本的配置文件, 这些文件在内核源码的 arch/$ARCH/configs 目录里, 可以直接将这个文件复制到 .config, 然后以此为起点进行配置。标准的做法是用 GNU Make 复制:

```
$ ARCH=arm make bb.org_defconfig
```

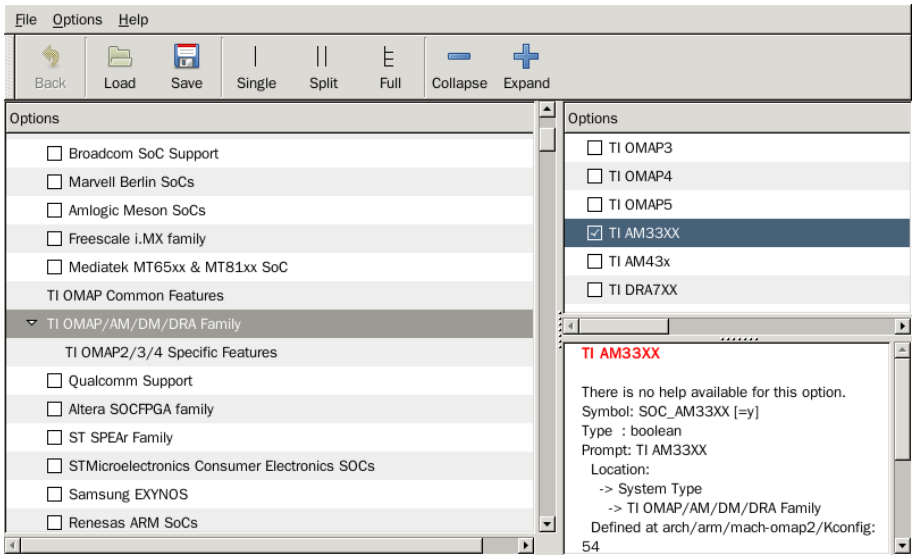


图 3.7: gconfig 界面

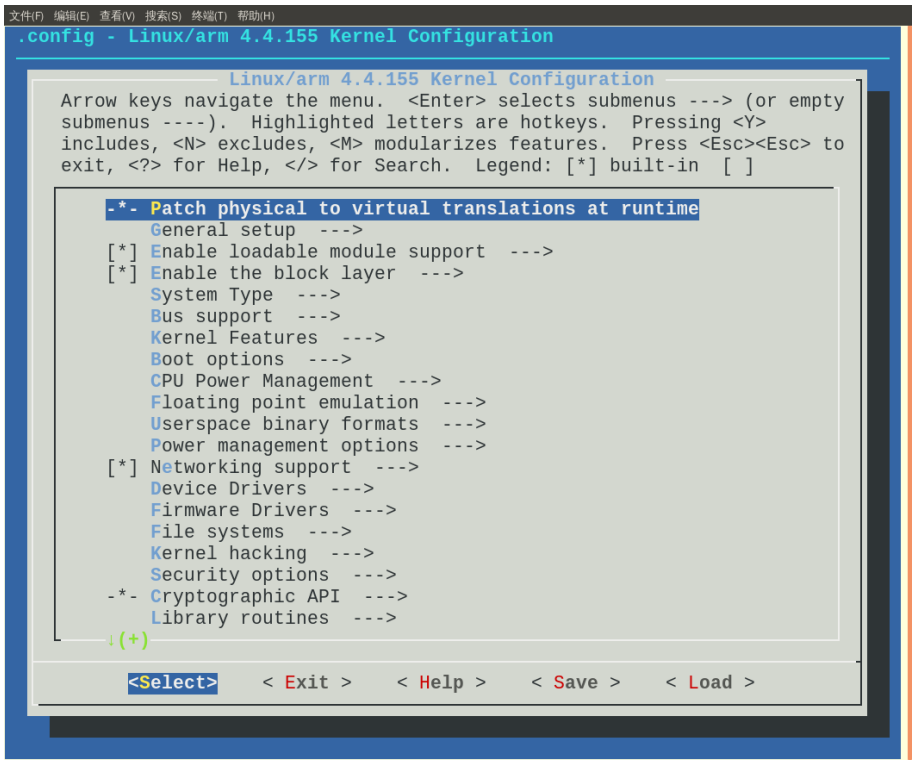


图 3.8: 配置内核主界面

预制的配置选项保证系统的基本可用, 开发者还需要在此基础上进行细心的配置。图3.8是内核配置主界面, 各子菜单包含如下内容:

- General setup: 一般性设置, 如编译器选择、内核镜像压缩格式、System V IPC 机制等。
- Enable loadable module support: Linux 系统大量的设备驱动和模块以可加载模块的方式存在, 需要使用时通过 `insmod` 将其加入内核。除非是限定功能的系统, 否则应允许模块加载。

- Enable the block layer: 块设备层, 支持块设备和一些大文件系统。
- Platform selection: 系统选型。图 3.5 是该菜单下的一个子菜单界面。
- Bus support: 总线 (PCI 和 PCIe)。
- Kernel Features: 内核特性选项, 如多处理器支持、分页方式、内核/用户空间内存划分、内核时间片设置等。
- Boot options: 启动选项。通常这些选项是由 Bootloader 传给内核的, 但也可以在配置内核时设定启动参数。
- Userspace binary formats: 用户空间的二进制格式。
- Power management options: 系统电源管理 (待机和休眠方式)。
- CPU Power Management: CPU 电源管理。
- Networking support: 网络协议支持, 包括因特网协议、蓝牙、CAN 总线、无线网络协议等。
- Device Drivers: 大量的设备驱动集中在这个子菜单下。
- Firmware Drivers: 固件驱动支持。
- File systems: 选择希望系统支持的文件系统类型。
- Virtualization: 虚拟化 (基于内核的虚拟机支持)。
- Kernel hacking: 用于探究内核工作过程的一些选项, 包括打印消息、允许内核级 debug 等等。
- Security options: 内核的安全性选项。
- Cryptographic API: 内核用到的加密/解密算法。
- Library routines: 主要是一些压缩和校验算法子程序。

对于嵌入式系统来说, 在保证满足功能需求的前提下, 越少的选项意味着越高的效率。开发嵌入式产品时, 在内核源码提供的功能上进行合理的裁剪, 是一个重要的过程。

完成配置、退出配置界面后, 会生成新的 .config 文件, 原来的文件备份到 .config.old, 以便于发现配置不合适时回撤一次。

下面是内核的编译过程。如果在配置内核时没有指定编译器, 需要在命令行中将编译器前缀以变量 `CROSS_COMPILE` 传递给 Makefile:

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make
```

成功编译后, 会在 arch/arm/boot 目录下生成内核镜像文件 zImage。剩下的工作就是将镜像文件交给引导加载器 (Bootloader) 引导启动了。

以上只实现了内核的部分功能。以模块方式支持的部件 (主要是设备驱动) 还要通过 `modules_install` 编译和安装。安装模块[18]时用户应拥有指定的模块安装目录 `modules_install_path` 的写权限。不指定 `INSTALL_MOD_PATH` 时, 普通用户执行安装命令时会出错, 因为默认的安装目录 (/lib/modules) 对普通用户没有写权限。

```
$ ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- make modules_install \
INSTALL_MOD_PATH=modules_install_path
```

最后, 将 `modules_install_path` 中的内容搬到目标系统的根文件系统 /lib/modules 目录, 一个新的内核就完成了。

[18]由于此时用于开发的计算机自身并不需要这些模块, 此处使用 “安装” 一词不是很贴切, 它仅仅是将目标系统需要的模块复制到指定的位置。

■ 本章练习

1. 分别讨论 old 和 new 各为一个已存在的文件或目录、以及 new 不存在时, 命令 "`mv old new`" 操作导致的结果。
2. 试创建与你同组的两个用户 user1 和 user2, 允许 user1 查看 user2 的文档, 不允许 user2 查看 user1 的文档。试问如何设置二者的主目录权限? 分别使用 user1 和 user2 登录计算机验证你的结果。
3. 以下哪个命令可以用来查看文件的内容?

   A. file　　B. type　　C. cat　　D. show

4. inode 中不含有下列的哪个信息?

   A. 文件类型　　B. 文件大小　　C. 文件名　　D. 创建时间

5. 要在系统启动时自动挂载一个特定的分区, 应该编辑下面的哪个文件?

   A. /etc/mount　　B. /etc/fstab　　C. /etc/filesystem　　D. /etc/mtab

6. 如果想知道某个分区上还有多少剩余空间, 应该用下面的哪个命令?

   A. du　　B. df　　C. free　　D. fsck

7. 以下哪个命令使用了管道?

   A. (id; ls)>output　　B. rm -- [A-Z]*　　C. ps ax|more　　D. cat << EOF

8. 空格在命令行中的作用是什么? 你如何创建一个文件名中带空格的文件, 又如何将它删除?
9. 如果要求修改 FAT 文件系统, 使其能在一个目录下同时创建 README.TXT 和 readme.txt 两个文件, 是需要增加代码还是删减代码?
10. 任选一个英文文本文件, 使用`sed` 命令完成如下工作:

    (a) 删除其中所有由大写字母组成的单词;
    (b) 将前 10 行所有单词改为首字母大写;
    (c) 提取所有含数字的行, 转存到另一文件;
    (d) 在每一行行尾添加一个特殊字符 "$"。

11. 组成 GCC 的有哪些软件包?
12. 编写一个文件复制程序 copy, 要求实现下面的功能:

    (a) 运行命令 "`./copy file1 file2`" 时, 将文件 file1 复制到 file2;
    (b) 运行命令 "`./copy < file`" 时, 将文件 file 打印在标准输出设备 (终端) 上;
    (c) 不带参数运行时, 该程序从键盘读取输入, 输出到 /dev/null。

13. 至少用 2 个 C 文件实现上题功能, 编写 Makefile 完成该项目的编译。
14. 尝试用你编写的 copy 将/dev/null 复制一份, 会得到什么? 为什么?
15. 试设计一个项目, 按以下目录结构组织文件, 项目实现功能自定, 使用 GNU Make 管理编译过程。

```
|-- src
|   |-- ts.c
|   |-- ts.h
|   `-- Makefile
|-- test
|   |-- ts_test.c
|   |-- private.h
|   `-- Makefile
|-- main.c
`-- Makefile
```

项目要求将 src 目录下的程序生成库 (libts.a 或 libts.so), 供主程序 main.c 开发时调用。test 目录中是一个测试程序, 依赖 libts, 与主程序无依赖关系。

16. 试将本章斐波那契序列算法的动态链接库改用通项计算公式实现。通项公式为:

$$F(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

17. 编译内核时, 哪些功能作为模块编译, 哪些功能编译进内核镜像, 决策根据是什么?

第 4 章

模块与设备驱动

设备驱动程序在操作系统内核中扮演着重要的角色。它们是一个个独立的黑盒子，为特定的硬件响应定义的内部编程接口，同时隐藏了设备的工作细节。用户操作通过一组标准化的调用完成，而这些调用是和特定的驱动程序无关的。将这些调用映射到作用于实际硬件的设备特定的操作上，则是设备驱动程序的任务。

在 *Linux* 操作系统中，设备驱动程序是内核中的一个模块。它运行于内核空间，具有对系统核心的操作权限，通过一组标准化的接口提供给应用程序操作设备的功能。一方面保证了系统的安全性，另一方面也隐藏了用户无需了解的设备细节，成为用户程序和设备之间的一个模块化接口。

本章从介绍 *Linux* 设备驱动程序的基本概念出发，阐述了内核模块、字符设备驱动程序的编写和调试，以及硬件管理与中断处理等方面的内容。

4.1 设备驱动程序简介

驱动程序是硬件和应用软件之间的接口。理论上，所有的硬件设备都需要安装相应的驱动程序才能正常工作。驱动程序为应用软件提供访问硬件设备的方法，允许应用软件从硬件设备上获得数据，向硬件设备传递信息。

设备驱动程序 (device drivers) 的开发与普通应用程序差别很大。由于设备驱动程序直接和硬件打交道，编写设备驱动程序需要掌握相关硬件的知识。不同的操作系统环境，驱动程序的结构也不同，因此还需要掌握操作系统核心与设备驱动程序之间的关系。

4.1.1 内核功能划分

Linux 内核 (kernel) 分成以下几个大的功能模块:

1. 进程管理

   进程管理功能负责创建和撤销进程以及处理它们和外部世界的连接 (输入和输出)。不同进程之间的通信是整个系统的基本功能，因此也由内核处理。

2. 内存管理

   用来管理内存的策略是决定系统性能的一个关键因素。内核在有限的可用资源上为每个进程创建了一个虚拟地址空间。内核的不同部分在和内存管理子系统交互时使用一套相同的系统调用，包括从简单的 `malloc()`、`free()` 到其他一些不常用的系统调用。

3. 文件系统

   Linux 中的每个对象几乎都可以被看作文件。内核在没有结构的硬件上构造结构化的文件系统，所构造的文件系统抽象在整个系统中广泛使用。

4. 设备控制

   几乎每个系统操作都会映射到物理设备上。除 CPU、内存以及其他很有限的几个实体以外, 所有设备控制操作都由与被控制设备相关的代码完成, 这段代码就叫做设备驱动程序。内核必须为系统中的每个外设嵌入相应的驱动程序, 这就是本章讨论的主题。

5. 网络功能

   大部分网络操作和具体进程无关, 因此必须由操作系统来管理。系统负责在应用程序和网络接口之间传递数据包, 并根据网络活动控制程序的执行。所有的路由和地址解析也都由内核解决。

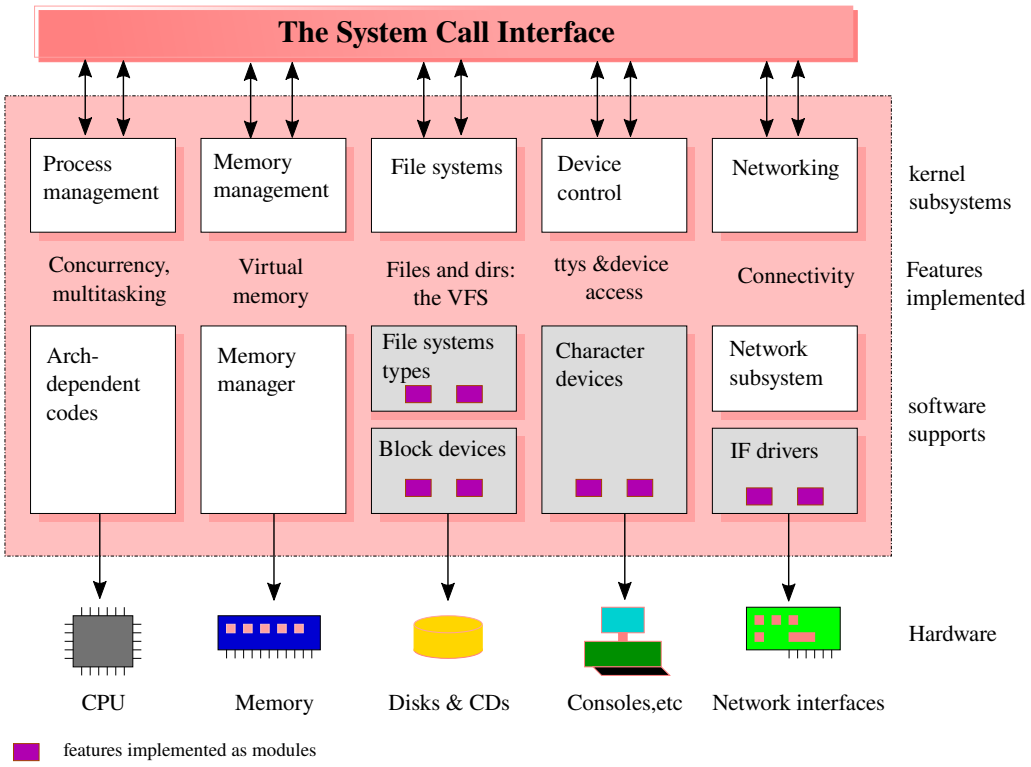


图 4.1: 内核功能划分

4.1.2 设备驱动程序的作用

设备驱动程序的作用在于提供机制 (mechanism), 而非提供策略 (policy)。驱动程序主要解决需要提供什么功能的问题, 而如何使用这些功能则应交给应用程序来处理。编写驱动程序时, 应该掌握这样一个基本原则: 编写访问硬件的内核代码时, 不要给用户程序强加任何策略。因为不同的用户有不同的需求。驱动程序应该处理如何使硬件可用的问题, 而将怎样使用硬件的问题留给上层应用。

如果从另一个角度来看驱动程序, 它可以被看作是应用和实际设备之间的一个软件层。对于相同的设备, 不同的驱动程序可以提供不同的功能。

实际上, 许多驱动程序是同用户程序一起发布的。这些用户程序主要用来帮助配置和访问目标设备。

4.1.3 设备和模块分类

系统运行时向内核中添加的代码被称为模块 (module)。Linux 内核支持几种不同类型的模块, 其中就包括驱动程序。每个模块是一段目标代码, 但不构成一个独立的可执行文件。将模块加入内核通过 `insmod` (**ins**tall **mod**ule) 命令, 将内核模块从内核中移除使用 `rmmod` (**rem**ove **mod**ule) 命令。如果没有内核模块的这种动态加载特性, 就不得不一次又一次重新编译生成一个庞大的内核镜像, 来加入新的功能, 而其中大量的模块可能是我们永远也用不到的。

图4.1列出了负责特定任务的几个不同类型的模块。

Linux 系统将设备分为三种类型: 字符设备 (char device)、块设备 (block device) 和网络接口 (network interface。每个模块通常只实现其中一种类型。相应地, 模块可分为字符模块、块模块和网络模块三种。

字符设备

字符设备是能够像字节流 (如文件) 一样被访问的设备, 由字符设备驱动程来实现这种特性。字符设备驱动通常至少需要实现 `open()`、`read()`、`write()` 和 `close()` 系统调用。字符终端和串口就是字符设备的两个例子, 它们能够用流抽象很好地表示。字符设备可以通过文件系统节点 (如上面的两个例子, 字符终端通过 /dev/console、串口设备通过 /dev/ttyS0 或 /dev/ttyS1 节点) 来访问。它和普通文件的唯一差别在于, 普通文件的访问通过前后移动访问指针来访问存储介质上不同位置上的数据, 而大多数字符设备只提供顺序访问数据的方式, 这是由设备的数据传输性质决定的。作为显示设备驱动的帧缓冲设备属于字符设备, 同样也可以通过移动指针实现数据访问。

块设备

和字符设备一样, 块设备也是通过 /dev 目录下的文件系统节点访问的。不同点在于块设备 (如磁盘) 上能容纳文件系统。Linux 允许应用程序像访问字符设备那样读写块设备, 可以一次传递任意多字节的数据。因而块设备和字符设备的区别仅仅在于内核内部管理数据的方式, 也就是内核和驱动程序的接口不同。块驱动程序除了给内核提供和字符驱动程序一样的接口以外, 还提供了专门面向块设备的接口。不过这些接口对于那些从 /dev 下某个目录项打开块设备的用户和应用程序都是不可见的。另外, 块设备的接口必须支持挂载文件系统。

网络接口

网络接口可以是一个硬件设备, 也可能是个纯软件设备, 如回环 (loopback) 接口。网络接口由内核中的网络子系统驱动, 负责发送和接收数据包。例如, TELNET 和 FTP 连接都是面向流的, 它们都使用了同一个设备, 但这个设备看到的只是数据包而不是独立的流。

由于不是面向流的设备, 因此将网络接口映射到文件系统中的节点比较困难。Linux 的做法是给它们分配一个唯一的名字, 如 eth0。但这个名字在文件系统中不存在对应的节点项。内核和网络驱动程序间的通信不同于内核和字符设备以及块设备驱动程序之间的通信方式。内核调用一套和数据包传输相关的函数, 即套接字 (socket)。

Linux 中还存在其他类型的驱动程序模块, 这些模块利用内核提供的公共服务来处理特定类型的设备。因此我们能够和通用串行总线 (USB) 模块、串口模块等通信。

4.2 构建和运行模块

4.2.1 第一个示例模块

我们以一个完整的模块为例来说明模块的用法。

清单 4.1: 第一个模块: example.c

```
/* example.c */
#include <linux/module.h>
#include <linux/kernel.h>
int init_module(void)
{
    printk("<1>Module installed.\n");
    return 0;
}
void cleanup_module(void)
{
    printk("<1>Module removed.\n");
}
```

函数 printk() 在内核中定义, 其功能和标准 C 库中的函数 printf() 几乎完全一样。内核不能使用 printf() 是因为它在运行时不能依赖于 C 库。模块只能访问内核的公共符号。代码中的字符串 $<1>$ 定义了这条消息的优先级。默认优先级的消息可能不会显示在控制台上。

4.2.2 模块的编译

开发内核模块和设备驱动, 最好有相应版本的内核源码, 至少也需要有与源码目录结构相同的头文件。如果是内核源码, 还需要对它进行一次正确的配置生成配置文件 ".config"。关于内核的配置方法, 请参考内核移植的相关内容。

编译清单 4.1 的模块使用下面的 Makefile:

清单 4.2: 编译模块的 Makefile

```
# Makefile (!not makefile)
ifneq ($(KERNELRELEASE),)
MODULE_NAME := example
$(MODULE_NAME)-y := example.o
obj-m    := $(MODULE_NAME).o
else
KDIR     := /lib/modules/`uname -r`/build
PWD      := $(shell pwd)
default:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

```
clean:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
endif
```

下面是对 Makefile 的解释:

- 编译模块和驱动的 GNU Make 脚本文件必须以 Makefile 命名, 不能使用 makefile 或 GNUmakefile;
- 定义 `MODULE_NAME` 为最后生成的模块文件名前缀。模块文件名后缀是 ".ko"。`$(MODULE_NAME)-y` 是构建模块的依赖文件列表。如果模块只由一个 C 文件源码构成, 第四行可以省去;
- `obj-m` 表示编译成模块, `obj-y` 表示编译进内核镜像。如果编译进内核, 必须与内核源码同时编译;
- 在这个 Makefile 里不需要设置编译器及编译选项, 这些内容均已在配置内核时设置好了。编译这个模块时会转到内核源码目录中执行编译 (GNU Make 的 "-C" 选项);
- 变量`KDIR` 必须设置指向正确的内核源码路径。通过软件仓库安装, 内核源码安装在 `/usr/src/linux-$VERSION`, 并通过符号链接到 `/lib/modules/$VERSION/build` 目录。命令`uname -r` 可以打印出当前运行的内核版本。因此上面的 Makefile 将编译出针对当前系统的模块。

  针对其他系统开发的模块或驱动, 包括面向嵌入式开发的交叉编译场合, `KDIR` 必须手工填写指向目标系统的内核源码路径。

  由于 Linux 内核的模块验证机制, 与当前运行系统版本不一致的模块不能加载。

将 Makefile 与 example.c 放在同一个目录。一切准备就绪, 在这个目录执行 `make` 命令, 如果没有错误, 将生成 example.ko, 这就是编译好的内核模块。

4.2.3 模块的运行

模块运行于内核空间, 一旦加载, 就拥有最高特权级。因此出于系统安全性的要求, 与模块相关的操作 (`insmod`、`rmmod`、`modprobe`) 被系统设置为必须以超级用户权限运行。

在终端模式中 [1] 通过命令 "`insmod example.ko`" 加载模块, 可以看到打印 "Module installed." 信息。`lsmod` 命令用于列出模块清单, 此时应可以看到模块 example 已存在。命令 "`rmmod example`" 将模块移除出内核, 同样也应可以看到打印出 "Module removed." 信息。

注意 `rmmod` 命令的参数是模块名, 不是文件名, 与 `insmod` 命令的参数形式要求不一样。

图形模式的终端可以用 `dmesg` 命令显示 `printk()` 的输出消息。由于不同系统传递消息机制的不同, 得到的输出结果可能不一样。开启守护进程 klogd 后, 在系统日志文件/var/log/messages 里面也可以找到相应的输出。

4.2.4 内核模块与应用程序

在继续学习之前, 有必要搞清楚内核模块和应用程序的不同。

[1]这里的终端模式不是图形界面下的一个 shell, 而是纯字符模式界面。在 PC 上, Linux 系统可以通过 Ctrl+Alt+F1 到 Ctrl+Alt+F6 切换出 6 个字符模式, Ctrl+Alt+F7 之后的几个用于图形模式。

模块的运行方式

应用程序从头至尾执行某个确定的任务, 而模块却不是一个独立的可执行程序, 因此上面的过程准确地说不能称作运行模块, 而只是模块的加载和卸载。函数 `init_module()` 是模块加载的入口点, 它将自己注册在内核中, 准备服务于将来的某个请求。它的任务是为以后调用模块函数预先做准备。这就像模块在说:“我在这儿, 我能做这些工作。” 这个函数很快就结束了, 而模块真正的运行是在今后服务于某个请求时。

模块的第二个入口点`cleanup_module()` 在模块卸载时被调用。它告诉内核:“我要离开了, 不要再让我做任何事情。”

模块使用的函数

应用程序可以调用它自身未定义的函数, 是因为链接过程可以解析外部引用, 从而使用适当的函数库。例如, 定义在 libc 中的 `printf()` 函数, 就是这种可被调用的函数之一。然而, 模块仅被链接到内核, 因此它能调用的函数仅仅是内核导出 (export) 的那些函数。`printk()` 就是由内核定义并导出给模块使用的一个 `printf()` 的内核版本。除了几个细小的差别之外, 它和 `printf()` 函数的功能类似, 最大的不同在于它缺乏对浮点数的支持。

因为模块不和 libc 等函数库连接, 因此在源文件中不能包含通常的头文件。内核模块只能使用作为内核一部分的函数。和内核相关的所有内容都在内核源代码的 include/linux 和 include/asm 目录下的头文件里声明。内核头文件中的许多声明仅仅与内核本身相关, 不应暴露给用户空间的应用程序。因此我们用 `#ifdef __KERNEL__` 块来保护这些声明。

模块运行在内核空间 (kernel space), 而应用程序运行在用户空间 (user space), 这个概念是操作系统理论的基础之一。每当应用程序执行系统调用或者被硬件中断挂起时, 操作系统将执行模式从用户空间切换到内核空间, 执行系统调用的内核代码运行在进程上下文中, 它代表调用进程执行操作, 因此能够访问进程地址空间的所有数据; 而处理硬件中断的内核代码则和进程是异步的, 因而与任何一个特定进程无关。

内核访问当前进程

内核当前执行的大多数操作和某个特定进程相关, 即当前进程。在一个系统调用执行期间, 例如 `open()` 或者 `read()`, 当前进程是指发出调用的进程。内核代码可以通过访问全局指针变量 `current` 来使用进程特定的信息[2]。`current` 是一个指向当前用户进程结构 `task_struct` 的指针。`current` 的具体实现细节对于内核中其他子系统是透明的。驱动程序只需要包含头文件 `linux/sched.h` 就可以引用当前进程。

从模块的角度来看, `current` 和 `printk()` 一样, 都是外部引用。如下面的语句通过访问 `task_struct` 结构的某些字段来输出当前进程的进程 ID 和命令名:

```
printk("The process is \"%s\" (pid %i)\n",
    current->comm, current->pid);
```

存储在 `current->comm` 中的命令名是当前进程所执行的程序文件的基名 (basename), 截短到 15 个字符。

[2]current 不是一个真正的全局变量。内核开发者将描述当前进程的结构隐藏在栈页 (stack page) 中, 从而优化了对该结构的访问。细节可以查阅 asm/current.h。

4.3 模块的结构

本节讨论一个模块的软件结构, 它是下面编写设备驱动程序的基础。

4.3.1 模块的初始化和清除函数

函数原型

初始化函数 `init_module()` 返回一个整型值。返回值为 0 表示模块成功加载, 返回值小于 0 则表示在初始化过程中有错误, 模块不加载进内核。由于该函数只有一次被调用的机会, 较新的内核为其增加了一个 `__init` 属性:

```
int __init init_module(void)
```

它表示无论模块是否加载成功, `init_module()` 这部分代码都不驻留在内存中。

清除函数 `cleanup_module()` 没有返回值。

函数重命名

如前所示, 内核调用 `init_module()` 来初始化一个刚刚加载的模块, 并在模块即将卸载之前调用 `cleanup_module()` 函数。所有模块都使用这样的函数名本身没有问题, 但从个性化方面考虑, 可以用下面两个宏给这个两个函数重新命名:

```
#include <linux/module.h>
module_init(user_init_func);
module_exit(user_cleanup_func);
```

它们通常与版权声明 `MODULE_LICENSE()` 一起写在源文件的末尾。注意, 要使用 `module_init()` 和 `module_exit()`, 则代码必须包含头文件 linux/module.h。

这样做的好处是内核中每个初始化和清除函数都有一个独特的名字, 给调试带来了方便, 同时也使那些既可以作为一个模块也可以直接链入内核的驱动程序更加容易编写。这两个宏对模块所做的唯一一件事情就是将 `__initcall()` 和`__exitcall()` 定义为给定函数的名字。

4.3.2 内核符号表

`insmod` 使用公共内核符号表解析模块中未定义的符号。公共内核符号表包含了所有的全局内核符号 (即函数和变量) 的地址。公共符号表可以从文件 /proc/ksyms 或 /proc/kallsyms 以文本格式读取。一旦模块装入内核, 它所导出的任何符号都变成了公共符号表的一部分。

新模块可以使用你的模块导出的符号, 你还可以在其他模块的上层堆叠新模块。模块堆叠技术在内核代码中很多地方都能看到, 例如 msdos 文件系统模块依赖于 fat 模块导出的符号; 而每个 USB 输入设备 (如 USB 键盘和鼠标) 模块堆叠在 usbcore 和 input 模块之上; USB 无线网卡则堆叠在 usbcore、mac80211 和网络协议上。

`modprobe` 是处理堆叠模块的一个实用工具。它的功能很大程度上和 `insmod` 类似, 但是它能自动解决指定模块的依赖关系, 同时加载所依赖的其他模块。因此, 一个 `modprobe` 命令有时候相当于若干次有序的 `insmod` 命令 (下层依赖模块未加载时, `insmod` 加载模块会失败)。`modprobe` 根据 /lib/modules 下面的 modules.dep 文件中建立的堆叠关系逐个加载模块, 文件 modules.dep 在编译内核时生成。

通常情况下, 模块只需要实现自己的功能, 不必导出符号。然而如果有其他模块需要使用模块导出的符号时, 可以使用下面的宏:

```
EXPORT_SYMBOL(name);
```

4.3.3 模块的卸载

使用计数

为了确定模块是否能安全卸载, 系统为每个模块保留一个使用计数器。系统需要这个信息来确定模块是否忙。在较新的内核版本中, 系统自动跟踪使用计数。模块列表命令 `lsmod` 打印的模块清单中, 第三个字段显示的是使用计数值。

```
...
i915                  1134592  4
i2c_algo_bit            16384  1 i915
drm_kms_helper         131072  1 i915
rtsx_pci                53248  2 rtsx_pci_ms,rtsx_pci_sdmmc
ahci                    36864  4
drm                    360448  5 i915,drm_kms_helper
e1000e                 237568  0
libahci                 32768  1 ahci
ptp                     20480  1 e1000e
pps_core                20480  1 ptp
video                   40960  2 i915,thinkpad_acpi
...
```

从上面的列表可以看到, 模块 i915 当前使用计数是 4, 原则上不能卸载。而模块 e1000e 使用计数是 0, 可以卸载。当模块 e1000e 卸载后, 模块 ptp 的使用计数也将变为 0。模块使用情况也可以通过 /proc/modules 文件查询。

卸载

使用 `rmmod` 可以卸载一个模块, 它调用系统调用 `delete_module()`。这个系统调用随后检查模块的使用计数, 如果为 0 则调用模块本身的 `cleanup_module()` 函数, 否则返回出错信息。

模块所注册的每一项功能都需要在函数 `cleanup_module()` 中注销, 模块导出的符号可以自动从内核符号表中删除。

4.3.4 资源使用

I/O 端口和 I/O 内存

典型驱动程序的最主要任务是读写 I/O 端口 (port) 和 I/O 内存。在初始化和正常运行时, 驱动程序都可能访问 I/O 端口和 I/O 内存。I/O 端口和 I/O 内存统称为 I/O 区域 (I/O region)。

设备驱动程序必须保证它对 I/O 区域的独占式访问, 以防止来自其他驱动程序的干扰。为了避免不同设备间的冲突, Linux 开发者实现了 I/O 区域的请求/释放机制。这种机制仅仅是一个帮助系统管理资源的软件抽象, 并不要求硬件支持。

/proc/ioports 和 /proc/iomem 文件以文本形式列出了已注册的资源。一个典型的 /proc/ioports 文件如下所示:

```
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial(set)
0300-031f : NE2000
0376-0376 : ide1
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(set)
1000-103f : Intel Corporation 82371AB PIIX4 ACPI
   1000-1003 : acpi
   1004-1005 : acpi
   1008-100b : acpi
   100c-100f : acpi
1100-110f : Intel Corporation 82371AB PIIX4 IDE
1300-131f : pcnet_cs
1400-141f : Intel Corporation 82371AB PIIX4 ACPI
1800-18ff : PCI CardBus #02
1c00-1cff : PCI CardBus #04
5800-581f : Intel Corporation 82371AB PIIX4 USB
d000-dfff : PCI Bus #01
   d000-d0ff : ATI Technologies Inc 3D Rage LT Pro AGP-133
```

文件中的每一项表示一个被某个驱动程序锁定或者属于某个硬件设备的端口范围。在传统的 ISA (**I**ndustral **S**tandard **A**rchitecture) 总线结构中, 系统新增一个设备并且通过跳线来选择一个 I/O 范围时, 这个文件可以用来避免端口冲突。用户检查已经使用的端口, 然后设置新设备使用一个空闲的 I/O 范围。尽管现在大部分的硬件不再使用跳线, 但这种方法在处理定制或者工业部件时仍然非常有用。

访问 I/O 注册表的编程接口有下面三个函数 [3]:

```
#include <linux/ioports.h>
int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start,
        unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
```

check_region() 用来检查是否可以分配某个端口范围。如果不可以, 则返回一个负数 (如 −EBUSY 或 −EINVAL)。request_region() 完成真正的端口范围分配, 成功则返回一个非空指针。请求成功后即可在 /proc/ioports 清单中看到。驱动程序并不需要使用或者保存这个返回的指针, 我们只检查它是否非空。

```
#include <linux/ioport.h>
#include <linux/errno.h>
static int ioport_detect(unsigned int port, unsigned int range)
{
    int err;
    if ((err = check_region(port, range)) < 0)
        return err;                          /* busy */
    if (ioport_probe_hw(port, range) != 0)
        return -ENODEV;                      /* not found */
    request_region(port, range, "driver"); /* never fail */
    return 0;
}
```

驱动程序所分配的所有 I/O 端口都必须在卸载驱动时释放, 避免给其他需要使用这个设备的驱动程序造成麻烦。

同 I/O 端口的情况类似, I/O 内存的信息可以从 /proc/iomem 文件中获得。下面是 PC 机上这个文件的部分内容:

```
    00000000-0009fbff : System RAM
    0009fc00-0009ffff : reserved
    000a0000-000bffff : Video RAM area
    000c0000-000c7fff : Video ROM
    000f0000-000fffff : System ROM
    00100000-03fefffe : System RAM
       00100000-0022c557 : Kernel code
       0022c558-0024455f : Kernel data
    20000000-2fffffff : Intel Corporation 440BX/ZX - 82443BX/ZX Host bridge
    68000000-68000fff : Texas Instruments PCI1225
    68001000-68001fff : Texas Instruments PCI1225 (#2)
```

[3]实际上, 它们是定义在 linux/ioports.h 中的宏。在内核中, 为提高效率, 很多功能都是通过宏定义或者内联 (inline) 函数实现的。本章对 “函数” 和 “宏” 的概念不严格区分, 重点关注它们的功能。

```
    e0000000-e3ffffff : PCI Bus #01
    e4000000-e7ffffff : PCI Bus #01
       e4000000-e4ffffff : ATI Technologies Inc 3D Rage LT Pro AGP-133
       e6000000-e6000fff : ATI Technologies Inc 3D Rage LT Pro AGP-133
    fffc0000-ffffffff : reserved
```

就驱动程序编程而言, I/O 内存的注册方式与 I/O 端口类似, 因为它们机制相同。驱动程序使用下列函数调用来获得和释放对某个 I/O 内存区域的访问:

```
int check_mem_region(unsigned long start, unsigned long len);
int request_mem_region(unsigned long start, unsigned long len,
             char *name);
int release_mem_region(unsigned long start, unsigned long len);
```

典型的驱动程序通常事先知道它的 I/O 内存范围。因此相对于前面 I/O 端口的请求方法, 对 I/O 内存的请求缩减为如下几行代码:

```
static int iomem_register(unsigned int mem_addr, unsigned int mem_size)
{
    if (check_mem_region(mem_addr, mem_size)) {
        printk("drivername: memory already in use\n");
        return -EBUSY;
    }

    request_mem_region(mem_addr, mem_size, "drivername");
}
```

自动和手动配置

有些设备, 除了 I/O 地址以外可能还有其他参数会影响驱动程序的行为, 如设备品牌和发行版本号。给驱动程序设置参数值 (即配置驱动程序) 有两种方法: 一种是由用户显式指定, 另一种是驱动程序自动检测。在很多时候, 这两种方法都是混合使用的。

参数值可由 `insmod` 或 `modprobe` 在装载模块时设置, `modprobe` 默认从 /etc/modules.conf 文件中获得参数赋值。因此, 如果模块需要获得一个名为 `timer_init` 的整型参数, 可以在装载模块时这样使用 `insmod` 命令:

```
# insmod timer.ko timer_init=3600
```

在 insmod 能够改变模块参数之前, 模块必须能够访问这些参数 (通常是全局变量)。参数由 linux/moduleparam.h 中的宏 `module_param()` 声明, 它带有三个参数: 变量名、变量类型和在 sysfs 中可见的读写权限。上面的参数可以使用下面几行语句来声明:

```
    int timer_init = 0;
    module_param (timer_init, int, 0666);
```

模块参数支持的变量类型有 byte、short、ushort、int、uint、long、ulong、charp (字符指针)、bool (0/1,Y/N) 和 invbool (反 bool, N=true)。

所有模块参数都应该被赋予一个默认值, 用户可以使用 `insmod` 来显式地改变。模块通过和默认值比较来确定显式参数值, 避免在加载模块不提供参数时导致不确定结果。

4.4 字符设备驱动程序

本节的目标是编写一个完整的字符设备驱动程序。它的驱动对象是 PC 中的一个可编程定时器 (timer)。在传统的 PC 中, 定时器的功能是由芯片 Intel8254 实现的, 目前该芯片在 PC 中已经不再独立存在, 其功能和其他可编程芯片一同被集成到一个大规模集成电路中。但编程控制方法仍与 Intel8254 兼容 [4]。

为便于叙述, 我们将下面讨论的驱动程序命名为 timer。

4.4.1 timer 的设计

编写驱动程序的第一步就是定义驱动程序为用户程序所提供的能力 (机制)。

Intel8254 内部包含三个可独立编程控制的 16 位减法计数器/定时器。在早期的 PC 中, 定时器 0 用于系统日时钟计时, 定时器 1 用于 DMA 刷新时钟, 定时器 2 用于驱动 PC 的扬声器。三个定时器统一使用 1.19MHz 的时钟, 编程方法完全一样。系统为 Intel8254 分配的 I/O 地址是 0x40–0x43, 在 8086 时代是系统的核心定时设备。如今它早已失去了往日的地位, 在 PC 中是一个极普通的设备, 可能仅仅是出于兼容性而保留着。

设置 Intel8254 的工作方式通过向控制寄存器中写入一个字节的方式控制字完成。图 4.2 是控制字各位的含义。

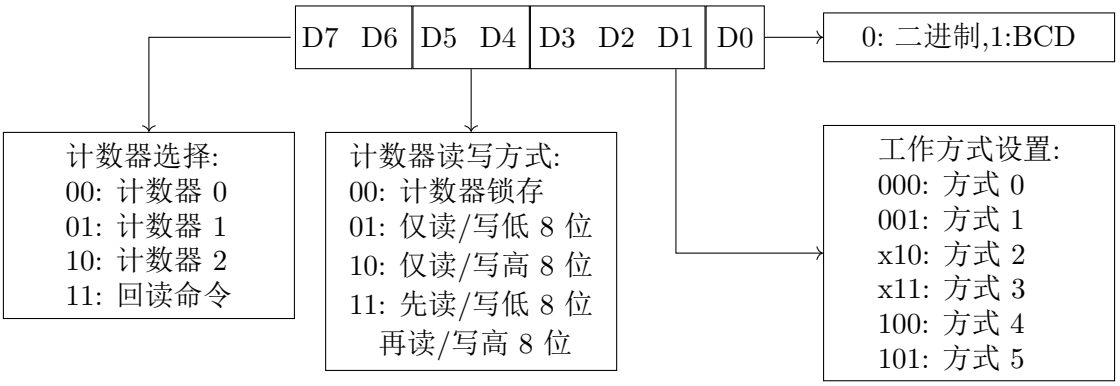


图 4.2: Intel8254 方式控制字格式

主设备号与次设备号

访问字符设备要通过文件系统内的设备名称进行。那些文件被称为特殊文件, 或设备文件, 或者简单称之为文件系统树的节点。根据 FHS规范, 它们应位于 /dev 目录中。字符设备驱动程序的设备文件可通过 `ls -l` 命令输出的第一列中的字母 `c` 来识别。块设备文件也在 /dev 下, 但它们是由字符 `b` 标识的。本章主要关注字符设备, 不过, 下面介绍的许多内容同样也适用于块设备。

[4]早期的 PC 使用 Intel8253, 除了工作频率稍低以外, 它与后代产品的最大不同是没有控制寄存器的回读功能。

当我们在 /dev 目录执行 `ls -l` 命令时, 就可以在设备文件项的最后修改日期前看到两个用逗号分隔的数字。如果是普通文件, 这个位置显示的是文件的长度。而对于设备文件来说, 这两个数就是相应设备的主设备号 (major number) 和次设备号 (minor number)。下面的列表给出了典型系统中的一些设备。它们的主设备号是 1、4、7 和 10, 而次设备号有 1、3、5、64 和 129。

```
crw-rw-rw-    1 root     root       1,   3 Feb 23  1999 null
crw-------    1 root     root      10,   1 Feb 23  1999 psaux
crw-------    1 root     tty        4,   1 Aug 16 22:22 tty1
crw-rw-rw-    1 root     dialout    4,  64 Jun 30 11:19 ttyS0
crw-rw-rw-    1 root     dialout    4,  65 Aug 16 00:00 ttyS1
crw-------    1 root     sys        7,   1 Feb 23  1999 vcs1
crw-------    1 root     sys        7, 129 Feb 23  1999 vcsa1
crw-rw-rw-    1 root     root       1,   5 Feb 23  1999 zero
```

主设备号表示设备对应的驱动程序。例如, /dev/null 和 /dev/zero 由驱动程序 1 管理, 而虚拟控制台和串口终端由驱动程序 4 管理 [5]; 类似地, vcs1 和 vcsa1 设备都由驱动程序 7 管理。内核利用主设备号在系统调用 `open()` 操作中将设备与相应的驱动程序对应起来。次设备号只是由那些主设备号已经确定的驱动程序使用, 内核的其他部分不会用到它。一个驱动程序控制多个设备是常有的事情。例如, 一块硬盘被格式化成四个分区, 在不同的分区上, 读写硬盘物理存储数据的方法是完全一样的, 没必要写四个驱动程序。类似地, 系统中有 2 个串行接口设备, 它们唯一的差别只是端口地址不同, 也不需要分别写两个驱动程序。次设备号为驱动程序提供了一种区分不同设备的方法。

注册主设备号

向系统增加一个新的驱动程序意味着为其分配一个主设备号。这个分配工作在驱动程序 (模块) 初始化函数中进行, 由下面的函数实现:

```
#include <linux/fs.h>
int register_chrdev_region(dev_t first, unsigned count,
                           const char *name);
int alloc_chrdev_region(dev_t *dev, unsigned firstminor,
                        unsigned count, const char *name);
```

函数的第一个参数是请求分配的第一个设备号 (主设备号和次设备号), 参数 `count` 是请求分配的连续设备号的数量, 通常每个设备驱动只需要一个主设备号 (少数驱动程序可能需要多个主设备号)。`name` 是设备的名称, 可以使用任意字符串作为设备名。该名称将出现在 /proc/devices 中, 仅仅具有文字识别意义, 不承担任何其他功能。返回值为 0 时表示分配成功。

主设备号是用来索引字符设备静态数组的一个小整数。长期以来 Linux 系统中的主设备号一直被限制在 1 到 255 之间。

`alloc_chrdev_region()` 用于动态注册设备, 成功返回后应从参数 `dev` 的返回值中分解出主设备号和次设备号以备后用。

[5]驱动程序本身不存在编号问题。这里所谓的 “驱动程序 1”、“驱动程序 4” 意指主设备号 1、4 所对应的驱动程序。

一部分主设备号已经静态分配给了大部分常用设备。内核源代码树中的 Documentation/devices.txt 文件列出了这些设备的清单。在编写设备驱动时应注意回避这些设备号。

接下来的问题就是如何给程序一个名字。通过这个名字, 程序才可向设备驱动程序发出请求, 并与驱动程序的主设备号和次设备号相关联。这个名字就是设备文件名 (或称设备节点)。在文件系统上创建一个设备节点的命令是 mknod:

```
# mknod /dev/timer0 c 123 0 -m 666
```

上面的命令创建一个字符设备 ("c"), 主设备号是 123, 次设备号是 0。次设备号应该在 0–255 的范围内, 由于历史原因, 次设备号存储在单个字节中 [6]。最后一个选项-m 表示创建设备的读写操作权限。由于 mknod 以超级用户权限运行, 默认创建出设备的权限属性可能会限制普通用户的访问。

一旦通过 mknod 创建了设备文件, 该文件就将一直保留下来, 除非显式地将其删除 (删除操作与普通文件一样)。这一点与存储在磁盘上的其他信息类似 [7]。

手工创建设备节点比较麻烦。Linux 系统通过一个用户空间的软件 udev 管理设备节点。它通过一些脚本文件描述设备驱动与设备文件的关系, 这些文件位于 /etc/udev/ 目录。当内核检测到有设备驱动状态发生变化时, 会根据脚本触发相应的动作, 创建或删除设备文件。

一旦分配了主设备号, 就可以从 /proc/devices 中读取得到。如下是该文件的内容:

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
123 timer
180 usb
  ...
```

卸载设备驱动

从系统中卸载模块时必须注销主设备号。这一操作可以在模块的清除函数中调用下面的函数完成:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

[6]新的 Linux 内核中, linux/kdev_t.h 文件中定义了 MINORBITS 作为次设备号的取值范围。

[7]启用了 devpts 的情况除外。此时 /dev 并不在磁盘上而是通过伪文件系统也就是在 RAM 中实现的, 断电后信息不再保存。

卸载驱动程序后, 通常还需要删除设备文件。

4.4.2 文件操作

打开的设备在内核内部由 file 结构标识。内核使用 file_operations 结构访问驱动程序的函数。file_operations 结构是一个定义在 linux/fs.h 中的函数指针数组。每个文件都与它自己的函数集相关联。这些操作主要负责系统调用的实现, 并因此被命名为 open()、read() 等。被定义为 file_operations 结构的指针 fops 中, 每一个字段指向驱动程序中实现特定操作的函数。对于不支持的操作, 对应的字段可被置为 NULL。

下面是 file_operations 结构中常用的操作。各操作的返回值为负数时说明发生了错误。

- 打开设备

```
int (*open) (struct inode *, struct file *);
```

该函数用于打开设备文件。通常打开设备文件意味着下面将要使用这个设备, 因此这里需要对设备做一些初始化工作。如果设备工作正常, 函数返回 0。此时内核会给调用这个函数的接口分配一个正的文件描述符 (注意系统调用 open() 的返回值和这里的返回值不同)。若该函数在 file_operations 结构中不存在 (NULL), 则只要设备文件满足权限要求, 打开操作总是成功的。

- 释放设备

```
int (*release) (struct inode *, struct file *);
```

当 file 结构被释放时, 将调用这个操作。它通常对应 close() 系统调用, 但只有最后一个设备被关闭时才真正释放设备。与 open() 类似, release() 在 file_operations 结构中也可以不存在。

- 读写文件

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```

read() 和 write() 分别用来从设备中读取数据和向设备发送数据, 返回成功读写的字节数。它们分别对应应用程序中的系统调用 read() 和 write()。

- 文件定位

```
loff_t (*llseek) (struct file *, loff_t, int);
```

方法 llseek() 用来修改文件的当前读写位置, 并将新位置作为返回值返回。

- 事件查询

```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

方法 poll() 是 poll() 和 select() 这两个系统调用的后端实现。

- I/O 控制

```
int (*ioctl) (struct inode *, struct file *,
              unsigned int, unsigned long);
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
```

系统调用 ioctl() 提供了一种执行设备特定的命令的方法。早期内核的方法是 ioctl()。从 2.6.36 开始, ioctl() 不再使用, 而是被 unlocked_ioctl() 替代, 同时增加了一个 compat_ioctl() 用于在 64 位内核中的 32 位系统调用。用户空间的系统调用 ioctl() 形式保持不变。

- 存储器映射

```
int (*mmap) (struct file *, struct vm_area_struct *);
```

mmap() 用于请求将设备内存映射到进程地址空间。如果设备没有实现这个方法, 则返回 −ENODEV。

以上很多函数和系统调用名称完全相同, 关系也非常密切: 系统调用的实现最后将落实在驱动程序的方法函数上。为区别起见, 以下将内核实现的函数称为 “方法”, 将用户空间的函数称 “系统调用”。

本设备驱动程序所实现的只是最重要的设备方法, 并且采用标记化格式声明它的 file_operations 结构:

```
/* timer.c */
struct file_operations fops = {
    llseek: timer_llseek,
    read: timer_read,
    write: timer_write,
    unlocked_ioctl: timer_ioctl,
    open: timer_open,
    release: timer_release,
};
```

将 file_operations 结构加入设备驱动的方法是:

```
/* timer.c */
...
    struct cdev *my_cdev = cdev_alloc();
    my_cdev->owner = THIS_MODULE;
    my_cdev->ops = &fops;
    cdev_init(my_cdev, &fops);
    cdev_add(my_cdev, dev, 1);
```

cdev_*() 一组函数的格式是:

```
#include <linux/cdev.h>
void cdev_init(struct cdev *dev, const struct file_operations *fops);
struct cdev *cdev_alloc(void);
int cdev_add(struct cdev *dev, dev_t num, unsigned count);
void cdev_del(struct cdev *dev);
```

以上函数的参数 dev 是 cdev 结构, 内核用该结构表示字符设备的特征, 其中一个重要的成员结构是 ops, 用于指向 file_operations 文件操作结构指针。num 是与设备驱动相关联的第一个设备号, count 是与设备驱动相关联的设备数目。

在较老版本的内核中, 设备的注册和注销是通过下面的函数实现的:

```
int register_chrdev(unsigned int major, const char *name,
        struct file_operations *fops);
void unregister_chrdev(unsigned int major, const char *name);
```

注册函数的返回值提示操作成功还是失败。负的返回值表示错误。函数的参数意义如下:

- 参数 major 是被请求的主设备号。在注册设备时, 设置一个大于 0 的 major 值, 内核按这个数字为驱动程序分配主设备号。如果成功, 函数返回 0; 而将 major 设为 0 时, 内核将寻找一个空闲的数字为驱动动态分配一个主设备号, 返回值就是分配成功的主设备号。动态分配的缺点是, 由于分配的主设备号不能保证始终一致, 所以无法预先创建设备节点。
- name 是设备的名称, 该名称将出现在 /proc/devices 中, 作为对设备的识别特征。在注销设备时, 该参数格式上要求与注册时用到的名称一致, 但并不强制要求。
- fops 是指向 file_operations 结构的指针, 该结构包含了设备驱动中对设备的所有操作方法, 是调用驱动程序的入口点。

一旦将驱动程序注册到内核表中, 它的操作就和指定的主设备号对应了起来。当我们在与主设备号对应的字符设备文件上进行某个操作时, 内核将从 file_operations 结构中找到并调用正确的函数。出于这个原因, 传递给 cdev_init() 或者 register_chrdev() 的 file_operations 结构指针应指向驱动程序中的一个全局数据结构, 而不是模块初始化函数 init_module() 中的局部数据结构。

4.4.3 打开设备

每次在内核调用一个驱动程序时, 它都会告诉驱动程序它正在操作哪个设备。应用程序系统调用 open() 打开一个设备文件, 这个设备文件包含了主设备号和次设备号的信息。系统调用向驱动程序传递两个参数: struct inode 指针和 struct file 指针。

inode 结构

inode 结构指针的 i_rdev 字段保存了主次设备号。

每次在内核调用一个驱动程序时, 它都会告诉驱动程序它正在操作哪个设备。主次设备号合在一起构成单个数据类型并用来标识特定的设备, 它保存在索引节点 (inode) 结构的 i_rdev 字段中。因此可通过 inode->i_rdev 得到设备号。设备号保存在 kdev_t 这个数据类型中。kdev_t 定义在 linux/kdev_t.h 中并被 linux/fs.h 包含。下面的这些宏和函数是可以对 kdev_t 进行的操作:

```
#include <linux/fs.h>
/* Extract the major number from a kdev_t structure. */
MAJOR(kdev_t dev);
/* Extract the minor number. */
MINOR(kdev_t dev);
/* Create a kdev_t built from major and minor numbers. */
MKDEV(int major, int minor);
```

驱动程序实际上完全不关心被打开设备的名字, 它仅仅知道设备号。两个主/次设备号完全相同的设备文件对设备驱动程序来说没有任何区分意义。用户可以利用这一点为设备取一个别名或创建链接。

file 结构

在 linux/fs.h 中定义的 **struct file** 是设备驱动程序所使用的另一个重要的数据结构。file 结构代表一个打开的文件。它由内核在 open() 时创建, 并传递给在该文件上进行操作的所有函数, 直到最后的 close() 函数。

在内核源代码中, 指向 **struct file** 的指针通常写成 file 或 filp。写成后者的目的是为了避免和这个结构本身的名称混淆。下面是 file 结构中一些重要的字段:

- mode_t f_mode

  通过 FMODE_READ 和 FMODE_WRITE 位来指明文件的读写属性。在 ioctl() 方法函数里可能需要检查这个字段以获知读/写允许, 但在 read() 和 write() 里不需要。因为内核在进入这些函数时已经检查过了。

- loff_t f_pos

  当前读写的位置。loff_t 是一个 64 位的值 (**long long**)。驱动程序如果需要了解当前处在文件的什么位置, 就可以读这个值, 但不应该改变它。read() 和 write() 过程应该用它们作为最后一个参数接收到的指针对其更新。

- **unsigned int** f_flags

  文件标志。例如 O_RDONLY、O_NONBLOCK、O_SYNC。驱动程序在非阻塞操作中需要检查这些标志, 其他标志很少用到。特别地, 读/写允许应该用 f_mode 而不是 f_flags。

- **struct** file_operations *f_op

  与文件相关的操作。内核给它一个指针, 作为打开其实现过程的部分, 然后在需要对这些操作进行分支的时候再读取它。filp->f_op 里的值从不保留给以后的引用, 这意味着你任何时候都可以改变与文件相关联的文件操作, 再返回调用之后, 新方法立即生效。例如, 用打开主设备号为 1 的设备文件 (/dev/null, /dev/zero 等等) 的代码代之以对 filp->f_op 的操作, 对 filp->f_op 的操作依赖于被打开的次设备号。这样的实用性允许若干个过程在同一个主设备号下实现, 而不需要在每个系统调用前引进。文件操作的替代能力是内核面向对象程中的方法覆盖。

- **void** *private_data

  在调用 open() 方法前这个指针指向 NULL。驱动程序可以自由使用或忽略这个字段。驱动可以用来指向已定位的数据, 但在文件结构被内核销毁之前, release() 方法中必须释放它。private_data 是一个保留跨系统调用状态信息的有用的资源。

打开设备的工作

open() 方法允许驱动程序对设备进行一些初始化, 从而为以后的操作做准备。此外, open() 一般还会递增设备的使用计数, 防止在文件关闭前模块被卸载出内核, 这个计数值在 release() 方法中被递减。在较老版本的内核中, 这个计数器由程序维护, 目前已由内核统一管理, 驱动程序不需要直接访问。

在大部分驱动程序中, open() 完成如下工作:

- 检查设备特定的错误;

- 如果设备是首次打开, 则对其初始化;
- 识别次设备号, 更新f_op指针;
- 分配并填写 filp->private_data 里的数据结构。

当驱动程序需要在方法函数之间传递参数时, 利用 filp->private_data 可以避免定义全局变量。

驱动程序 timer 以定时器在 Intel8254 中的序号为次设备号。数据寄存器的地址就是 0x40 加上次设备号, 驱动程序 timer 对每个定时器操作的唯一不同就是寄存器地址的差别。

一个稍复杂的设备可以对每个次设备定义一个特定的 file_operations 结构, 它在 open() 操作时赋给 filp->f_op。下面的代码显示了多个 fops 是如何实现的:

```
struct file_operations *device_fop_array[] = {
    &device_fops,      /* 类型0 */
    &device_priv_fops, /* 类型 1 */
    &device_pipe_fops, /* 类型 2 */
    &device_sngl_fops, /* 类型 3 */
    &device_user_fops, /* 类型 4 */
    &device_wusr_fops, /* 类型 5 */
};
#define MAX_TYPE 5
/* 在 open() 方法实现中, device_fop_array数组根据TYPE(dev)的
   值来决定对其数组成员的引用 */
    ...
    int type = TYPE(inode->i_rdev);
    if (type > MAX_TYPE)
        return -ENODEV;
    filp->f_op = device_fop_array[type];
```

内核根据主设备号调用 open() 方法, device 则使用上述宏分解出的次设备号。TYPE 用来索引 device_fop_array 数组。根据次设备号判断设备的类型, 并把 filp->f_op 赋给由设备类型所决定的 file_operations 结构。然后在新的 fops 中声明的 open() 方法将得到调用。

timer 的功能比较简单, 它只需要分离出计数器的序号, 并分配一块内存用于和应用程序之间的数据交换空间。为此我们定义一个数据结构:

```
/* timer.h */
...
struct dtimer {
    int port;   /* Address of Intel8254 counter register */
    char buf[1024];
};
```

timer_open() 的实际代码如下:

```
/* timer.c */
...
int timer_open(struct inode *inode, struct file *filp)
{
    struct dtimer *i8254;

    /* Timer */
    int port = MINOR(inode->i_rdev);

    if (!filp->private_data) {
        filp->private_data =
              kmalloc(sizeof(struct dtimer), GFP_KERNEL);
        i8254 = (struct dtimer *)filp->private_data;
        i8254->port = port + 0x40;

        filp->f_pos = 0;
    } else {
    }

    return 0; /* success */
}
```

在内核中内存分配和释放可以使用下面的函数:

```
#include <linux/slab.h>
void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *);
```

open() 方法中为 filp->private_data 分配的内存应该在设备关闭的时候使用 kfree() 释放。

释放设备

release() 方法的作用正好与 open() 相反。常常实现这个方法的函数名被写成 device_close() 这样的形式而不是 device_release()。无论是哪种形式, 这个设备方法都应该完成下面的任务:

- 释放由 open() 分配的、保存在 filp->private_data 中的所有内容;
- 在最后一次关闭操作时关闭设备。

同 open() 方法一样, 在 2.4 之前的内核版本中还需要对使用计数器进行递减操作。

如果某个时刻, 一个尚未被打开的文件被关闭了, 计数将如何保持一致呢? 提出这个问题并不奇怪, dup() 和 fork() 都会在不调用 open() 的情况下创建已打开文件的副本, 但每一个副本都会在程序终止时被关闭。例如, 大多数程序从来不打开标准输入输出文件 (或设备), 但它们都会在终止时关闭它。

Linux 的内核是这样做的: 并不是每个 close() 系统调用都会引起对 release() 方法的调用。仅仅是那些真正释放设备数据结构的 close() 调用才会调用这个方法, 这正是内核中方法函数名字是 release() 而不是 close() 的原因。

内核维护一个 file 结构被使用多少次的计数器。无论是 fork() 还是 dup() 都不创建新的数据结构, file 结构仅由 open() 方法创建。它们只是增加已有结构中的计数。只有在 file 结构的计数归零时, close() 系统调用才会执行 release() 方法, 而这只发生在 file 结构被删除时。驱动程序中的 release() 方法与 close() 系统调用间的关系保证了模块使用计数的一致性。

注意: flush() 方法在应用程序每次调用 close() 时都会被调用。不过很少有驱动程序去实现 flush(), 因为在 close() 时并没有什么事情需要去做, 除了调用 release() 以外。

即使应用程序未调用 close() 显式地关闭它所打开的文件就终止, 以上的讨论同样适用, 原因是, 内核在进程退出的时候会通过内部使用 close() 系统调用自动关闭所有相关的文件。

读/写方法

读/写方法完成的任务是相似的, 即拷贝数据到应用程序空间, 或从应用程序空间拷贝数据。它们的原型相似, 如下所示:

```
ssize_t read(struct file *filp, char *buff,
             size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char *buff,
              size_t count, loff_t *offp);
```

参数 filp 是文件指针, count 是请求传输的数据长度, buff 是指向用户空间缓冲区的指针, 最后的 offp 是一个指向 long offset (长偏移量类型) 对象的指针, 这个对象指明用户在文件中进行存取操作的位置。

和这两个设备方法相关的主要问题是需要在内核地址空间和用户地址空间传输数据。我们不能沿用写应用程序的习惯, 使用指针或者 memcpy() 来完成这样的操作。由于许多原因, 不能在内核空间中直接使用用户空间地址。

内核空间地址与用户空间地址之间很大的一个差异就是, 用户空间的内存是可被换出的。当内核访问用户空间指针时。相对应的页面可能已经不在内存中了, 这样的话就会产生页面失效。

在 Linux 系统中, 此类跨空间的拷贝是由一些特定的函数完成的, 它们在 linux/uaccess.h 中定义。这样的拷贝或者通过一般的 (如 memcpy()) 函数完成, 或者通过为特定的数据大小做了优化的函数来完成。

驱动程序中的 read() 和 write() 代码要做的工作, 就是在用户地址空间和内核地址空间之间进行数据的拷贝。这种能力是由下面的内核函数提供的, 它们用于拷贝任意的一段字节序列。这也是每个 read() 和 write() 方法实现的核心部分:

```
#include <linux/uaccess.h>
unsigned long copy_to_user(void *to, const void *from,
       unsigned long count);
unsigned long copy_from_user(void *to, const void *from,
        unsigned long count);
```

这两个函数的作用并不限于拷贝数据, 它们还检查用户空间的指针是否有效。如果指针无效, 就不会进行拷贝; 如果在拷贝过程中遇到无效地址, 则仅仅会复制有效部分的数据。函数的返回值是尚未拷贝完的内存数据字节数。如果不需要检查用户空间指针, 则可以调用下面的两个函数:

```
unsigned long __copy_to_user(void *to, const void *from,
       unsigned long count);
unsigned long __copy_from_user(void *to, const void *from,
        unsigned long count);
```

无论这些方法传输了多少数据, 一般都应更新 *offp 所表示的文件位置。大多数情况下, offp 参数就是指向 filp->f_pos 的指针。

图 4.3 表明了一个典型的 read() 实现是如何使用其参数的。

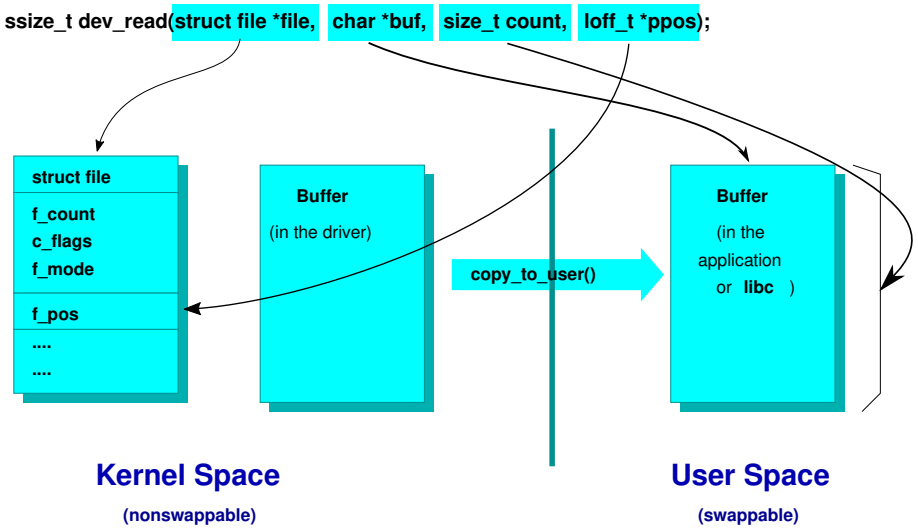


图 4.3: 系统调用参数的传递

调用程序对 read() 的返回值解释如下:

- 如果返回值等于传递给 read() 系统调用的 count 参数, 说明所请求的字节数传输成功完成了。这是最理想的情况;
- 如果返回值是小于 count 的正数, 说明只有部分数据传送成功;
- 如果返回值是 0, 则表示已经达到了文件尾;
- 返回值负数意味着发生了错误, 通过对该值的分析可以获知发生了什么错误。linux/errno.h 中定义了错误标识的宏。

timer 代码利用了部分读取的规则, 每次只提供最多 1024 字节数据。

```
/* timer.c */
...
ssize_t timer_read(struct file *filp, char *buffer,
                   size_t count, loff_t *f_pos)
{
    struct dtimer *i8254 = (struct dtimer *)filp->private_data;
```

```
    int port = i8254->port;
    int retval;

    if(count > 1024)
        count = 1024;

    insb(port, i8254->buf, count);

    retval = copy_to_user(buffer, i8254->buf, count);
    *f_pos += count;
    return count - retval;
}
```

与 read() 类似, write() 方法也有相对应的返回值规则, 它们与 read() 的返回值意义基本相同。

由于 Intel8254 每个计数器的数据寄存器只有 16 位, 最多只允许写入两个字节, 后写入的数据将覆盖之前的数据。当工作方式设置为仅高字节或仅低字节读写时, 写入计数器数据寄存器的值只有最后写入的一个字节生效, 因此在 write() 方法中只处理用户空间传来的最后两个字节。

```
/* timer.c */
...
ssize_t timer_write(struct file *filp, const char *buffer,
                size_t count, loff_t *f_pos)
{
    struct dtimer *i8254 = (struct dtimer *)filp->private_data;
    int port = i8254->port;
    int i, len;
    char localbuf[2];

    len = count;
    if (count > 2)
        count = 2;
    copy_from_user(localbuf, &buffer[len - count], count);

    for (i = 0; i < count; i++)
        outb(localbuf[i], port);

    *f_pos += count;
    return count;
}
```

llseek 实现

`llseek()` 方法实现了 `lseek()` 和 `llseek()` 系统调用 (glibc 中没有实现 `llseek()` 的系统调用)。如果定位操作对应于设备的一个物理操作, 或者要实现基于文件尾的定位, 你可能就需要提供自己的 `llseek()` 方法。在 timer 驱动程序中简单实现了定位功能:

```
/* timer.c */
...
loff_t timer_llseek(struct file *filp, loff_t off, int whence)
{
    loff_t newpos;
    switch(whence) {
        case SEEK_SET:
            newpos = off;
            break;
        case SEEK_CUR:
            newpos = filp->f_pos + off;
            break;
        case SEEK_END:
            newpos = 1024 + off;
            break;
        default:     /* can't happen */
            return -EINVAL;
    }
    if (newpos < 0)
        return -EINVAL;

    if (newpos >= 1024)
        newpos = 0;   /* rewind */

    filp->f_pos = newpos;
    return newpos;
}
```

4.4.4 I/O 控制

除了读写设备之外, 设备驱动程序还需要提供各种各样的硬件控制能力。这些控制操作一般都是通过 `ioctl()` 方法来支持的。`ioctl()` 系统调用为设备驱动程序执行 “命令” 提供了一个设备特定的入口点。与 `read()` 等方法不同, `ioctl()` 是设备特定的, 它允许应用程序访问被驱动硬件的特殊功能, 如配备设备、进入或退出某种操作模式等。这些控制操作不能简单地通过 `read()/write()` 文件操作来完成。例如, 串行设备控制器有数据寄存器和控制寄存器, 向串口写入的数据通过数据寄存器发送, 它不能用于写控制寄存器来

改变波特率和数据格式。ioctl() 的作用就在于此: 控制 I/O 通道。

在驱动程序 timer 中, 改变每个定时器的工作方式, 需要将一个具有特定格式的数据写到 Intel8254 的控制寄存器。这项工作不应与写到数据寄存器的初始化计数值相混淆, 因此也需要通过 ioctl() 实现。

在用户空间内调用的 ioctl() 函数一般具有如下原型:

```
int ioctl(int fd, int cmd, ...);
```

原型中的 "..." 并不是数目不定的一串参数, 而只是一个可选参数, 习惯上定义为字符指针。这里用 "..." 只是为了在编译时防止编译器进行类型检查。第三个参数的具体形式依赖于要完成的控制命令, 也就是第二个参数。

另一方面, 设备驱动程序的 ioctl() 方法, 是按照如下原型获取其参数的:

```
int (*unlocked_ioctl) (struct file *filp,
            unsigned int cmd, unsigned long arg);
```

参数 cmd 由用户空间不经修改地传递给驱动程序, 可选的 arg 参数无论用户程序使用的是指针还是整数值, 它都以 unsigned long 的形式被传递给驱动程序。如果调用程序没有传递第三个参数, 那么驱动程序所接收的 arg 没有任何意义。

在编写 ioctl() 方法代码之前, 需要选择对应不同命令 cmd 的编号。理论上命令号可以是任意 32 位的整数。为方便程序员创建唯一的 ioctl() 命令号, Linux 内核采用结构化的 IOCTL 格式定义 ioctl() 命令号 (请阅读内核帮助文档 ioctl-number.txt)。它将命令号分解成四个字段:

1. type(类型)

   选择一个 8 位 (_IOC_TYPEBITS) 的魔数 (magic number) 作为类型码, 并在整个驱动程序中统一使用这个数字。
2. number(号码)

   序数, 在其他字段都相同的情况下由它来进行区分命令的功能。它的位宽也是 8 位 (_IOC_NRBITS)。
3. direction(方向)

   如果该命令有数据传输, 它就定义数据传输的方向。可以使用的值有: _IOC_NONE (不产生数据传输)、_IOC_READ、_IOC_WRITE 和_IOC_READ | _IOC_WRITE。它在命令编号中占 2 位。
4. size(大小)

   所涉及的用户数据大小, 占据剩余的 14 位。

头文件 asm/ioctl.h 定义了如下一些可以用于构造命令号的宏:

- _IO(type, nr): 无数据传输
- _IOR(type, nr, size): 读数据
- _IOW(type, nr, size): 写数据
- _IOWR(type, nr, size): 读写数据

另外, 以下宏用于解码 IOCTL 命令, 提取相关的字段:

- _IOC_DIR(nr): 提取方向标志位
- _IOC_TYPE(nr): 提取类型 (魔数)
- _IOC_NR(nr): 提取命令序号

- `_IOC_SIZE(nr)`: 提取数据大小

下面是驱动程序 timer 中的一些 IOCTL 命令定义。这些命令用来设置和获取驱动程序的配置参数。

```
/* timer.h */
...
/* Use 'x' as a magic number */
#define IOCTL_MAGIC 'x'

#define TIMER_IOCRESET  _IO(IOCTL_MAGIC, 0)
/*
 * S means "Set" through a pointer
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
 * X means "eXchange": G and S atomically
 * H means "sHift": T and Q atomically
 */
#define TIMER_IOCSBYTE  _IOW (IOCTL_MAGIC, 1, int)
#define TIMER_IOCSMODE  _IOW (IOCTL_MAGIC, 2, int)
#define TIMER_IOCTBYTE  _IO  (IOCTL_MAGIC, 3)
#define TIMER_IOCTMODE  _IO  (IOCTL_MAGIC, 4)
#define TIMER_IOCGBYTE  _IOR (IOCTL_MAGIC, 5, int)
#define TIMER_IOCGMODE  _IOR (IOCTL_MAGIC, 6, int)
#define TIMER_IOCQBYTE  _IO  (IOCTL_MAGIC, 7)
#define TIMER_IOCQMODE  _IO  (IOCTL_MAGIC, 8)
#define TIMER_IOCXBYTE  _IOWR(IOCTL_MAGIC, 9, int)
#define TIMER_IOCXMODE  _IOWR(IOCTL_MAGIC, 10, int)
#define TIMER_IOCHBYTE  _IO  (IOCTL_MAGIC, 11)
#define TIMER_IOCHMODE  _IO  (IOCTL_MAGIC, 12)
#define TIMER_IOCHNONE  _IO  (IOCTL_MAGIC, 13) /* debugging tool */
#define TIMER_SPEAKERON _IO  (IOCTL_MAGIC, 14) /* turn speaker on */
#define TIMER_IOC_MAXNR 14

#define TIMER_HIGH    (0b00100000)    /* access counter MSB */
#define TIMER_LOW     (0b00010000)    /* access counter LSB */
```

以上定义的命令宏也应对用户空间的应用程序可见。

从调用方的观点 (即从用户空间) 来看, 传送和接收参数有下面的六种途径:

清单 4.3: 用户空间调用 ioctl() 的六种途径

```
int cword;

ioctl(fd, TIMER_IOCSMODE, &cword);
ioctl(fd, TIMER_IOCTMODE, cword);
ioctl(fd, TIMER_IOCGMODE, &cword);
cword = ioctl(fd, TIMER_IOCQMODE);
ioctl(fd, TIMER_IOCXMODE, &cword);
cword = ioctl(fd, TIMER_IOCHMODE, cword);
```

尽管根据已有的约定, `ioctl()` 应该使用指针完成数据交换。我们可以尝试使用更多的方法实现整数参数的传递—通过指针和通过显式的数值。同样, 返回值也有两种方法: 通过 `ioctl()` 第三个参数指针或通过设置返回值。此外, 写入新的参数, 同时返回旧的参数, 也是 `ioctl()` 一种有用的功能, 在我们的驱动程序里把它叫做 “Exchange” 和 “Shift”, 分别用 `TIMER_IOCXMODE` 和 `TIMER_IOCHMODE` 实现。

在驱动程序 timer 中, 我们打算通过 `ioctl()` 读写 Intel8254 的控制寄存器。写控制寄存器格式已在图 4.2 中说明。对控制寄存器的操作包含三部分的内容: 1. 计数器操作方式 (8 位读写还是 16 位读写[8]); 2. 工作方式 (方式 0–5); 3. 计数值格式 (二进制还是十进制)。驱动程序只设置工作方式字的计数器读写方式 (图 4.2 中 D5、D4 两位) 和计数器工作方式 (图 4.2中的 D3、D2、D1 三位), 对方式控制字的最低一位仅考虑 D0=0 一种情况。表4.1 是写回读命令的格式。

表 4.1: Intel8254 回读命令格式

D7	D6	D5	D4	D3	D2	D1	D0
1	1	$\overline{\text{CNT}}$	$\overline{\text{ST}}$	CNT2	CNT1	CNT0	0

当回读命令 D5 为 0 时, 将锁存 D3~D1 指定的计数器, 以便随后正确读取 [9]。驱动程序暂不打算实现这个功能。

回读命令第四位 D4 为 0 时, 锁存状态信息。随后从对应计数器寄存器读到的值称为状态字, 其最低 6 位对应该计数器当前的工作方式 (与图 4.2 的低 6 位含义相同)。

驱动程序 timer 实现对状态字操作的代码在 `ioctl()` 方法中。

当一个指针指向用户空间时, 必须确保指向的用户地址是合法的, 而且对应的页面也已经映射。内核 2.2.x 及以后版本的地址验证是通过函数 `access_ok()` 实现的:

```
#include <linux/uaccess.h>
int access_ok(int type, const void *addr, unsigned long size);
```

第一个参数应该是 `VERIFY_READ` 或 `VERIFY_WRITE`, 取决于要执行的动作是读还是写用户空间内存区。`addr` 参数是用户空间地址, `size` 是字节数。例如, 如果 `ioctl()` 要从用户空间读一个整数, `size`就是 `sizeof(int)`。如果在指定的地址处既要读又要写, 则应该用 `VERIFY_WRITE`, 写允许的时候, 读允许的条件一定是满足的。

[8]由于 Intel8254 的每个计数器都是 16 位的, 但数据缓冲器只有 8 位, 因此读写 16 位寄存器时必须分两次写入高、低字节。

[9]由于读计数器需要对 16 位寄存器的高低 8 位分两次读取。如果计数器不锁存, 当计数值变化于临界状态时 (如低字节为 0x00, 再经一个计数周期将变为 0xff, 而高字节也发生了变化), 连续两次读取后拼接出的 16 位数字会导致错误。

与大多数函数不同, access_ok() 返回一个布尔值 1 表示访问成功, 0 表示访问失败。如果失败, 则驱动程序通常要返回 −EFAULT 给调用者。

timer 的源代码在 **switch** 语句前, 通过分析 IOCTL 编码的位字段来检查参数:

```
/* timer.c */
...
long timer_ioctl(struct file *filp, unsigned int cmd,
              unsigned long arg)
{
    int err = 0, tmp;
    int ret = 0;
    /*
     * extract the type and number bitfields, and don't decode
     * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
     */
    if (_IOC_TYPE(cmd) != IOCTL_MAGIC)
        return -ENOTTY;
    if (_IOC_NR(cmd) > TIMER_IOC_MAXNR)
        return -ENOTTY;
    /*
     * the direction is a bitmask, and VERIFY_WRITE catches R/W
     * transfers. 'Type' is user-oriented, while
     * access_ok is kernel-oriented, so the concept of "read" and
     * "write" is reversed
     */
    if (_IOC_DIR(cmd) & _IOC_READ)
        err = !access_ok(VERIFY_WRITE, (void *)arg, _IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
        err = !access_ok(VERIFY_READ, (void *)arg, _IOC_SIZE(cmd));
    if (err)
        return -EFAULT;
    ...
```

在调用 access_ok() 之后, 驱动程序可以安全进行实际的数据传送了。除了 copy_from_user() 和 copy_to_user() 函数外, 还可以使用已经为最常用的数据大小优化过的一组函数:

```
#include <linux/uaccess.h>
int put_user(datum, ptr);
int __put_user(datum, ptr);
int get_user(local, ptr);
int __get_user(local, ptr);
```

这些宏把数据写到用户空间, 或从用户空间接收数据保存在局部变量 local中。它们的执行效率相对高一些。如果要传递单个数据时, 应该使用它们而不是 copy_to/from_user() 一组。被传送的字节数依赖于 sizeof(*ptr)。返回值为 0 表明操作成功。与 copy_to/from_user() 一组类似, __put_user() 和 __get_user() 不检查用户空间指针。

timer 的 ioctl() 实现中只传递设备的可配置参数:

```
/* timer.c */
...
long timer_ioctl(struct file *filp, unsigned int cmd,
              unsigned long arg)
{
    ...
    i8254 = (struct dtimer *)filp->private_data;
    port = i8254->port;
    portnum = (port & 3) << 6;
    cntbit = 2 << (port & 3)
    outb(0b11100000 | cntbit, 0x43);  /* 发送状态字回读命令 */
    oldstatus = inb(port);         /* 读状态字 */

    switch(cmd) {
    case TIMER_IOCRESET:      /* 复位, 扬声器关闭 */
        outb(0x0, 0x61);
        ret = 0;
        break;
    case TIMER_SPEAKERON:     /* 扬声器打开 */
        outb(0x3, 0x61);
        ret = 0;
        break;
    /* 以下用于工作方式控制 */
    case TIMER_IOCSMODE:      /* Set: 参数来自指针 */
        ret = get_user(status, (int *)arg);
        status = (status << 1) | portnum;
        oldstatus &= (TIMER_HIGH|TIMER_LOW);
        status |= oldstatus;
        outb(status, port);
        break;
    case TIMER_IOCTMODE:      /* Tell: 参数来自变量 */
        status = (arg << 1) | portnum;
        oldstatus &= (TIMER_HIGH|TIMER_LOW);
```

```
        status |= oldstatus;
        outb(status, port);
        ret = 0;
        break;
    case TIMER_IOCGMODE:     /* Get: 通过指针返回 */
        oldstatus &= 0b00001110;
        oldstatus >>= 1;
        ret = put_user(oldstatus, (char *)arg);
        break;
    case TIMER_IOCQMODE:     /* Query: 通过函数返回值 */
        oldstatus &= 0b00001110;
        ret = oldstatus >> 1;
        break;
    case TIMER_IOCXMODE:     /* eXchange: 新旧参数通过指针交换 */
        ret = get_user(status, (int *)arg);
        if (ret == 0) {
            tmp = (oldstatus & 0b00001110) >> 1;
            ret = put_user(tmp, (int *)arg);
            oldstatus &= (TIMER_LOW|TIMER_HIGH);
            status = (status << 1) | portnum;
            status |= oldstatus;
            outb(status, port);
        }
        break;
    case TIMER_IOCHMODE: /* sHift: Tell + Query */
        status = (arg << 1) | portnum;
        status |= (oldstatus & (TIMER_HIGH|TIMER_LOW));
        outb(status, port);
        oldstatus &= 0b00001110;
        ret = oldstatus >> 1;
        break;

    /* 以下用于高低字节控制 */
    case TIMER_IOCSBYTE:
        ret = get_user(status, (int *)arg);
        status |= portnum;
        oldstatus &= 0b00001110;
        status |= oldstatus;
        outb(status, port);
```

```
        break;
    case TIMER_IOCTBYTE:
        status = arg | portnum;
        oldstatus &= 0b0001110;
        status |= oldstatus;
        outb(status, port);
        ret = 0;
        break;
    case TIMER_IOCGBYTE:
        oldstatus &= (TIMER_HIGH|TIMER_LOW);
        ret = put_user(oldstatus, (char *)arg);
        break;
    case TIMER_IOCQBYTE:
        ret = oldstatus & (TIMER_HIGH|TIMER_LOW);
        break;
    case TIMER_IOCXBYTE:
        ret = get_user(status, (int *)arg);
        if (ret == 0) {
            tmp = oldstatus & (TIMER_HIGH|TIMER_LOW);
            ret = put_user(tmp, (int *)arg);
            oldstatus &= 0b00001110;
            status |= portnum;
            status |= oldstatus;
            outb(status, port);
        }
        break;
    case TIMER_IOCHBYTE:
        status = arg | portnum;
        status |= (oldstatus & 0b00001110);
        outb(status, port);
        ret = oldstatus & (TIMER_HIGH | TIMER_LOW);
        break;
    default:
        return -ENOTTY;
    }
    return ret;
}
```

程序中, 控制扬声器的两个 I/O 控制命令 TIMER_IOCRESET 和 TIMER_SPEAKERON 基于图 4.4 的电路。在 PC 中, Intel8254 计数器 2 的门控开关来自另一个可编程 I/O 设备 8255A

B 口的最低位 PB0, 高电平使能计数器; 8255A 的 B 口 I/O 地址为 0x61, 已初始化为输出功能, 只需要将该位置 1 即可; 同时, B 口次低位 PB1 控制扬声器开关。当 B 口最低 2 位为 1 时, 计数器的方波输出可以让扬声器发声。

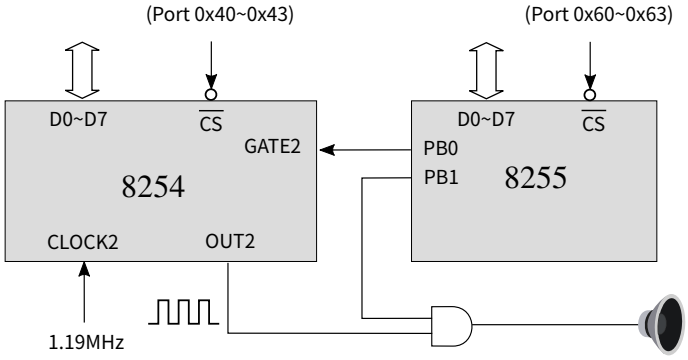


图 4.4: PC 扬声器发声逻辑图

4.4.5 阻塞型 I/O

睡眠和唤醒

当进程等待一个事件 (如数据到达或其他进程终止) 时, 它应该进入睡眠。睡眠使该进程暂时挂起, 腾出处理器给其他进程使用。在将来的某个时间, 等待的事件发生了, 进程被唤醒并继续执行。

Linux 中有几种处理睡眠和唤醒的方法, 每一种分别适合于不同的需求。不过所有方法都要处理同一个基本数据类型: 等待队列 (`wait_queue_head_t`)。确切地说, 等待队列 (wait queue) 其实是由正等待某事件发生的进程组成的一个队列。下面是等待队列的声明和初始化部分:

```
#include <linux/wait.h>
wait_queue_head_t my_queue;
    ...
init_waitqueue_head (&my_queue);
```

如果一个等待队列被声明为静态的, 可以在编译时对它进行初始化:

```
DECLARE_WAIT_QUEUE_HEAD (my_queue);
```

一旦声明了等待队列, 并完成了初始化, 进程就可以使用它进入睡眠。根据睡眠深度的不同, 可调用 `wait_event()` 的不同变体宏来进入睡眠 [10]。

```
wait_event(wait_queue_head_t queue, int condition);
```

把进程放入这个队列睡眠, 等待条件 `condition` 为真。如果进程所等待的事件永远也不发生, 进程就醒不过来了。

```
wait_event_interruptible(wait_queue_head_t queue, int condition);
```

除了睡眠可以被信号中断以外, 该变体工作方式和 `wait_event()` 类似。

[10] 2.4 内核版本中使用 sleep_on() 的一组函数

```
wait_event_timeout(wait_queue_head_t queue,
        int condition, long timeout);
wait_event_interruptible_timeout(wait_queue_head_t queue,
        int condition, long timeout);
```

与前面相似, 不同之处是到了指定时间 timeout 后就不再睡眠。timeout 以 jiffies 为单位 (见 4.8.1 节)。

驱动程序开发人员原则上应该使用这些函数/宏中的可中断的形式。

相应的, 内核提供的用来唤醒进程的高级函数有:

```
#include <linux/wait.h>
/* 唤醒等待队列中的所有进程 */
void wake_up(wait_queue_head_t *queue);
/* 只唤醒可中断睡眠的进程 */
void wake_up_interruptible(wait_queue_head_t *queue);
```

作为使用等待队列的一个例子, 想像一下当进程读设备时进入睡眠而在其他人写设备时被唤醒的情景。下列代码完成此事:

清单 4.4: 可中断睡眠 sleepy.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/wait.h>

#define DEVICE_NAME "sleepy"
#define MAJOR_NUM 234

static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

ssize_t sleepy_read(struct file *filp,
                    char *buffer,
                    size_t length,
                    loff_t *offset)
{
    printk("Process %i (%s) is going to sleep.\n",
           current->pid, current->comm);
    wait_event_interruptible(wq, flag!=0);
    flag = 0;
    printk("Awoken %i (%s) .\n", current->pid, current->comm);
    return 0;
```

```
}

ssize_t sleepy_write(struct file *filp,
                     const char *buffer,
                     size_t length,
                     loff_t *offset)
{
    printk("Process %i (%s) is awakening the readers...\n",
           current->pid, current->comm);
    flag = 1;
    wake_up_interruptible(&wq);
    return length;
}

struct file_operations fops = {
    read : sleepy_read,
    write: sleepy_write
};

int init_module()
{
    int major;
    major = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);
    if (major == 0) {
        printk("Device registed.\n");
        return 0;
    } else
        return -1;
}

void cleanup_module()
{
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
}
```

测试睡眠和唤醒

编译程序 4.4, 加载 sleepy.ko, 创建一个主设备号为 234 的设备文件 /dev/sleepy。在一个终端中输入命令 `cat /dev/sleepy` 时, 命令被挂起, 内核打印 “Process XXX (cat) is going to sleep.” 的消息 (`cat` 命令导致内核 `read()` 方法被调用)。从另一个终端输入

`echo "0" >/dev/sleepy` 将结束之前的 `cat` 命令, 并可以看到内核打印 “Awoken XXX (cat) .” 消息, 进程 `cat` 被唤醒 (`echo` 命令的输出重定向导致内核 `write()` 方法被调用)。

阻塞型和非阻塞型操作

在分析 `read()` 和 `write()` 方法之前, 我们先要了解 `filp->f_flags` 中的 `O_NONBLOCK` 标志的作用。这个标志在 linux/fcntl.h 中定义, 而此头文件自动被包含在 linux/fs.h 中。

这个标志的名字取自 “非阻塞打开”(open-nonblock) [11]。这个标志在默认时要被清除, 因为等待数据的进程一般只是睡眠。

执行阻塞型操作时, 应该实现下列动作以保持和标准语义一致:

- 如果一个进程调用了 `read()`, 但是还没有数据可读, 此进程就必须阻塞。数据一旦到达, 进程随即被唤醒, 并把数据返回给调用者。
- 如果一个进程调用了 `write()`, 但缓冲区没的空间, 此进程就必须阻塞。而且必须睡眠在与读进程不同的等待队列上。当向硬件设备写入一些数据, 从而腾出了部分输出缓冲区后, 进程即被唤醒, `write()` 调用成功。

如果指定了 `O_NONBLOCK` 标志, `read()` 和 `write()` 的行为就会有所不同。如果在数据没有就绪时调用 `read()` 或是在缓冲区没有空间时调用 `write()`, 则该调用简单地返回 −EAGAIN。

非阻塞型操作会立即返回, 使得应用程序可以查询数据。

4.5 设备驱动程序的使用

4.5.1 驱动程序与应用程序

在计算机应用系统中, 设备驱动为应用程序提供访问设备的接口, 它必须通过应用程序发挥作用。根据前面提到的设计原则, 设备驱动程序本身不实现设备的具体功能。例如, 在编写声卡的设备驱动时, 我们不应考虑是用它来播放 MP3 文件还是 WMA 文件, 而只需要考虑如何用它将数字量转换成模拟量。至于数字量是通过 MP3 格式的解码还是 WMA 格式的解码得到, 则应交给应用程序完成。

应用程序访问设备驱动的唯一途径是设备文件, 实现设备功能最终将落实在对设备端口的读写操作上。一个标准的应用程序形式通常包含下面的代码:

```
int fd = open("/dev/device", O_RDWR);
/* get "len" bytes from device. */
n = read(fd, buf, len);
...
/* send "len" bytes to device. */
n = write(fd, buf, len);
```

如果驱动程序通用性好, 上面的 `read()`、`write()` 系统调用也可以使用系统的基本命令实现。例如, `cat` 命令会读取文件 (包括设备文件), 将设备文件作为输出重定向目标则会调用 `write()` 函数。如果没有对输入/输出数据的特殊处理要求, 无须特地编写程序, 直接使用系统命令就可以实现设备的功能。

[11] 内核和一些应用软件中可能还会有 `O_NDELAY` 标志的引用, 这是 `O_NONBLOCK` 的另一个名字, 它是为保持和 System V 代码的兼容性而设计的。

4.5.2 内核源码中的模块结构

本章讨论的模块 (或设备驱动) 是独立于内核之外编写的, 并未纳入官方维护的内核源码。内核开发人员会将具有一定普遍意义的、其重要性值得为内核引入的代码加入内核树, 为新的内核版本增加新的特性。内核源码中的配置文件 Kconfig 为扩展内核功能提供了接口。

配置文件 Kconfig 包含为内核提供的模块配置选项和其他目录的配置。其他目录的配置通过 “source” 包含另一个 Kconfig 完成。

清单 4.5: drivers/char/Kconfig

```
#
# Character device configuration
#

menu "Character devices"
...
source "drivers/tty/Kconfig"
...
config TTY_PRINTK
    tristate "TTY driver to output user messages via printk"
    depends on EXPERT && TTY
    default n
    ---help---
      If you say Y here, the support for writing user messages (i.e.
      console messages) via printk is available.

      The feature is useful to inline user messages with kernel
      messages.
      In order to use this feature, you should output user messages
      to /dev/ttyprintk or redirect console to this TTY.

      If unsure, say N.
...
endmenu
```

“config” 定义模块的名称, 该模块名加上前缀 “CONFIG_” 后被添加到最后生成的配置文件 “.config” 中, 作为定义变量提供给 Makefile。如果模块的属性是 bool, 该变量的值有两个选择: “y” 或 “n”, 分别表示 “是” 与 “否”; 如果模块的属性是 tristate, 则它有三个选择: “y”、“n” 或 “m”, “m” 表示 “模块”。实际上 “n” 并不出现在 “.config” 中, 而是在 “CONFIG_” 前面加上 “#” 符号将其注释掉, 相当于这个变量不存在。help 下面的文字会出现在配置内核的帮助界面中, 帮助用户理解该模块的作用。

源码树中每一个子目录的 Makefile 通过定义一系列的 obj-$(CONFIG_*) 变量告诉内核编译器, 哪些模块应编入内核 (obj-y), 哪些模块应编译成模块 (obj-m), 编译时依赖哪

些文件。顶层 Makefile 会根据各级子目录 Makefile 定义的变量形成最后的编译规则。

4.5.3 将模块加入内核

如果需要将我们自己编写的模块加入内核, 需要做下面的工作:

1. 确定模块的名称, 避免与内核已经定义的模块宏名冲突。这里我们将模块宏定义为 MYDEVICE。
2. 修改 Makefile, 以适应内核树的格式。

   将清单4.2 中的 `obj-m` 改成 `obj-$(CONFIG_MYDEVICE)`, 并根据模块的性质, 选择一个适当的目录, 将我们自己编写的模块 (或称设备驱动) 目录整体复制到该目录下。
3. 添加 Kconfig

```
#
# Add DEVICE to kernel source tree
#
menu "My Device"        # 向上级菜单提示

config MYDEVICE         # 对应 Makefile 中 CONFIG_MYDEVICE
   tristate "my device" # tristate or bool
   default m
   help
       This module will be added to kernel
endmenu
```

4. 上级 Makefile 增加 CONFIG_MYDEVICE 选项, 使其指向该模块目录:

   obj-$(CONFIG_MYDEVICE) += mydevice/
5. 上级 Kconfig 增加入口:

   source ”drivers/char/mydevice/Kconfig”

   这里假定将我们编写的模块目录 mydevice 置于内核源码的 drivers/char 子目录中。

完成上述工作后, 在内核的配置选项 My Device 就会出现在 Character Devices 菜单中。

4.6 调试技术

内核是一个比较特殊的程序, 它与特定的进程无关, 不宜使用普通的调试器跟踪和调试。内核的错误常常会导致系统宕机, 调试器也无能为力。本节讨论一些监控内核代码运行的方法, 试图跟踪可能出现错误的场合。

4.6.1 输出调试

printk

通过附加不同日志级别 (loglevel), 或者说消息优先级, 可让 `printk()` 根据这些级别所标示的严重程度, 对消息进行分类。一般采用宏来指示日志级别。例如, `KERN_INFO`, 我们在前面已经看到它被添加在一些打印语句的前面, 就是一个可以使用的消息日志级别。日

志级别宏展开为一个字符串, 在编译时由预处理器将它和消息文本拼接在一起。这也就是为什么下面的例子中优先级和格式字串间没有逗号的原因。

下面有两个 printk() 的例子, 一个是调试信息, 一个是临界信息:

```
printk(KERN_DEBUG "Here I am: %s: %i\n", __FILE__,
       __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

在头文件 linux/kernel.h 中定义了八种可用的日志级别字符串。

KERN_EMERG 用于紧急事件消息, 它们一般是系统崩溃之前提示的消息;

KERN_ALERT 用于需要立即采取动作的情况;

KERN_CRIT 临界状态, 通常涉及严重的硬件或软件操作失败;

KERN_ERR 用于报告错误状态。设备驱动程序会经常使用这个级别来报告来自硬件的问题;

KERN_WARNING 对可能出现问题的情况进行警告, 这类情况通常不会对系统造成严重问题;

KERN_NOTICE 有必要进行提示的正常情形。许多与安全相关的状况用这个级别进行汇报;

KERN_INFO 提示性信息。很多驱动程序在启动的时候, 以这个级别打印出它们找到的硬件信息;

KERN_DEBUG 用于调试信息。每个字符串 (以宏的形式展开) 代表一个尖括号中的整数, 整数值的范围从 0 到 7。数值越小, 优先级越高。

没有指定优先级的 printk() 语句默认采用的级别是 DEFAULT_MESSAGE_LOGLEVEL, 这个宏在文件 kernel/printk/printk.c 中指定为一个整数值。在内核源码配置选项菜单 Kernel hacking -> printk and dmesg options 可以看到这样的选项:

```
[*] Show timing information on printks
(7) Default console loglevel (1-15)
(4) Default message log level (1-7)
[*] Delay each boot printk message by N milliseconds
[ ] Enable dynamic printk() support
```

这个配置下的内核, 默认的消息日志级别是 4。根据日志级别, 内核可能会把消息打印到当前控制台上。这个控制台可以是一个字符模式的终端、一个串口打印机或是一个并口打印机。如果优先级小于控制台日志级别 console_loglevel 这个整数值的话 (上面选项中同时也设定了控制台日志级别是 7), 消息才能显示出来。如果系统同时运行了 klogd 和 syslogd, 则无论为何值, 内核消息都将追加到 /var/log/messages 中, 而与控制台日志级别的设置无关。如果 klogd 没有运行, 这些消息就不会传递到用户空间, 这种情况下, 就只好查看 /proc/kmsg 了。

变量 console_loglevel 的初始值是 DEFAULT_CONSOLE_LOGLEVEL, 而且还可以通过 sys_syslog() 系统调用进行修改。调用 klogd 时可以指定 "-c" 开关选项来修改这个变量, klogd 的帮助手册对此有详细说明。如果要修改它的当前值, 必须先终止守护进程 klogd, 再用 "-c" 选项重新启动它。

如果在控制台上工作, 而且常常遇到内核错误的话, 就有必要降低日志级别。因为出错处理代码会把 console_loglevel 增为它的最大数值, 导致随后的所有消息都显示在控制台上。如果需要查看调试信息, 就有必要提高日志级别。这在远程调试内核, 并且在交互会话未使用文本控制台的情况下是很有帮助的。

proc 文件系统的 /proc/sys/kernel/printk 提供了读取和修改控制台的日志级别的功能。这个文件包含 4 个整数值, 分别对应控制台的当前日志级别、默认日志级别、最小允许日志级别和系统启动时的默认级别。可以通过简单地输入下面的命令使所有的内核消息得到显示:

```
# echo 8 > /proc/sys/kernel/printk
```

注意, 不能使用普通的文本编辑器修改 proc 系统上的文件。

到此应该解释了为什么在本章的第一个范例 (清单 4.1) 中使用 “< 1 >” 这个标记。它确保这些消息能在控制台上显示出来。

消息如何被记录

printk() 函数将消息写到一个长度为 __LOG_BUF_LEN (定义在 kernel/printk/printk.c 中) 字节的循环缓冲区中, 然后唤醒任何正在等待消息的进程, 即那些睡眠在 syslog 系统调用上的进程, 或者读取 /proc/kmsg 的进程。这两个访问日志引擎的接口几乎是等价的, 不过请注意, 对 /proc/kmsg 进行读操作时, 日志缓冲区中被读取的数据就不再保留, 而 syslog 系统调用却能随意地返回日志数据, 并保留这些数据以便其他进程也能使用。一般而言, 读 /proc 文件要容易些, 这使它成为 klogd 的默认方法。

手工读取内核消息时, 在停止 klogd 之后, 可以发现 /proc 文件很像一个 FIFO, 读进程会阻塞在里面以等待更多的数据。显然, 如果已经有 klogd 或其他的进程正在读取相同的数据, 就不能采用这种方法进行消息读取, 因为会与这些进程发生竞争。

如果循环缓冲区填满了, printk() 就绕回缓冲区的开始处填写新数据, 覆盖最陈旧的数据, 于是记录进程就会丢失最早的数据。但与使用循环缓冲区所带来的好处相比, 这个问题可以忽略不计。例如, 循环缓冲区可以使系统在没有记录进程的情况下照样运行, 同时覆盖那些不再会有人去读的旧数据, 从而使内存的浪费减到最少。Linux 消息处理方法的另一个特点是, 可以在任何地方调用 printk(), 甚至在中断处理函数里也可以调用, 而且对数据量的大小没有限制。这个方法的唯一缺点就是可能丢失某些数据。

klogd 运行时, 会读取内核消息并将它们分发到 syslogd。syslogd 随后查看 /etc/syslog.conf, 找出处理这些数据的方法。syslogd 根据设施和优先级对消息进行区分。这两者的允许值均定义在 sys/syslog.h 中。内核消息由 LOG_KERN 设施记录, 并以 printk() 中使用的优先级记录 (例如, 在 printk() 中使用的 KERN_ERR 对应于 syslogd 中的 LOG_ERR)。如果没有运行 klogd, 数据将保留在循环缓冲区中, 直到被某个进程读取或缓冲区溢出为止。

如果想避免因为来自驱动程序的大量监视信息而扰乱系统日志, 则可以为 klogd 指定 “-f” 选项 (file), 指示 klogd 将消息保存到某个特定的文件, 或者修改 /etc/syslog.conf 来适应自己的需求。另一种可能的办法是采取强硬措施: 停止 klogd 守护进程, 而将消息详细地打印到空闲的虚拟终端上。

开启和关闭消息

在驱动程序开发过程中, printk() 为调试和测试代码带来了很大的方便。但当程序正式发布时, 可能需要删除这些打印语句, 以降低系统的资源消耗, 提高性能, 减小代码规模。但如果代码需要增加新的功能继续升级, 或是发现了其中的错误需要修改时, 有可能希望恢复之前删除的打印消息的功能。下面是解决这个问题的一个思路。

这里给出了一个编写 printk() 调用的方法, 可以单独或者全部地对它们进行开关。这个技巧是定义一个宏, 在需要时, 这个宏被展开为一个 printk() 或 printf() 调用。改变这个宏的名字 (只要增减一个字母) 将使宏展开失效。这种技巧不限于内核空间, 用户空间的程序同样也可以借鉴。

下面这些来自 timer.h 的代码实现了这些功能。

```
/* timer.h */
...
#undef PDEBUG             /* undef it, just in case */
#ifdef TIMER_DEBUG
#  ifdef __KERNEL__
    /* This one if debugging is on, and kernel space */
#  define PDEBUG(fmt, args...) \
      printk( KERN_DEBUG "timer: " fmt, ## args)
#  else
    /* This one for user space */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...)   /* nothing: it's a placeholder */
```

符号 PDEBUG 依赖于是否定义了 TIMER_DEBUG, 它能根据代码所运行的环境选择合适的方式显示信息: 内核态运行时使用 printk() 系统调用; 用户态下则使用 libc 调用 fprintf(), 向标准错误设备进行输出。符号 PDEBUGG 则什么也不做, 它可以用来将打印语句注释掉, 而不必把它们完全删除。由于预处理程序的条件语句 (以及代码中的常量表达式) 只在编译时执行。要再次打开或关闭消息必须重新编译。

另一种方法就是在程序中使用条件判断语句, 它在运行时执行, 因此可以在程序运行期间打开或关闭打印功能, 无须重新编译。这一功能带来的问题是, 每次代码执行时系统都要进行额外的处理, 甚至在消息关闭后仍然会影响性能。由此引入的处理器分支指令, 有时候给处理器带来的开销还是比较可观的。

每一个驱动程序都会有自身的功能和监视需求。良好的编程技术在于选择灵活性和效率的最佳折衷点。

4.6.2 查询调试

由于 syslogd 会一直保持对其输出文件的同步刷新, 每打印一行都会引起一次磁盘操作, 因此大量使用 printk() 会严重降低系统性能。从 syslogd 的角度来看, 这样的处理是正确的。它试图把每件事情都记录到磁盘上, 以防系统万一崩溃时, 最后的记录信息能反应崩溃前的状况。然而, 因处理调试信息而使系统性能减慢, 是我们所不希望的。

驱动程序开发人员对系统进行查询时, 可以采用两种主要的技术: 在 PROC 文件系统

中创建文件, 或者使用驱动程序的 ioctl() 方法。这里重点讨论通过 PROC 文件系统查询信息的方法。

PROC 文件系统是一种特殊的文件系统—伪文件系统, 系统启动后通过下面的命令 (在启动脚本中) 将其挂载到 /proc 目录:

```
# mount -t proc proc /proc
```

内核使用它向外界输出信息。/proc 下面的每个数字名称的目录都对应一个进程, 其他每个文件都绑定于一个内核函数, 这个函数在文件被读取时, 动态地生成文件的 “内容”。我们已经见到过这类文件的一些输出情况, 例如, /proc/modules 列出的是当前载入模块的列表。

Linux 系统对 /proc 的使用很频繁。现代 Linux 系统中的很多工具都通过 /proc 来获取它们的信息。例如 ps、top 和 uptime 这些命令。我们在编写设备驱动程序时也可以通过 /proc 输出信息。因为 PROC 文件系统是动态的, 所以驱动程序模块可以在任何时候添加或删除其中的文件项。

/proc 文件不仅可以用于读出数据, 也可以用于写入数据, 只是在大多数应用场合, /proc 文件项是只读文件。本节将只涉及简单的只读情形。所有使用 PROC 的模块必须包含 linux/proc_fs.h, 通过这个头文件定义正确的函数。

为创建一个只读 /proc 文件, 驱动程序必须实现一个函数, 用于在文件读取时生成数据。当某个进程读这个文件时 (使用 read() 系统调用), 该请求会通过接口函数发送到驱动程序模块。使用哪个接口取决于注册情况。2.4 版本以后的内核通过 create_proc_read_entry() 注册 read_proc() 方法: [12]

```
int (*read_proc)(char *page, char **start, off_t offset,
                 int count, int *eof, void *data);
```

参数表中的 page 指针指向将写入数据的缓冲区, *start 被函数用来说明有意义的数据写在页面的什么位置, offset 和 count 这两个参数与在 read() 实现中的用法相同。eof 参数指向一个整型数, 当没有数据可返回时, 驱动程序必须设置这个参数; data 参数是一个驱动程序特有的数据指针, 可用于内部记录。

无论采用哪个接口, 内核都会分配一页内存 (也就是 PAGE_SIZE 个字节, 在一个标准的 32 位系统中一般是 4KB), 驱动程序向这片内存写入将返回给用户空间的数据。

对于 PROC 文件系统的用户扩展, 其最初实现中的主要问题在于, 数据传输只使用单个内存页面。这样就把用户文件的总体尺寸限制在了 4KB 以内 (或者是适合于主机平台的其他值)。start 参数在这里就是用来实现大数据文件的, 不过该参数可以被忽略。

如果 read_proc() 函数不对 *start 指针进行设置 (它最初为 NULL), 内核就会假定 offset 参数被忽略, 并且数据页包含了返回给用户空间的整个文件。反之, 如果需要通过多个片段创建一个更大的文件, 则可以把 *start 赋值为页面指针, 从而调用者也就知道了新数据放在缓冲区的开始位置。当然, 应该跳过前 offset 个字节的数据, 因为这些数据已经在前面的调用中返回。

一旦定义好了一个 read_proc() 函数, 就需要把它与一个 /proc 文件项关联起来。下面的调用, 以 /proc/proctest 的形式来提供 /proc 功能。

```
    create_proc_read_entry("proctest",
        0     /* default mode */,
        NULL /* parent dir */,
```

[12]更老的还有一个名为 get_info() 的接口。

```
        procfile_read,
        NULL /* client data */);
```

这个函数的参数表包括: /proc 文件项的名称、应用于该文件项的文件许可权限 (0 是个特殊值, 会被转换为一个默认的、完全可读模式的掩码)、文件父目录的 `proc_dir_entry` 指针 (我们使用 NULL 值使该文件项直接定位在 /proc 下)、指向 `read_proc()` 的函数指针, 以及将传递给 `read_proc()` 函数的数据指针。

目录项指针 (`proc_dir_entry`) 可用来在 /proc 下创建完整的目录层次结构。不过请注意, 将文件项置于 /proc 的子目录中有更为简单的方法, 即把目录名称作为文件项名称的一部分—只要目录本身已经存在。例如, 有个新的约定, 要求设备驱动程序对应的 /proc 文件项应转移到子目录 driver/ 中。驱动程序可以简单地指定它的文件项名称为 driver/proctest, 从而把它的 /proc 文件放到这个子目录中。

内核 3.10 版本以后, 直接使用和设备注册相同的方式注册一个 `file_operations` 结构:

```
struct proc_dir_entry *proc_create(
        const char *name,
        umode_t mode,
        struct proc_dir_entry *parent,
        const struct file_operations *proc_fops
        );
```

模块初始化创建一个 /proc 项:

```
/* procfs.c */
...
#define PROCFS_NAME    "proctest"
static struct proc_dir_entry *myprocfile;
int init_module()
{
    /* create the /proc file */

    myprocfile = proc_create(PROCFS_NAME, 0666, NULL, &fops);

    if (myprocfile == NULL) {   /* create fail */
        remove_proc_entry(PROCFS_NAME, myprocfile);
        printk(KERN_ALERT "Error: Could not initialize /proc/%s\n",
                PROCFS_NAME);
        return -ENOMEM;
    }

    printk(KERN_INFO "/proc/%s created\n", PROCFS_NAME);
    return 0;   /* everything is ok */
}
```

下面是模块中 read() 和 write() 函数的简单实现:

```
/* procfs.c */
...
static char procfs_buffer[1024];

ssize_t procfile_read(struct file *file,
                      char *buffer,
                      size_t count,
                      loff_t *f_pos)
{
    int ret;

    printk(KERN_INFO "procfile_read (/proc/%s) called\n", PROCFS_NAME);

    if (f_pos > 0) {
        /* we have finished to read, return 0 */
        ret  = 0;
    } else {
        /* fill the buffer, return the buffer size */
        memcpy(buffer, procfs_buffer, procfs_buffer_size);
        ret = procfs_buffer_size;
    }

    return ret;
}

ssize_t procfile_write(struct file *file,
                       const char *buffer,
                       size_t count,
                       loff_t *f_pos)
{
    printk(KERN_INFO "procfile_write (/proc/%s) called\n", PROCFS_NAME);
    /* get buffer size */
    procfs_buffer_size = count;

    if (procfs_buffer_size > PROCFS_MAX_SIZE)
        procfs_buffer_size = PROCFS_MAX_SIZE;

    /* write data to the buffer */
```

```
    if(copy_from_user(procfs_buffer, buffer, procfs_buffer_size))
        return -EFAULT;

    return procfs_buffer_size;
}
```

在模块卸载时, /proc 中的文件项也应被删除。remove_proc_entry()就是用来撤消 create_proc_read_entry() 或 proc_create()|所做工作 函数。

```
void remove_proc_entry(const char *, struct proc_dir_entry *);
```

4.6.3 监视调试

有时, 通过监视用户空间中应用程序的运行情况, 可以捕捉到一些小问题。监视程序同样也有助于确认驱动程序工作是否正常。例如, 看到 timer 的 read() 实现如何响应不同数据量的 read() 请求后, 我们就可以判断它是否工作正常。

有许多方法可监视用户空间程序的工作情况。可以用调试器一步步跟踪它的函数, 插入打印语句, 或者在 strace 状态下运行程序。在检查内核代码时, 最后一项技术最值得关注。我们将在此对它进行讨论。

strace 命令是一个功能非常强大的工具, 它可以显示程序所调用的所有系统调用。它不仅可以显示调用, 而且还能显示调用参数, 以及用符号方式表示的返回值。当系统调用失败时, 错误的符号值 (如 ENOMEM) 和对应的字符串 (如 Out of memory) 都能被显示出来。strace 有许多命令行选项, 下面是几个常用的:

-t 显示调用发生的时间;
-T 现实调用花费的时间;
-e 限定被跟踪的调用类型;
-o 输出到一个文件中。默认的情况下 strace 将跟踪信息打印到标准错误输出设备上。

strace 从内核中接收信息。这意味着一个程序无论是否按调试方式编译 (用 gcc 的 "-g" 选项) 或是被去掉了符号信息都可以被跟踪。与调试器可以连接到一个运行进程并控制它一样, strace 也可以跟踪一个正在运行的进程。

跟踪信息通常用于生成错误报告, 然后发给应用开发人员, 它对内核开发人员调试设备驱动也非常有用, 它可以帮助检查系统调用中输入和输出数据的一致性。例如, 在一个使用了 "open(fd, buf, 2000)" 语句的应用程序中, 调用驱动程序 timer 读取设备文件 /dev/timer, 可以看到下面这些相关信息:

```
$ strace  ./readfunc
  ...
  open("/dev/timer", O_RDWR)                    = 3
  read(3, "\240\23\235\23\231\23"..., 2000) = 1024
  exit_group(0)                                 = ?
```

设备文件 /dev/timer 被打开, 返回文件描述符 3, 试图从 3 号文件读取 2000 字节, 实际返回值 1024, 这正是驱动程序限制的。

有经验的开发人员可以在 strace 的输出中发现很多有用信息。如果觉得这些符号过于拖累的话, 则可以仅限于监视文件方法 (open()、read() 等) 是如何工作的。strace 能够确切查明系统调用的哪个参数引发了错误, 这一点对调试是大有帮助的。

4.6.4 故障调试

大部分错误来源于不正确的指针使用上。在用户空间的应用程序也是如此。用户空间错误的指针引用通常会导致段错误 (Segment Fault), 在内核空间同样会导致严重问题, 内核出问题时更难调试。尽可能多地收集错误信息, 对于解决问题是有帮助的, 内核的 oops 消息起到的就是这样的作用。

由于处理器使用的地址都是虚拟地址, 任何一个逻辑地址都需要通过页表转换映射到物理地址。当引用一个未经页表记录的逻辑地址时, 页面映射机制就不能将地址映射到物理地址。此时处理器就会向操作系统发出一个页面失效的信号。如果地址非法, 内核就无法换页到并不存在的地址上。如果此时处理器处于超级用户模式, 系统就会触发一个 oops 消息。

klogd 守护进程能在 oops 消息到达记录文件之前对它们解码, 并记录下故障发生时的地址 (通常是非法的地址) 和寄存器的状态, 这些是开发人员定位错误的重要信息。有时候, 这些信息还不够, 而且在内核出错的时候 klogd 崩溃也是常事。一个更为强大的分析工具 ksymoops 有助于开发人员进一步解决问题。Ksymoops 由 Linux 开发团队维护, 可以在 https://www.kernel.org/pub/linux/utils/kernel/ksymoops/ 获得它的源码和各发行版的可执行程序。

4.6.5 使用 GDB 调试工具

使用 GDB 调试内核时, 必须把内核看作是一个应用程序。除了指定未压缩的内核镜像文件名外, 还应该在命令行中提供 core 文件的名称。对于正运行的内核, 所谓 core 文件就是这个内核在内存中的核心镜像 /proc/kcore。典型的 gdb 调试命令采用如下方式启动:

```
$ gdb /usr/src/linux/vmlinux /proc/kcore
```

第一个参数是未经压缩的内核可执行文件的名字, 而不是 zImage 或 bzImage 以及其他任何压缩过的内核。gdb 命令行的第二个参数是 core 文件的名字。与其他 /proc 中的文件类似, /proc/kcore 也是在被读取时产生的。当在 /proc 文件系统中执行 `read()` 系统调用时, 它会映射到一个用于数据生成而不是数据读取的函数上。kcore 用来按照 core 文件的格式表示内核 “可执行文件”。由于它要表示对应于所有物理内存的整个内核地址空间, 所以是一个非常巨大的文件。在 gdb 的使用中, 可以通过标准 gdb 命令查看内核变量。例如, `p jiffies` 可以打印从系统启动到当前时刻的时钟滴答数。

gdb 打印数据时, 内核仍在运行, 不同数据项的值会在不同时刻有所变化。gdb 为了优化调试过程, 会将已经读到的数据缓存起来。如果再次打印这些数据, 会得到和之前所看到的相同的值。对普通应用程序来说, 这个结果没有问题, 但对于正在运行的内核来说, 再次打印出与上次相同的 jiffies 值就是错的。解决的方法是在需要刷新 gdb 缓冲区的时候, 执行命令 core-file /proc/kcore, 调试器将使用新的 core 文件并丢弃所有的旧信息。不过, 读新数据时并不总是需要执行 core-file 命令, gdb 以几 KB 大小的小数据块形式读取 core 文件, 缓存的仅是已经引用的若干小块。

对内核进行调试时, gdb 通常能提供的许多功能都不可用。例如, gdb 不能修改内核数据, 因为在处理其内存镜像之前, gdb 期望把待调试程序运行在自己的控制之下。同样, 也不能设置断点或观察点, 或者单步跟踪内核函数。

编译内核时, 也可以启用调试选项 “-g” 为 gdb 提供有用的源程序符号, 产生的 vmlinux 更适合于 gdb。当用 “-g” 选项编译内核并且和 /proc/kcore 一起使用 vmlinux 运行调试器时, gdb 可以返回很多内核内部信息。例如, 可以使用下面的命令来转储结构数据, 如 p *module_list、p *module_list->next 和 p *chrdevs[4]->fops 等。为了在使用 p 命令时取得最好效果, 有必要保留一份内核映射表和随手可及的源码。

不过要注意, 用 “-g” 选项编译出的内核镜像比不使用 “-g” 选项编译出来的大得多。在第 3 章介绍 `strip` 命令时, 我们已从相反的过程看到了, 调试信息在程序里占有多大的空间。

利用 `gdb` 的另一个有用的功能是反汇编, 用 disassemble 反汇编代码, 用 x/i 检查十六进制指令或数据。但 `gdb` 不能反汇编一个模块的函数, 因为调试器作用的是 vmlinux, 它并不知道模块的情况。如果试图通过地址反汇编模块代码, `gdb` 很有可能会返回 “Cannot access memory at xxxx” 这样的信息。基于同样的原因, 也不能查看属于模块的数据项。如果已知变量的地址, 可以从 /dev/mem 中读出它们的值。但要弄明白从系统内存中分解出的原始数据的含义, 难度是相当大的。

如果需要反汇编模块函数, 最好对模块的目标文件用 `objdump` 工具进行处理。遗憾的是, 该工具只能对磁盘上的文件复本进行处理, 而不能对运行中的模块进行处理。因此, 由 `objdump` 给出的地址都是未经重定位的地址, 与模块的运行环境无关。对未经链接的目标文件进行反汇编的另一个不利因素在于, 其中的函数调用仍是未作解析的, 所以就无法轻松地区分是对 `printk()` 的调用呢, 还是对 `kmalloc()` 的调用。

正如上面看到的, 当目的在于查看内核的运行情况时, `gdb` 是一个有用的工具。但对于设备驱动程序的调试, 它还缺少一些至关重要的功能。

4.6.6 使用内核调试工具

另外两个有用的工具是 kdb 和 kgdb, 他们可以看作是内核的调试器前端。kdb 可以用于检查内核运行时的存储器、寄存器、进程列表, 甚至也可以在指定位置设置断点, 如果编译内核时开启了 `CONFIG_KALLSYMS`, 还可以以名称访问内建的一些符号。但它不是一个源码级的调试器。源码级调试器是 kgdb 设计的初衷。

调试内核时, kgdb 要与 `gdb` 配合使用, 它让 `gdb` 可以进入内核, 监视存储器和变量, 查看调用栈的信息, 在内核中设置断点, 执行一些有限的单步调试, 就像调试应用程序的效果一样。

使用 kgdb 时同样也需要配置内核的相关选项。使能 kgdb 的相关选项也在 kernel hacking 子菜单下:

```
<*>   KGDB: use kgdb over the serial console
[*]   KGDB: internal test suite
[ ]     KGDB: Run tests on boot
[*]   KGDB_KDB: include kdb frontend for kgdb
(0x1)   KDB: Select kdb command functions to be enabled by default
[ ]     KGDB_KDB: keyboard as input device
(0)     KDB: continue after catastrophic errors
```

kgdb 调试时, 需要两台计算机: 一台正常工作的作为主机, 运行一个针对目标系统 vmlinux 的 `gdb` 实例, 经过特殊配置的待调试内核运行在另一台目标系统中, 二者通过特定的 I/O 连接, 连接方式取决于目标系统的支持模块。使用串行终端的是 kgdboc (意为 kgdb over console)。如果通过以太网接口调试, 则是 kgdboe (kgdb over ethernet)。这两个参数中的一个需要在目标系统启动时通过 Bootloader 传递给内核。例如, 通过下面这种方式将监控终端和调试终端都设为 ttySAC2:

```
console=ttySAC2,115200 kgdboc=ttySAC2,115200
```

主机则在 gdb 提示符下通过

```
(gdb) target remote /dev/ttyS0
```

与目标系统连接。/dev/ttyS0 是主机连接串口的设备文件。

4.7 硬件管理与中断处理

4.7.1 I/O 寄存器和常规内存

I/O 寄存器和 RAM 的最主要区别就是 I/O 操作具有边际效应 (side effect) [13], 而内存操作则没有: 内存写操作的唯一结果就是在指定位置存储一个数值; 内存读操作则仅仅返回指定位置最后一次写入的数值。由于内存访问速度对 CPU 的性能至关重要, 而且也没有边际效应, 所以可用多种方法进行优化, 如使用高速缓存保存数值、重新排序读/写指令等。

编译器能够将数值缓存在 CPU 寄存器中而不写入内存, 即使存储数据, 读写操作也都能在高速缓存中进行而不用访问物理内存。无论在编译器一级或是硬件一级, 指令的重新排序都有可能发生: 一个指令序列如果以不同于程序文本中的次序运行, 常常能执行得更快。例如, 在防止 RISC 处理器流水线的互锁时就是如此。在 CISC 处理器上, 耗时的操作则可以和运行较快的操作并发执行。

在对常规内存进行这些优化的时候, 优化过程是透明的, 而且效果良好 (至少在单处理器系统上是这样)。但对 I/O 操作来说这些优化很可能造成致命的错误, 因为它们会干扰边际效应, 而这却是驱动程序访问 I/O 寄存器的主要目的。处理器无法预料到某些其他进程 (在另一个处理器上运行, 或在某个 I/O 控制器中) 是否会依赖于内存访问的顺序。因此驱动程序必须确保不会使用高速缓存, 并且在访问寄存器时不会发生读或写指令的重新排序: 编译器或 CPU 可能会自作聪明地重新排序所要求的操作, 结果是发生奇怪的错误, 并且很难调试。

由编译器优化和硬件重新排序引起的问题的解决办法是, 在从硬件角度看必须以特定顺序执行的操作之间设置内存屏障。Linux 提供了四个函数 (宏) 来解决所有可能的排序问题。

```
#include <linux/kernel.h>
void barrier(void);
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

函数 `barrier()` 通知编译器插入一个内存屏障, 但对硬件无效。编译后的代码会把当前 CPU 寄存器中的所有修改过的数值存到内存, 需要这些数据的时候再重新读出来。后三个函数在已编译的指令流中插入硬件内存屏障; 具体的插入方法是平台相关的。`rmb()` (读内存屏障) 保证了屏障之前的读操作一定会在后来的读操作执行之前完成。`wmb()` 保证写操作不会乱序, `mb()` 指令保证了两者都不会。这些函数都是 `barrier()` 的超集。

设备驱动程序中使用内存屏障的典型格式如下:

```
writel(dev->registers.addr, io_destination_address);
```

[13]此处所谓的 “边际效应”, 有时又被译为 “副作用”。它是指对 I/O 寄存器的读出值与上一次的写入值之间没有关系。

```
    writel(dev->registers.size, io_size);
    writel(dev->registers.operation, DEV_READ);
    wmb();
    writel(dev->registers.control, DEV_GO);
```

在这个例子中, 最重要的是要确保控制某特定操作的所有设备寄存器一定要在操作开始之前正确设置。其中的内存屏障会强制写操作以必需的次序完成。因为内存屏障会影响系统性能, 所以应该只用于真正需要的地方。不同类型的内存屏障影响性能的方面也不同, 所以最好尽可能使用针对需要的特定类型。例如在当前的 X86 体系结构上, 由于处理器之外的写不会重新排序, `wmb()`就没什么用。可是读会重新排序, 所以 `mb()` 就会比 `wmb()`慢一些。

在有些体系结构上允许把赋值语句和内存屏障进行合并以提高效率。2.4 版本内核提供了几个执行这种合并的宏, 它们默认情况下定义如下:

```
#define set_mb(var, value)  do {var = value; mb();}  while 0
#define set_wmb(var, value) do {var = value; wmb();} while 0
#define set_rmb(var, value) do {var = value; rmb();} while 0
```

在适当的地方, asm/system.h 中定义的这些宏可以利用体系结构特有的指令更快地完成任务。

使用 I/O 端口

I/O 端口是驱动程序与许多设备的之间信息交换的通道。4.3.4 介绍了分配和释放端口的函数。

驱动程序请求了需要使用的 I/O 端口范围后, 肯定是需要读或者写这些端口。为此, 大多数硬件都把 8 位、16 位和 32 位的端口区分开来。它们不能像访问系统内存那样混淆。因此, C 语言程序必须调用不同的函数来访问大小不同的端口。仅支持存储器映射 I/O 的处理器架构通过把 I/O 端口地址重新映射到内存地址来模拟端口 I/O, 并且为了易于移植, 内核对驱动程序隐藏了这些细节。Linux 内核与体系结构相关的头文件 asm/io.h 中定义了如下一些访问 I/O 端口的内联函数。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned short inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned long inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

函数名后缀 “b”、“w”、“l” 分别表示按 8 位、16 位或者 32 位读写端口。它们与处理器架构相关, 在一些平台上可能不支持。

上面这些函数主要是提供给设备驱动程序使用的, 但它们也可以在用户空间使用, 至少在 PC 类计算机上可以使用。GNU 的 C 库在 sys/io.h 中定义了这些函数。如果要在用户空间代码中使用 `inb()` 这类函数, 必须满足下面这些条件:

1. 编译该程序时必须带 -O 选项来强制内联函数的展开。

2. 必须用 ioperm() 或 iopl() 来获取对端口进行 I/O 操作的许可。ioperm() 用来获取对单个端口的操作许可, 而 iopl() 用来获取对整个 I/O 空间的操作许可。这两个函数都是 Intel 平台特有的。
3. 必须以 root 身份运行该程序才能调用 ioperm() 或 iopl()。或者, 该程序的某个祖先已经以 root 身份获取了对端口操作的权限。

如果宿主平台没有 ioperm() 和 iopl() 系统调用, 用户空间程序仍然可以使用 /dev/port 设备文件访问 I/O 端口。不过要注意, 该设备文件的含义和平台的相关性是很强的, 并且除 PC 上以外, 它几乎没有用处。

程序 4.6 是在 PC 机上用户空间直接读写端口的例子, 它利用 PC 的可编程计数器和扬声器输出一个音阶的声音。我们在 4.4 一节实现了它的驱动。这里跳过驱动层直接操作端口。在本例中, 2 号计数器是一个分频器 (工作方式三), 对 1.19 MHz 时钟进行分频, 输出一个方波。原理图见图 4.4。

清单 4.6: 直接读写端口输出声音 notes.c

```
/* notes.c */
#include <sys/io.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int notes[] = {
        131, 147, 165, 175, 196, 220, 247, 262};
    int i, cnt;
    unsigned char hi, lo;

    /* setuid(geteuid());  */
    ioperm(0x42, 0x2, 0xffff);
    ioperm(0x61, 1, 0xffff);
    outb(0xb6, 0x43);
    outb(0x3, 0x61);       /* Enable Speaker */
    for(i = 0; i < 8; i++) {
        cnt = 1193182 / notes[i];
        hi = cnt / 256;
        lo = cnt % 256;
        outb(lo, 0x42);
        outb(hi, 0x42);
        usleep(200000);
    }
    outb(0, 0x61);         /* Disable Speaker */
    return 0;
}
```

如果想冒险, 可以将它设置上 SUID 位 (去掉第 12 行的注释, 用 `chown` 将编译后的可执行程序拥有者改为超级用户 root, 用 chmod u+s 设置运行用户 ID)。这样, 不用显式地获取特权就可以使用硬件了。也可以直接以超级用户身份运行。

以上的 I/O 操作都是一次传输一个数据。作为补充, 有些处理器上实现了一次传输一个数据序列的特殊指令, 序列中的数据单位可以是字节、双字节或字 (四字节)。这些指令称为串操作指令 (例如 X86 的串操作指令), 它们执行这些任务时比 C 语言写的循环语句快得多。下面列出的宏实现了串 I/O, 它们或者使用一条机器指令实现, 或者在没有串 I/O 指令的平台上使用紧凑循环实现。

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

函数名后缀 “b”、“w”、“l” 分别表示对数据总线 8 位、16 位和 32 位的端口操作。

使用 I/O 内存

I/O 内存仅仅是类似 RAM 的一个区域, 在那里处理器可以通过总线访问设备。这种内存有很多用途, 比如存放视频数据或网络包。这些用设备寄存器也能实现, 其行为类似于 I/O 端口 (比如, 读写时有边际效应)。

访问 I/O 内存的方法和计算机体系结构、总线, 以及设备是否正在使用有关, 不过原理都是相同的。这里我们只讨论 ISA 和 PCI 内存。

根据计算机平台和所使用总线的不同, I/O 内存可能是通过页表访问的, 但也可能不是通过页表。如果访问是经由页表进行的, 内核必须首先安排物理地址使其对设备驱动程序可见 (这通常意味着在进行任何 I/O 之前必须先调用 `ioremap()`)。如果访问无需页表, 那么 I/O 内存区域就很像 I/O 端口, 可以使用适当形式的函数读写它们。下面是一组与此相关的内核函数:

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

不管访问 I/O 内存时是否需要调用 `ioremap()`, 都不建议直接使用指向 I/O 内存的指针, 尽管 I/O 内存在硬件一级是像普通 RAM 一样寻址的。使用包装的函数访问 I/O 内存, 一方面在所有平台上都是安全的, 另一方面, 在可以直接对指针指向的内存区域执行操作的时候, 这类函数是经过优化的。因此, 即使在 X86 平台允许使用指针的情况下, 用指针代替宏的做法也会影响驱动程序的可移植性和可读性。

前已述及 (见4.3.4节), 设备内存区域在使用前应先分配。这和 I/O 端口注册过程类似。

在不同的处理器平台上, 内核对 I/O 区域的内存映射有不同的策略。一些计算机平台上保留了部分内存地址空间留给 I/O 区域, 并且自动禁止对该内存范围内的虚拟地址的映射。用在个人数字助理 PDA (**P**ersonal **D**igital **A**ssistant) 中的 MIPS 处理器就是这种配置的一个有趣的实例。两个各为 512MB 的地址段直接映射到物理地址, 对这些地址范围

内的任何内存访问都绕过存储器管理单元 (**M**emory **M**anagement **U**nit, MMU), 也绕过缓存。这些 512MB 地址段中的一部分是为外设保留的, 驱动程序可以用这些无缓存的地址范围直接访问设备的 I/O 内存。

还有一些使用特殊的地址空间来解析物理地址, 另一些则使用虚拟地址, 这些虚拟地址被设置成访问时绕过处理器缓存。

当需要访问直接映射的 I/O 内存区时, 仍然不应该直接使用 I/O 指针指向的地址 — 即使在某些体系结构这么做也能正常工作。为了编写出的代码在各种系统和内核版本都能工作, 应该避免使用直接访问的方式, 而代之以下列的宏:

```
#include <asm/io.h>
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

上述 `read*()`、`write*()` 宏用来从 I/O 内存读/写 8 位、16 位和 32 位的数据。使用宏的好处是不用考虑参数的类型: 参数 `address`是在使用前才强制转换的, 因为这个值不清楚是整数还是指针, 所以两者都要接收。读函数和写函数都不会检查参数 `address` 是否合法, 因为这在解析指针指向区域的同时就能知道 (我们已经知道有时它们确实扩展成指针的反引用操作)。

```
memset_io(address, value, count);
```

当需要在 I/O 内存上调用 memset() 时, 这个函数可以满足需要, 同时它保持了原来的 `memset()` 的语义。

```
memcpy_fromio(dest, source, num);
memcpy_toio(dest, source, num);
```

这两个函数用来和 I/O 内存交换成块的数据, 功能类似于 C 库函数 `memcpy()`。

在较新的内核版本中, 这些函数在所有体系结构中都是可用的。当然具体实现会有不同, 在一些平台上是扩展成指针操作的宏, 在另一些平台上是真正的函数。不过作为驱动程序开发人员, 不需要关心它们具体是怎样工作的, 只要会用就行了。

通过软件映射的 I/O 内存

尽管 MIPS 类的处理器使用直接映射的 I/O 内存, 但这种方式在现在的平台中是相当少见的, 特别是当使用外设总线处理映射到内存的设备时更是如此。

使用 I/O 内存时最普遍的硬件和软件处理方式是这样的: 设备对应于某些约定的物理地址, 但是 CPU 并没有预先定义访问它们的虚拟地址。这些约定的物理地址可以是硬连接到设备上的, 也可以是在启动时由系统固件 (如 BIOS) 指定的。前一种的例子有 ISA 设备, 它的地址或者是固化在设备的逻辑电路中, 因而已经在局部设备内存中静态赋值, 或者是通过物理跳线设置; 后一种的例子有 PCI 设备, 它的地址是由系统软件赋值并写入设备内存的, 只在设备加电初始化后才存在。

不管哪种方式, 为了让软件可以访问 I/O 内存, 必须有一种把虚拟地址赋于设备的方法。这个任务是由 `ioremap()` 函数完成的。这个函数因为与内存的使用相关, 在前面已经

提到过, 它就是为了把虚拟地址指定到 I/O 内存区域而专门设计的。此外, 由内核开发人员实现的 `ioremap()` 在用于直接映射的 I/O 地址时不起任何作用。

`iounmap()` 用于解除 `ioremap()` 的映射。一旦有了 `ioremap()` 和 `iounmap()`, 设备驱动程序就能访问任何 I/O 内存地址, 而不管它是否直接映射到虚拟地址空间。不过要记住, 这些地址不能直接引用, 而应该使用像 `readb()` 这样的函数。

函数 `ioremap_nocache()` 与硬件相关。如果有某些控制寄存器在这个区域, 并且不希望发生写操作合并或读缓存的话, 可以使用它。实际上, 大多数计算机平台上这个函数的实现和 `ioremap()` 是完全一样的, 因为在所有 I/O 内存都已经可以通过非缓存地址访问的情况下, 没有必要再实现一个单独的、非缓存的 `ioremap()`。

4.7.2 中断

中断的整体控制

由于设计上和硬件上的改变, Linux 处理中断的方法近几年来有所变化。早期 PC 的中断处理很简单, 中断的处理仅仅涉及到一个处理器和 16 条中断信号线。而现代硬件则可以有更多的中断, 并且还可能装配价格高昂的高级可编程中断控制器 (APIC), 控制器可以以一种智能 (和可编程) 的方式在多个处理器之间分发中断。

中断处理函数面临的一个问题是如何处理较长时间的中断服务。大量的中断需要及时响应, 同时每个中断也不应该占用太长的时间。Linux 系统解决这个问题的办法是将中断任务分成两个 “半部”: 顶半 (top half) 和底半 (bottom half)。顶半是响应中断的子程序, 也就是用 `request_irq()` 注册的中断句柄。典型的应用中, 顶半中将耗时的任务纳入调度器, 以便稍后在方便的时候处理, 这部分任务就是底半。顶半完成一些必要的工作后迅速结束。这两个半部的最大差别是, 底半是在中断允许模式下运行的, 不会耽误其他中断请求继续得到响应。

中断处理程序

如果读者确实想 “看到” 产生的中断, 仅仅通过向硬件设备写入是不够的, 必须要在系统中安装一个软件处理程序。如果 Linux 内核没有被通知硬件中断的发生, 那么内核只会简单应答并忽略该中断。

中断信号线是非常珍贵且有限的资源, 尤其是在系统上只有 15 根或 16 根中断信号线时更为如此。内核维护了一个中断信号线的注册表, 它类似于 I/O 端口的注册表。模块在使用中断前要先申请一个中断通道 (或者中断请求 IRQ), 然后在使用后释放该通道。我们将会在后面看到, 在很多场合下, 模块也希望可以和其他的驱动程序共享中断信号线。下列函数实现了该接口:

```
#include <linux/sched.h>
int request_irq(unsigned int irq,
    irqreturn_t (*handler)(int, void *),
    unsigned long flags,
    const char *dev_name,
    void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

通常, 从 request_irq() 函数返回到调用函数的值为 0 时表示请求成功, 负值表示错误码。函数返回 −EBUSY 表示已经有另一个驱动程序占用了正在申请的中断信号线。该函数的参数如下:

```
unsigned int irq;  /* 这是要申请的中断号 */
```

```
irqreturn_t (*handler)(int irq, void *dev_id);
```

这是要安装的中断处理函数指针。

```
unsigned long flags;
```

这是一个与中断管理有关的位掩码选项。可以在标志中设置的位如下:

- SA_INTERRUPT

  当该位被设置时, 表明这是一个 “快速” 的中断处理程序, 快速处理程序是运行在中断禁止状态下的。
- SA_SHIRQ

  该位表示中断可以在设备之间共享。
- SA_SAMPLE_RANDOM

  该位指出产生的中断能对 /dev/random 设备和 /dev/urandom 设备使用的熵池有贡献。从这些设备读取数据, 将会返回真正的随机数, 从而有助于应用软件选择用于加密的安全密钥。这些随机数是从一个熵池中得到的, 各种随机事件都会对该熵池作出贡献, 如果读者的设备以真正随机的周期产生中断, 就应该设置该标志位。另一方面, 如果中断是可预期的 (例如, 帧捕获卡的垂直消隐), 就不值得设置这个标志位—它对系统的熵没有任何贡献。能受到攻击者影响的设备不应该设置该位, 例如, 网络驱动程序会被外部的事件影响到预定的数据包的时间周期, 因而也不会对熵池有贡献。

```
const char *dev_name;
```

传递给 request_irq() 的字符串, 用来在 /proc/interrupts 中显示中断的所属设备。

```
void *dev_id;
```

这个指针用于共享的中断信号线。在释放中断信号线时, 它是标识设备的唯一标识符。驱动程序也可以使用它指向驱动程序自己的私有数据区 (用来识别哪个设备产生中断)。在没有强制使用共享方式时, dev_id 可以设置为 NULL, 用它来指向设备的数据结构是一个比较好的思路。

中断处理程序可在驱动程序初始化时或者设备第一次打开时安装。虽然在模块初始化函数中安装中断处理程序似乎是个好主意, 实际上并非如此。因为中断信号线的数量是非常有限的, 我们不会想着肆意浪费。计算机拥有的设备通常要比中断信号线多得多, 如果一个模块在初始化时请求了 IRQ, 即使驱动程序只是占用它而从未使用, 也将会阻止任意一个其他的驱动程序使用该中断。而在设备打开的时候申请中断, 则可以共享这些有限的资源。

出现这样的情况并非鲜见。例如, 在运行一个与调制解调器共用同一中断的帧捕获卡驱动程序时, 只要不同时使用两个设备就可以。用户在系统启动时装载特殊的设备模块是一种普遍作法, 即使该设备很少使用。数据捕获卡可能会和第二个串口使用相同的中断。我们可以在捕获数据时, 避免使用调制解调器连接到互联网服务供应商。但是如果为了使用调制解调器而不得不卸载一个模块, 毕竟是一件麻烦的事。

调用 request_irq() 的正确位置应该是在设备第一次打开、硬件被告知产生中断之前, 调用 free_irq() 正确的位置是最后一次设备关闭、硬件被告知不要再中断处理器之后。这种技术的缺点是必须为每个设备保存一个打开计数。如果使用一个模块控制两个或者更多的设备, 那么仅仅使用模块计数是不够的。

下面这段代码片段演示了对 PC 并口的驱动。要申请的中断是 parport_irq。对这个变量的实际赋值与硬件结构有关。parport_base 是并口使用的 I/O 地址空间的基地址 (在早期的 PC 中可能是 0x378 或 0x278)。向并口的 2 号寄存器 (控制寄存器) 的 bit4 写入 1, 可以打开中断使能。

```
/* parport.c */
...
    if (parport_irq >= 0) {
        result = request_irq(parport_irq, parport_interrupt,
                             SA_INTERRUPT, "parport", NULL);
        if (result) {
            printk(KERN_INFO "parport: can't get assigned irq %i\n",
                   parport_irq);
            parport_irq = -1;
        }
        else { /* bit4=并口中断允许位 */
            outb(0x10, parport_base+2);
        }
    }
```

从代码能够看出, 已经安装的中断处理程序是一个快速的处理程序 (SA_INTERRUPT), 不支持中断共享 (没有设置 SA_SHIRQ), 并且对系统的熵没有贡献 (也没有设置 SA_SAMPLE_RANDOM)。最后, 代码执行 outb() 开放中断允许。

4.8 内核的定时

本节讨论如何理解内核定时机制: 如何获得当前时间, 如何将操作延迟指定的一段时间, 如何调度异步函数经过指定的时间之后再执行。

4.8.1 时间间隔

定时中断

时钟中断由系统计时硬件以周期性的间隔产生, 这个间隔由内核根据 HZ 的值设定。HZ 是一个与体系结构有关的常数, 在文件 linux/param.h 中定义。可以将它理解为以 Hz 为单位的时间片任务调度频率。在主频 1GHz 左右的硬件平台上, 这个值在内核中缺省设置为 250。HZ 值越大, 任务的实时响应越好, 但同时也增加了任务切换的开销。驱动程序应尽可能以 HZ 为基础进行计数, 而不要使用特定的频率计数值。

内核维护一个全局变量 jiffies, 在系统启动时初始化为 0。每次时钟中断发生时, 变量 jiffies 的值就会加 1。因此, jiffies 的值就是自操作系统启动以来响应的时

钟中断的次数 (以 HZ 为计时周期数值)。jiffies 在头文件 linux/sched.h 中被定义为 **unsigned long volatile** 型变量, 这个变量在经过长时间的连续运行后有可能溢出。

如果想改变系统时钟中断发生的频率, 可以修改 HZ 值。有人使用 Linux 处理硬实时任务, 为此他们增大了 HZ 值以获得更快的响应时间, 而情愿忍受额外的时钟中断产生的系统开销。总而言之, 时钟中断的最好方法是保留 HZ 的缺省值, 我们有理由相信内核的开发者们, 他们选择的 HZ 值是合理的。

TSC 寄存器

如果需要度量非常短的时间, 或是需要极高的时间精度, 可以使用与特定平台相关的资源, 这是将时间精度的重要性凌驾于代码的可移植性之上的做法。

绝大多数现代处理器都包含一个随时钟周期不断递增的计数寄存器。基于不同的平台, 在用户空间, 这个寄存器可能是可读/写的, 也可能是不可读/写的; 可能是 64 位的也可能是 32 位的。如果是 32 位的, 还得注意处理溢出的问题。无论该寄存器是否可写, 我们都强烈建议不要重置它, 即使硬件允许这么做。因为总可以通过先后两次读取该寄存器并比较其数值的差异来达到我们的目的, 无须要求独占该寄存器并修改它的当前值。

最有名的计数器寄存器就是 TSC (**T**ime **S**tamp **C**ounter, 时间戳计数器)。Intel X86 系列从 Pentium 处理器开始提供该寄存器, 并包含在以后的所有 CPU 中。它是一个 64 位的寄存器, 记录 CPU 时钟周期数。内核空间和用户空间都可以读取它。在 32 位 X86 系统中, 指令 `rdtsc` 将该计数器的值读入两个 32 位的寄存器 EDX 和 EAX 中, 在 X86-64 中则是读入 RDX 和 RAX 的低 32 位中。其他处理器架构也都有实现类似功能的计数器, 只是名称不同。

包含了头文件 asm/msr.h (意指 machine-specific registers, 机器特有的寄存器) 之后, 就可以使用如下的宏获得计数器:

```
rdtsc(low, high);
rdtscl(low);
```

`high`、`low` 是对应 TSC 寄存器的高、低 32 位的变量。由于是用宏实现的, 作为返回值的参数可以不使用指针。前一个宏把 64 位的数值读到两个 32 位变量中, 后一个只把寄存器的低 32 位读入一个 32 位变量。在大多数情况下, 32 位计数已经够用了。举例来说, 一个 1GHz 的系统使一个 32 位计数器溢出需 4.2 秒 ($2^{32} \times 10^{-9}$)。如果要处理的时间比这短的话, 就没有必要读出整个寄存器 (但需要考虑一个 32 位寄存器回绕问题)。

在 X86 平台, 下面的宏使用内联汇编实现读取 64 位 TSC 的功能:

```
#define rdtsc(low,high) \
     __asm__ __volatile__("rdtsc" : "=a" (low), "=d" (high))
```

下面这段代码可以测量该指令自身的运行时间:

```
unsigned long ini, end;

rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

4.8.2 获取当前时间

内核一般通过 jiffies 值来获取当前时间。该数值表示的是自最近一次系统启动到当前的时间间隔, 它和设备驱动程序不怎么相关, 因为它的生命期只限于系统的运行期 (uptime)。但驱动程序可以利用 jiffies 的当前值来计算不同事件间的时间间隔 (比如在输入设备驱动程序中就用它来分辨鼠标的单击和双击)。

如果驱动程序真的需要获取当前的日历时间, 可以使用 `do_gettimeofday()` 函数。该函数并不直接返回今天是几月几号星期几这样的日历信息, 它是用秒和微秒值来填充一个指向 `struct timeval` 的指针变量, 计时零点是 1970 年 1 月 1 日 0 时。用户空间的 `gettimeofday()` 系统调用函数中用的也是同一变量。`do_gettimeofday()` 的原型如下:

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

下面是一段获取当前时间的内核代码, 参考自 Jonathan Corbet 等人所著的《*Linux Device Drivers*》书中的示例模块 jit (Just In Time), 源代码可以从 O'Reilly 的 FTP 站点获得。

jit 模块将创建 /proc/currenttime 文件, 读取该文件将以 ASCII 码的形式返回三项:

- jiffies 的当前值
- 由 `do_gettimeofday()` 返回的当前时间
- 从 xtime 获得的当前时间

我们选择用动态的 /proc 文件, 是因为这样模块代码量会小些, 不值得为返回三行文本而写一个完整的设备驱动程序。

清单 4.7: 获取当前时间的内核代码 jit_current.c

```
/* jit_current.c */
...
static int opened = 0;
int currenttime_open(struct inode *inode,
                     struct file *filp)
{
    opened = 1;   /* read only once for each open */
    return 0;
}

ssize_t currenttime_read(struct file *file,
                         char *buffer,
                         size_t count,
                         loff_t *f_pos)
{
    int len;
    struct timeval tv1;
    struct timespec tv2;
```

```
    unsigned long j1;

    if (opened == 0)
        return 0;

    opened = 0;
    j1 = jiffies;
    do_gettimeofday(&tv1);
    tv2 = current_kernel_time();

    /*  print */

    len = sprintf(buffer, "0x%08lx %10i.%06i\n"
                          "%22i.%09i\n",
                          j1,
                          (int)tv1.tv_sec, (int)tv1.tv_usec,
                          (int)tv2.tv_sec, (int)tv2.tv_nsec);

    return len;
}

struct file_operations fops = {
    owner:   THIS_MODULE,
    open:    currenttime_open,
    read:    currenttime_read,
};

int init_module()
{
    proc_create("currenttime", 0666, NULL, &fops);
    return 0;   /*  everything is ok */
}

void cleanup_module()
{
    remove_proc_entry("currenttime", NULL);
}
```

如果用 `cat` 命令在一个时钟滴答内多次读该文件, 就会发现 xtime 和 `do_gettimeofday()` 两者的差异了: xtime 更新的次数不那么频繁。

```
$ cd /proc; cat currenttime currenttime currenttime
0x100066d66 1504249131.380391     (gettimeofday)
            1504249131.375766908  (kernel_time)
0x100066d66 1504249131.380480     (gettimeofday)
            1504249131.375766908  (kernel_time)
0x100066d66 1504249131.380548     (gettimeofday)
            1504249131.375766908  (kernel_time)
```

4.8.3 延迟执行

设备驱动程序经常需要将某些特定代码延迟一段时间后执行, 通常是为了让硬件能完成某些任务。

长延迟

如果想把执行延迟若干个时钟滴答, 或者对延迟的精度要求不高 (比如毫秒到秒量级的延时等待), 最简单的实现方法就是所谓的忙等待 (busy wait):

```
    unsigned long j = jiffies + jit_delay * HZ;


    while (jiffies < j)
                /* nothing */;
```

它的工作原理很简单: 因为内核的头文件中 jiffies 被声明为 `volatile` 型变量, 每次访问它时都会重新读取, 而该变量会在每次时钟中断时更新, 因此该循环可以起到延迟的作用。尽管也是正确的实现, 但应尽量避免使用, 无论在用户空间还是内核空间, 用这类方法实现延迟都不是好办法。忙等待循环在延迟期间会锁住处理器, 对于不可抢占内核, 调度器不会中断运行在内核空间的进程。如果碰巧在进入循环之前又关闭了中断, jiffies 值就不会得到更新, `while`循环的条件就永远为真, 这时就不得不重启计算机了。

同样使用 jiffies, 下面的例子是实现延迟较好的方法, 它允许其他进程在延迟的时间间隔内运行, 尽管这种方法不能用于硬实时任务或者其他对时间要求很严格的场合:

```
    while (jiffies < j)
        schedule();
```

这种循环延迟方法仍不是最好的。程序通过 `schedule()` 让出了 CPU, 运行系统调度其他任务, 但当前任务本身仍在队列中, `schedule()` 会被反复执行多次。如果它是系统中唯一的可运行的进程, 系统仍处于相当忙碌的状态 (系统调用调度器, 调度器选择同一个进程运行, 此进程又再调用调度器, ...), idle 进程 (进程号为 0。由于历史原因被称为 swapper) 绝不会被运行。系统空闲时运行 idle 进程可以减轻处理器负载、降低处理器温度、降低功耗、延长处理器寿命, 对于电池供电的设备, 还能延长电池的使用时间。而且, 延迟期间实际上进程是在执行的, 因此进程在延迟中消耗的所有时间都会被记录。反之, 如果系统很忙, 驱动程序等待的时间可能会比预计多得多。一旦一个进程在调度时让出了处理器, 无法保证在 jiffies 到达预计时间时能很快获得处理器的资源。如果对时间响应延迟有要求的话, 这种方式调用 `schedule()`, 对驱动程序来说并不是一个安全的解决方案。

获得延迟的最好方法, 是请求内核为我们实现延迟。如果驱动程序使用等待队列等待某个事件, 而又想确保在一段时间后一定运行该驱动程序, 可以使

用 wait_event() 函数的超时版本, 即 4.4.5 节介绍的 wait_event_timeout() 和 wait_event_interruptible_timeout()。这两个函数都能让进程在指定的等待队列上睡眠, 并在超时期限到达时返回。由此它们就实现了一种有上限的不会永远持续下去的睡眠。这里设置的超时值仍是要等待的 jiffies 数, 而不是绝对的时间值。

如果驱动程序无须等待其他事件, 可以用一种更直接的方式获取延迟, 即使用 schedule_timeout():

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (jit_delay * HZ);
```

上述代码行使进程进入睡眠直到指定时间。schedule_timeout() 处理一个时间增量而不是一个 jiffies 的绝对值。

短延迟

有时驱动程序需要非常短的延迟来和硬件同步。此时, 使用 jiffies 值无法达到目的 (HZ 为 250 时, 一个 jiffies 单位的时间是 4ms)。这时可以使用内核函数 ndelay()、udelay() 和 mdelay()。它们的原型如下:

```
#include <linux/delay.h>
void ndelay(unsigned long nanoseconds);
void udelay(unsigned long microseconds);
void mdelay(unsigned long milliseconds);
```

这些函数在绝大多数体系结构上是作为内联函数编译的。所有平台都实现了 udelay(), 而其他两个则未必。udelay() 使用软件循环延迟指定数目的微秒数, mdelay() 以 udelay() 为基础做循环, 用于方便程序开发。udelay() 函数里要用到 BogoMips[14] 值, 它的循环基于整数值 loops_per_second。这个值是在系统引导阶段计算 BogoMips 时得到的结果。目前大多数平台还没有实现纳秒级的延迟。

udelay() 函数只能用于获取较短的时间延迟, 因为 loops_per_second 值的精度只有 8 位, 所以, 当计算更长的延迟时会积累出相当大的误差。尽管最大能允许的延迟将近 1 秒 (因为更长的延迟就要溢出), 推荐的 delay() 函数的参数的最大值是取 1000 微秒 (1 毫秒)。延迟大于 1 毫秒时可以使用函数 mdelay()。

要特别注意的是 udelay() 是个忙等待函数 (所以 mdelay() 也是忙等待), 在延迟的时间段内无法运行其他的任务, 因此要十分小心, 尤其是 mdelay()。除非别无他法, 要尽量避免使用。

以上方法都在 jit 模块中实现, 有兴趣的读者可阅读源代码。

4.8.4 定时器

内核中最终的计时资源还是定时器。定时器用于调度函数 (定时器处理程序) 在未来某个特定时间执行。

内核定时器被组织成双向链表 (**struct hlist_node**)。这意味着我们可以加入任意多的定时器。定时器包括它的超时值 (单位是 jiffies) 和超时时要调用的函数。定时器处理程序需要接收一个参数, 该参数和处理程序函数指针本身一起存放在一个数据结构中。

定时器的数据结构如下 (linux/timer.h):

[14]"bogus" 和 MIPS 两个词的组合, 一种不太严谨的计算机速度评估指标。

```
struct hlist_node {
    struct hlist_node *next, **pprev;
};

struct timer_list {
    /*
     * All fields that change during normal runtime
     * grouped to the same cacheline
     */
    struct hlist_node   entry;
    unsigned long       expires;
    void            (*function)(unsigned long);
    unsigned long       data;
    u32         flags;

#ifdef CONFIG_TIMER_STATS
    int         start_pid;
    void        *start_site;
    char        start_comm[16];
#endif
#ifdef CONFIG_LOCKDEP
    struct lockdep_map  lockdep_map;
#endif
};
```

定时器的超时值是个 jiffies 值。当 jiffies 值大于等于 `timer->expires` 时, 就要运行 `timer->function()` 函数。

一旦完成对 `timer_list` 结构的初始化, `add_timer()` 函数就将它插入一张有序链表中, 该链表每秒钟会被查询 100 次左右。即使某些系统 (如 Alpha) 使用更高的时钟中断频率, 也不会更频繁地检查定时器列表。因为如果增加定时器分辨率, 遍历链表的代价也会相应增加。

用于操作定时器的有如下函数:

```
void init_timer(struct timer_list *timer);
```

该内联函数用来初始化定时器结构。目前, 它只将链表前后指针清零 (在对称多核处理器系统上还有运行标志)。强烈建议程序员使用该函数来初始化定时器而不要显式地修改结构内的指针, 以保证向前兼容。

```
void add_timer(struct timer_list *timer);
```

该函数将定时器插入活动定时器的全局队列。

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

如果要更改定时器的超时时间则调用它, 调用后定时器使用新的 expires 值。

```
int del_timer(struct timer_list *timer);
```

如果需要在定时器超时前将它从列表中删除, 则应调用 del_timer() 函数。但当定时器超时时, 系统会自动地将它从链表中删除。

```
int del_timer_sync(struct timer_list *timer);
```

该函数的工作类似 del_timer(), 不过它还确保了当它返回时, 定时器函数不在任何 CPU 上运行。当一个定时器函数在无法预料的时间运行时, 使用 del_timer_sync() 可避免产生竞争。大多数情况下都应该使用这个函数。调用 del_timer_sync() 时还必须保证定时器函数不会使用 add_timer() 把它自己重新加入队列。

jiq 示例模块是使用定时器的一个例子。/proc/jitimer 文件使用一个定时器来产生两行数据, 所使用的打印函数和前面任务队列中用到的是同一个。第一行数据是由 read() 调用产生的 (由查看 /proc/jitimer 的用户进程调用), 而第二行是 1 秒后定时器函数打印出的。

用于 /proc/jitimer 文件的代码如下所示:

清单 4.8: 内核使用定时器的例子 jitimer.c

```
/* jitimer.c */
...
#define LIMIT   (PAGE_SIZE-128)

static DECLARE_WAIT_QUEUE_HEAD (jiq_wait);

static struct clientdata {
    struct seq_file *m;
    unsigned long jiffies;
    int wait_cond;
} jiq_data;

/*
* 打印, 任务重新调度时返回非零
*/
static int jiq_print(void *ptr)
{
    struct clientdata *data = ptr;
    struct seq_file *m = data->m;
    unsigned long j = jiffies;

    if (m->count > LIMIT) {
        data->wait_cond = 1;
        wake_up_interruptible(&jiq_wait);
        return 0;
```

```
    }

    if (m->count == 0)
        seq_puts(m,"    time  delta preempt
pid cpu command\n");

    seq_printf(m, "%9li  %4li     %3i %5i %3i %s\n",
               j, j - data->jiffies,
               preempt_count(), current->pid, smp_processor_id(),
               current->comm);

    data->jiffies = j;
    return 1;
}

static void jiq_timedout(unsigned long ptr)
{
    struct clientdata *data = (void*)ptr;
    jiq_print((void *)data);            /* print a line */

    data->wait_cond = 1;               /* 睡眠结束的条件 */
    wake_up_interruptible(&jiq_wait);  /* 唤醒 */
}

static int jiq_read_run_timer(struct seq_file *m, void *v)
{
    struct timer_list jiq_timer;

    jiq_data.m = m;
    jiq_data.jiffies = jiffies;
    jiq_data.wait_cond = 0;

    init_timer(&jiq_timer);          /* 初始化定时器结构 */
    jiq_timer.function = jiq_timedout;
    jiq_timer.data = (unsigned long)&jiq_data;
    jiq_timer.expires = jiffies + HZ;   /* 准备睡眠 1 秒 */

    jiq_print(&jiq_data);           /* 打印数据，睡眠 */
    add_timer(&jiq_timer);
```

```
    wait_event_interruptible(jiq_wait, jiq_data.wait_cond);
    del_timer_sync(&jiq_timer);  /* 删除定时器 */

    return 0;
}

static int jiq_read_run_timer_proc_open(struct inode *inode,
                                        struct file *file)
{
    return single_open(file, jiq_read_run_timer, NULL);
}

struct file_operations jiq_proc_fops = {
    open : jiq_read_run_timer_proc_open,
    read : seq_read,
};

int init_module(void)
{
    proc_create("jitimer", 0, NULL, &jiq_proc_fops);
    return 0; /* succeed */
}
...
```

模块加载后, 运行命令 cat /proc/jitimer 得到如下输出结果:

```
       time  delta preempt   pid cpu command
  4296512102     0       0 12459   0 cat
  4296512352   250     256     0   0 swapper/0
```

注意 command 一列, 第一行是访问 /proc/jitimer 的命令 (可以换成 head、more 命令尝试一下), 而第二行与用户进程无关。

定时器是竞争资源, 即使是在单处理器系统中。定时器函数访问的任何数据结构都要进行保护以防止并发访问, 保护方法可以用原子类型或者用自旋锁。删除定时器时也要小心避免竞争。考虑这样一种情况: 某一模块的定时器函数正在一个处理器上运行, 这时在另一个处理器上发生了相关事件 (文件被关闭或模块被删除)。结果是, 定时器函数等待一种已不再出现的状态, 从而导致系统崩溃。为避免这种竞争, 模块中应该用 del_timer_sync() 代替 del_timer()。如果定时器函数还能够重新启动自己的定时器 (这是一种普遍使用的模式), 则应该增加一个 “停止定时器” 标志, 并在调用 del_timer_sync() 之前设置。这样定时器函数执行时就可以检查该标志。如果已经设置, 就不会用 add_timer() 重新调度自己了。

还有一种会引起竞争的情况是修改定时器: 先用 del_timer() 删除定时器, 再用 add_timer() 加入一个新的定时器以达到修改目的。其实在这种情况下简单地使用

`mod_timer()` 是更好的方法。

■ 本章练习

1. 内核模块和应用程序结构上有什么不同?
2. 内核模块 `init_module()` 强行返回 −1, 还可以继续使用 `rmmod` 将其移除吗? 为什么?
3. 在开发设备驱动程序的过程中, 错误的操作常常导致驱动模块崩溃, 占用的设备号无法释放。在不重启设备的前提下, 有办法释放这种情况被占用的设备号吗?
4. 当一个设备驱动程序被安装 (`insmod`) 后, 怎样查到该设备对应的设备号?
5. 能否将 /dev 目录下的一个设备文件通过 cp 命令复制到个人的目录? 如果不能, 原因是什么?
6. 不使用 `ioctl()` 方法, 如何实现用户程序与设备驱动之间的数据和控制/状态字的交换? 仍以定时器 Intel8254 为例, 简述实现方案。
7. 试分别用 jiffies、TSC 和 `udelay()` 或 `mdelay()` 设计 60 分钟的计时, 测试它们的精度。

第 5 章

数据传输

总线是计算机各模块之间进行信息传输的通道。它包括通道控制、仲裁方法和传输方式等基本内容。

模块间的数据通过数据总线进行传递。各种传递方式中, 中断方式在微处理器系统中具有极其重要的地位。*DMA* 是高速大量数据传递的有效手段。

5.1 模块间的数据传递

在功能上相对独立的电路称作一个模块 (module)。模块可以小到只有一个元件构成, 也可以复杂到由大量元件构成的系统。模块与模块经适当组合可以构成具有新的功能的更高级别的模块。微处理器系统的模块与模块之间通过总线连接在一起, 在 CPU 的统一指挥下协调工作。

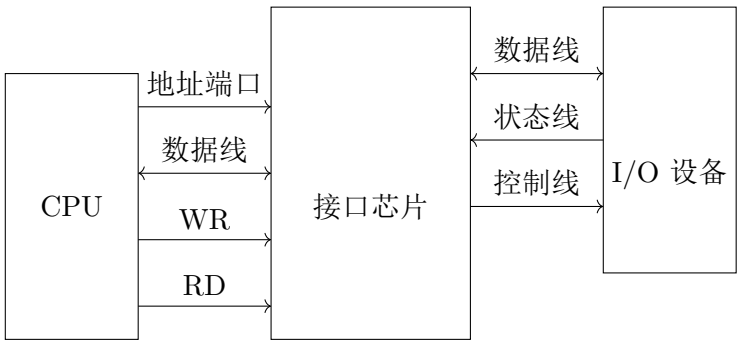


图 5.1: CPU 与外设间的接口信号

CPU 与其它模块之间的输入输出数据传递通常需要经过接口器件作为桥梁。图 5.1 是 CPU、外设和接口器件之间数据传递通常需要的信号。这些信号按功能可以分为如下四类:

数据信号 在 CPU 与 I/O 接口之间或接口与外设之间传递数据。通常是 8 位或 16 位位宽的二进制数字量。

地址信号 CPU 通过系统地址线 (或经过地址译码) 连接到接口芯片, 以区分不同的设备。

控制信号 控制 I/O 接口芯片或通过接口芯片控制外设的信号, 如读写、选通等信号。

状态信号 外设通过 I/O 接口芯片向 CPU 反映设备工作的状态信息, 例如设备是否准备就绪、缓冲器是否空, 等等。

根据 CPU 在数据传输过程中起的作用, 可以将数据传输方式分成直接数据传输 (或称无条件数据传输)、查询数据传输 (或称轮询数据传输) 和中断数据传输三种方式; 借助 DMA 控制器, 还有第四种方式: 直接存储器存取 (**D**irect **M**emory **A**ccess, DMA)。

5.1.1 直接数据传输

这种传输方式中, CPU 与外设之间没有联络信号, CPU 根据自己的需要和外设直接交换数据。它没有在查询方式下的查询条件，故也称无条件数据传输。交通路口信号灯和倒计时牌的控制就是这种情况。这种方式不保证数据传输的可靠性。

需要说明的是，与设备无关的条件不在考虑之内。例如交通信号灯变化的时间是触发控制信号的基础, 但这个时间与信号灯设备无关。CPU 并不知道信号灯是否工作正常，它只按自己的需要发送控制数据。

5.1.2 查询数据传输方式

CPU 和外设之间通过一些联络信号进行协调, CPU 通过查询 (polling) 这些联络信号完成与外设的数据交换, 在一定程度上保证了数据交换的正确性 (图 5.2)。

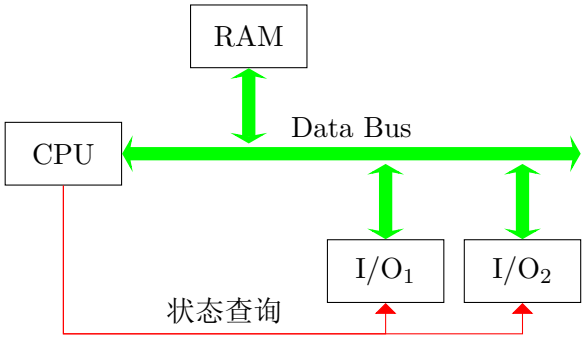


图 5.2: 查询方式

查询方式下, CPU 不断地循环查询外设的状态, 直到外设做出响应为止。软件运行完全被外设同步。由于 CPU 通常比外设的速度快得多, 故最大传输速度取决于外设的响应时间。假设某外设 (如打印机) 的处理能力为 100 字节/秒, 则传输 100 字节就需要 1 秒的时间，而实际上执行有效数据传输的指令只有 100 条。CPU 把大量的时间花在等待中。

5.1.3 中断传输方式

这种方式中, CPU 不主动查询外设的联络信号, 而是被动地等待外设的中断请求, 由 CPU 在中断服务中完成数据交换。这种方式可以省去查询方式下 CPU 耗费的大量时间, 有效地提高 CPU 的工作效率 (图5.3)。

如果采用中断 (interrupt) 方式, CPU 可以在启动外设后就去执行自己的任务, 直到外设请求中断为止。CPU 在中断服务程序中完成数据传输, 无须等待时间。此时的数据传输时间取决于中断服务所需要的时间, 而和外设的速度没有直接关系。同样以 100 字节/秒的打印机为例, 如果每传输一个字节的中断服务程序为 10 微秒 (主频 100MHz 的处理器，100 微秒大致可以执行 1 千条指令), 完成 100 个字节的数据传输只需 1 毫秒。

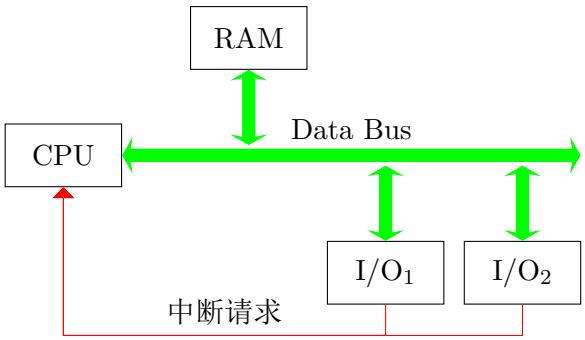


图 5.3: 中断方式

5.1.4 DMA 传输方式

这种传输方式下, 数据传输由一个可以获得总线控制权的设备 DMAC (DMA Controller) 在外设与系统内部存储器之间进行数据交换。不需要 CPU 在数据传输过程中干预, CPU 只需要在数据传输之前做一些准备工作 (如为 DMAC 编程)。这种方式适用于大量数据的高速传输。

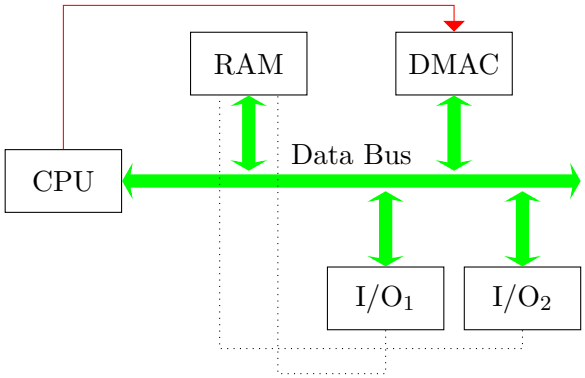


图 5.4: DMA 方式

前面提到的打印机的例子, 如果同样的传输采用 DMA 方式完成, 每个存取周期可以完成一个字节的数据传输, 总时间就是 100 个存取周期, 软件开销将进一步降低。并且这个数据传输过程无需 CPU 参与, 不会破坏 CPU 的流水线。

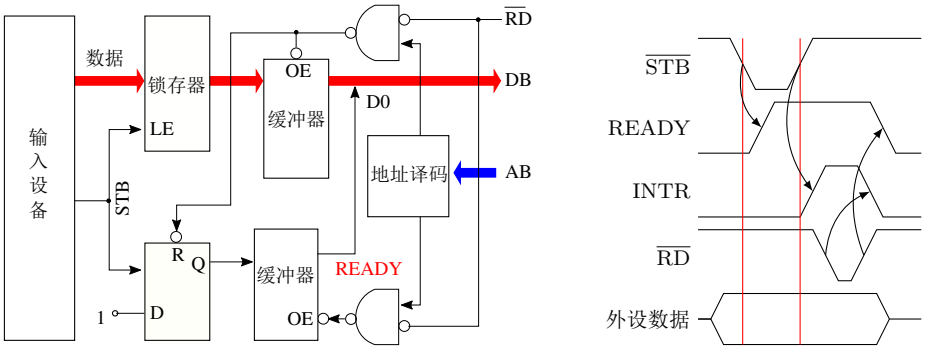
5.2 查询输入输出

接口电路中, CPU 通过特定的端口和外设交换数据和信息。这样的数据传输实际上就是对这个端口的 “读” 或者 “写”。除要求作为数据交换的端口外, 在查询方式下还需要一个状态端口, 以便 CPU 了解外设的工作状态。在需要可靠的双向数据传输场合, 应考虑完善的握手信号, 保证输入设备未将上次数据取走之前, 输出设备不发送新的数据, 或者在新数据未到达之前, 不重复读取旧的数据。

在程序控制下的 I/O 传输中, CPU 首先查询状态口, 了解外设的工作状态。如果外设就绪, CPU 就可以通过输入或输出的指令操作完成写或读的工作。

5.2.1 查询输入

图 5.5(a) 是典型的查询输入电路。外设数据准备好后, 发出一个选通信号 STB, 将数据送入锁存器, 同时 (或稍后) 使 D 触发器置位, 输出就绪信号 READY。CPU 通过读数据总线的相应位查询到有效的 READY 信号后即可进行读操作, 读取缓冲器中的数据, 并在读的同时复位 D 触发器, 清除 READY 信号, 为下一次输入做好准备。图 5.5(b) 是选通方式的一种可能的协议。



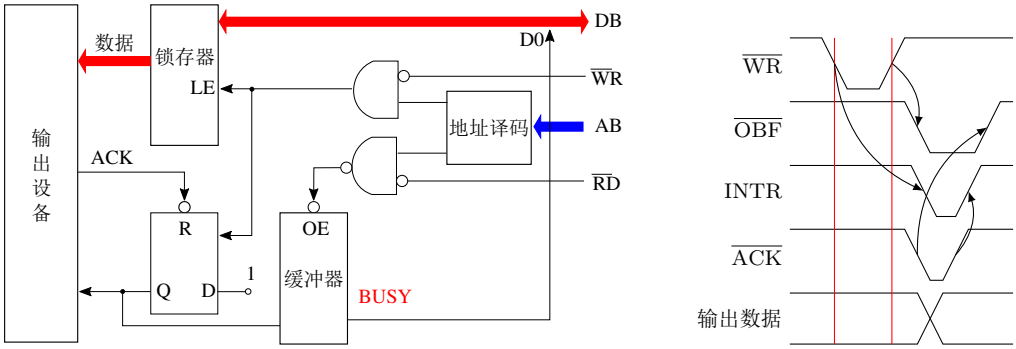
(a) 查询输入的接口电路 (b) 选通输入方式协议

图 5.5: 查询输入

5.2.2 查询输出

查询输出的接口典型电路如图 5.6(a)。CPU 在输出数据前, 也必须了解外设的工作状态, 确定外设是否可以接收数据。

输出时, CPU 发出的写信号 $\overline{\text{WR}}$ 将数据打入锁存器, 并使输出缓冲器满 $\overline{\text{OBF}}$ 有效, 同时复位中断请求 INTR; 当外设接收了数据后, 回送 $\overline{\text{ACK}}$, 此信号清除 $\overline{\text{OBF}}$, 同时产生中断请求 INTR, 进入下一轮的输出过程。只要 CPU 每次输出前检查 $\overline{\text{OBF}}$ 的有效性，就可以确保外设在读取上一个数据之前不会用新的数据覆盖。图 5.6(b) 是输出方式的协议。



(a) 查询输出的接口电路 (b) 选通输出方式协议

图 5.6: 查询输出

5.3 中断

5.3.1 中断的概念

微处理器在执行正常的程序过程中, 当出现某种异常事件或某种特殊请求时, 暂时中止正在执行的程序, 转向对这个异常事件或特别请求的处理过程; 处理完毕后再回到原来的断点继续刚才的任务。这一过程被称为中断。

中断是提高微处理器效率的手段, 它使得微处理器具有初步并行工作的能力。中断服务与子程序调用有相似之处: 子程序和中断服务子程序都是事先安排好的任务; 不同点在于, 对子程序的调用是在程序运行的确定位置实现的, 而中断的发生位置则是不确定的。因此, 子程序调用会有明确的指令, 而中断调用通常却不会有明确的指令 (软中断除外)。

按与 CPU 的关系密切程度划分，中断有内部 (internal) 中断和外部 (external) 中断。内部中断是 CPU 自身原因引起的, 例如除以 0 就会引发一个内部中断, 现代处理器把这类情况归入异常 (exception)。软件中断也是内部中断的一种形式, 它是 CPU 的一条指令。在操作系统中, 软中断常常用于构造系统功能调用。外部中断是由外部设备产生的, CPU 通常会引出一些外接引脚, 用于接收这种中断信息。

在外部中断中, 又分为可屏蔽 (maskable) 中断和不可屏蔽 (non-maskable) 中断两类。在一段程序中，CPU 认为此项工作不能被其他事件干扰, 可以通过一条禁止中断指令屏蔽外部的可屏蔽中断。不可屏蔽中断的意义在于, 它不能被 CPU 的指令禁止, 因此通常将不可屏蔽中断的功能用于紧急事务的处理, 如系统意外掉电时的现场保护等。

当系统存在多个中断源时, 应按外设服务任务的轻重安排适当的优先级 (priority)。CPU 对各级中断处理的一般原则是: 不同级别的中断同时发生, 首先响应高优先级的请求; 低优先级中断服务过程中, 高优先级的中断可以进行嵌套; 同级别的中断请求同时发生时, 按事先安排好的顺序依次处理。CPU 遵循的优先级原则是: CPU 内部中断的优先级高于外部中断, 外部中断的优先级由中断控制器 (interrupt controller) 通过程序安排。除了安排中断优先级, 中断控制器还可以有选择地屏蔽某些中断。

5.3.2 中断的处理过程

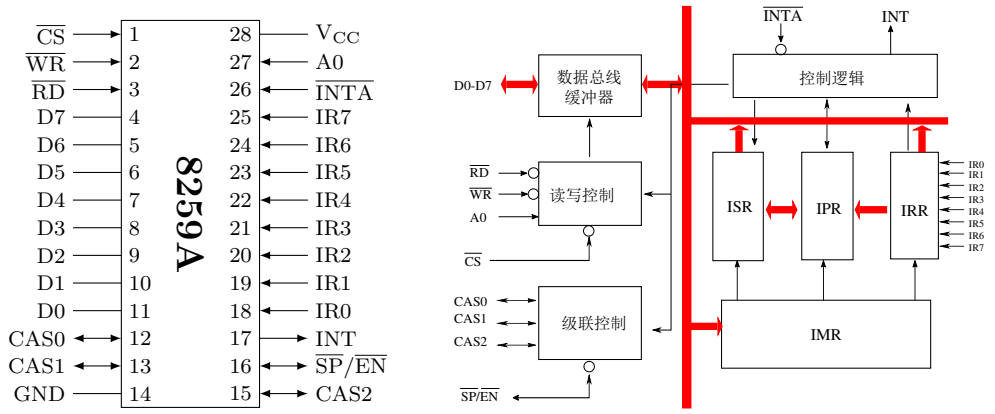
中断处理过程有以下几个步骤:

1. 中断请求。外设需要中断服务时, 向 CPU 发中断请求信号, 送到 CPU 的中断请求线上。
2. 中断响应。CPU 在每条指令执行的最后阶段检测中断请求信号。当得到有效电平或有效边沿、且 CPU 处于中断开放状态时, 将对该信号予以响应。此时, CPU 自动完成以下工作:
   - 关中断, 以免在中断现场尚未保护的情况下再次响应新的中断, 从而导致处理逻辑上的混乱。
   - 保护断点。这里的断点指 CPU 响应中断前程序计数器中保留的下一条指令地址。该地址的值通常会被压入堆栈, 以便中断结束后取回该地址返回原断点。
   - 程序转到该设备设定的中断处理程序首地址。
3. 中断处理。CPU 进入中断处理程序首地址后, 顺序执行中断服务子程序的代码, 完成中断源请求的处理操作。中断处理程序一般包括保护现场、有选择地开放中断、服务程序、恢复现场等工作。所谓保护现场, 是将中断服务中所有用到的寄存器进行保护, 以便在返回断点之前恢复原来的状态; 开放中断的目的在于能够让高优先级的中断继续得到响应, 实现中断嵌套。

4. 中断返回。同子程序类似, 从堆栈栈顶获得返回地址指针, 返回到中断的断点, 与子程序不同的是, 中断响应时还将标志寄存器（或机器状态字）保存在了堆栈，返回时也要恢复这个状态。

5.3.3 中断控制器

早期 PC 的中断控制器使用 8259A, 目前的 PC 中已不存在这个独立的芯片, 但其控制逻辑仍然保留。本节就以 8259A 为例介绍其控制方法。单片 8259A 可直接管理 8 个外设的中断请求, 可以实现优先级判别、提供中断向量, 在级联 (cascade) 方式下可管理多达 64 级的中断输入。其外部引脚和内部结构如图 5.7 所示。



(a) 8259A 的引脚 (b) 8259A 的内部结构

图 5.7: 8259A

引脚说明如下:

1. D0~D7: 双向三态数据线, 直接和系统数据总线相连, 实现与 CPU 的数据交换;
2. $\overline{\text{CS}}$: 片选输入, 低电平有效;
3. $\overline{\text{RD}}$: 读输入信号;
4. $\overline{\text{WR}}$: 写输入信号;
5. CAS0~CAS2: 级联线;
6. $\overline{\text{INTA}}$: 中断响应输入;
7. INT: 中断请求输出信号。8259A 通过它向 CPU 发送中断请求;
8. IR0~IR7: 外部 I/O 设备或级联从片输入的中断请求信号;
9. $\overline{\text{SP}}/\overline{\text{EN}}$: 主从设备的设定/缓冲器读写控制, 输出;
10. A0: 地址选择输入信号, 用于对 8259A 的可编程寄存器进行选择, 通常与地址总线连接。

8259A 的内部结构

8259A 的内部结构从功能上分为三组:

1. 中断请求与响应

   IR0~IR7 为中断请求输入, 它们既可以是沿触发, 也可以是电平触发, 由编程决定。

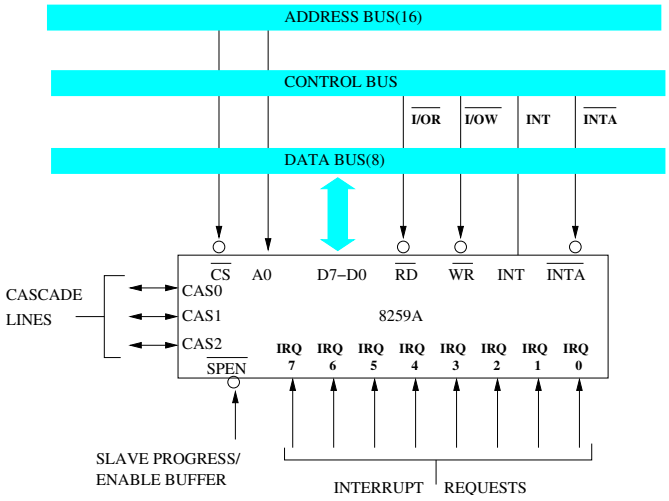


图 5.8: 8259A 与标准系统总线的接口

INT 和 $\overline{\text{INTA}}$ 分别为中断请求输出和中断响应应答, 和 CPU 相应的引脚相连。中断请求信息存放在中断请求寄存器 (**I**nterrupt **R**equest **R**egister, IRR) 中; 中断屏蔽寄存器 (**I**nterrupt **M**ask **R**egister, IMR) 中存放的位屏蔽信息, 可以禁止 IRR 中相应位的中断请求。8259A 根据 IRR 中各位的优先级和 IMR 中的屏蔽位状态, 经过优先级判别电路后, 将最高优先级的中断类型码通过 D0~D7 送给 CPU; 在得到 CPU 对它的回应 $\overline{\text{INTA}}$ 后, 将其记录在在服务寄存器 (**I**n-**S**ervice **R**egister, ISR) 中。

2. 级联

   CAS0~CAS2 作为 8259A 的级联专用总线, 可构成多个 8259A 的主–从级联控制结构。当 8259A 作为主设备时, CAS0~CAS2 为输出。组内的其它 8259A 均为从设备, CAS0~CAS2 为输入信号, 通过级联信息被主设备确认。

   $\overline{\text{SP}}/\overline{\text{EN}}$ 是双向信号, 在缓冲方式下为输出功能 $\overline{\text{EN}}$, 允许缓冲器接收和发送; 在非缓冲方式下为输入, 标志该 8259A 是主设备 ($\overline{\text{SP}}$=1) 还是从设备 ($\overline{\text{SP}}$=0)。

3. 控制逻辑

   用来与 CPU 控制信号连接。CPU 将命令送入 8259A 的寄存器或从 8259A 中读取 IRR、ISR 或 IMR 的内容。

8259A 的初始化

CPU 对 8259A 发出的命令分为两类, 一类是初始化命令 (**I**nitial **C**ommand **W**ord, ICW), 一类是操作命令 (**O**perate **C**ontrol **W**ord, OCW)。8259A 在工作之前首先必须初始化, 初始化之后任何时候都可以进行操作。

8259A 有四条初始化命令, 按规定的顺序送入:

- ICW1, 其特征标志为 A0=0, D4=1。

A0	D7	D6	D5	D4	D3	D2	D1	D0
0	A7	A6	A5	1	LTIM	ADI	SNGL	IC4

A7、A6、A5: 中断向量地址的 A7~A5 (仅用于 MCS80/85)

LTIM: 中断触发方式。0: 上升沿触发, 1: 高电平触发,

ADI: 0: 调用地址间隔为 8; 1: 调用地址间隔为 4 (X86 系统不用)

SNGL: 单片或级联。0: 级联 1: 单片

IC4: IC4=1 时, 需要配置 ICW4。8086 系统要求 ICW4

- ICW2

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	A15/T7	A14/T6	A13/T5	A12/T4	A11/T3	A10	A9	A8

该命令字设置中断向量。在 X86 系统中, 仅设置中断类型码的高 5 位 T7~T3, 低 3 位由 8259A 根据 IR0~IR7 自动插入编码。

- ICW3

ICW1 中的 SNGL 位为 0 时, 8259A 处于级联工作方式, 需要 ICW3 分别设置主、从设备的状态。对于主设备:

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	S7	S6	S5	S4	S3	S2	S1	S0

ICW3 装入 8 位从设备标志, 置位的位表示对应的 IR 端接至从设备的 INT 输出。中断响应时, 中断向量由对应的从设备发送。

对于从设备:

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	×	×	×	×	×	ID2	ID1	ID0

其中三位 ID 为从设备的标志码, 等于主设备对应的 IR 输入端编码。中断响应时, 主设备把 IRx 编码送至级联线 CAS2~CAS0, 当与某个从设备相同时, 该从设备被选中。

- ICW4, 在 ICW1 中的 D0=1 时, 需要预置。

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	SFNM	BUF	M/$\overline{\text{S}}$	AEOI	μPM

SFNM: 特殊全嵌套方式

BUF: 缓冲方式

M/$\overline{\text{S}}$: 主从标志位。1: 主片, 0: 从片

AEOI: 自动中断结束

μ PM: 8086 系统中, 该位应为 1

8259A 对中断进行管理的核心是对中断优先级的管理。8259A 对中断优先级的设置方式有四种: 全嵌套方式 (full nested mode)、特殊全嵌套方式 (special full nested mode)、优先级自动循环方式和优先级特殊循环方式。

1. 全嵌套方式

全嵌套方式是 8259A 最基本的工作方式。如果对 8259A 进行初始化后没有设置其他优先级方式, 则自动按这种方式工作。在这种工作方式下, 各中断请求信号的优先级是固定的, IR0 最高, IR7 最低。当一个中断请求 IRx 被响应时, 中断服务寄存器 ISR 中的相应位被置 1。一般情况下, 该位将一直保持到 CPU 发出中断结束 EOI, 以便为优先级判别器 PR 提供依据。

当有新的中断申请时, 优先级判别器 PR 将该中断申请与当前服务寄存器 ISR 中置 1 的位进行比较, 判别新收到的中断请求是否比当前正在处理的中断优先级更高。若是, 且该位未被屏蔽, 则准备实行中断嵌套, 该位对应的中断请求送往 CPU。如果得到 CPU 的响应, 就将 ISR 寄存器中新申请中断对应位置 1。

2. 特殊全嵌套方式

   所谓特殊全嵌套方式, 是指当一个中断源被响应后, 仅屏蔽低级的中断请求, 而允许同级和高级中断申请的一种优先级管理方式。该方式一般用于 8259A 级联系统中, 主要解决对来自从片的同级申请的响应问题。其他情况与全嵌套方式相同。

3. 自动循环方式

   这种方式使用于系统中具有相等优先级中断设置情况。如果 8259A 按照这种方式工作, 则刚刚获得服务的中断设备在中断结束后变为最低优先级。因此这个设备再次请求中断时, 在与其他中断请求进行优先级比较时就处于低位。在最坏情况下, 可能要等到其他 7 个设备都被服务一次以后, 才轮到它再次服务。

   自动循环方式可通过操作命令字 OCW2 来设置。

4. 特殊循环方式

   在需要一次改变优先级别的场合, 可以使用指定优先级循环命令。随着最低优先级的设置, 依次决定其他引脚的优先级。例如最初由软件设 IR2 为最低级, 则 IR3 就为最高优先级。其他依次类推。

   特殊循环方式及最初的最低优先级中断源的设置可用操作命令字 OCW2 来实现。

8259A 的工作控制

设置完 ICW1 到 ICW4 后, 可以在任何时候以任何顺序用 OCW1 到 OCW3 访问 8259A 的内部寄存器。

- OCW1, 中断屏蔽控制。

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

该命令设置中断屏蔽寄存器。被置 1 的位, 相应 IR 引脚的中断请求被屏蔽。通常, CPU 有自己的中断禁止指令, 可以禁止所有的可屏蔽中断的进入, 即屏蔽所有的中断源。但如果要想屏蔽某个或某几个中断源, 就需要中断控制器来管理了。8259A 通过 IMR 的屏蔽方式来实现单个中断的屏蔽。8259A 对中断源的屏蔽的方式有两种:

普通屏蔽方式 通过 OCW1 设置 8259A 的中断屏蔽寄存器 IMR。该寄存器与 8 个中断源 IR0~IR7 相对应。当屏蔽字中某一位或某几位为 1 时, 与这些位对应的中断源被屏蔽。

特殊屏蔽方式 常用于多级中断嵌套中。在具有较高优先级的中断程序中设置特殊屏蔽方式, 允许低级中断打断高级中断。

- OCW2, 优先级管理方式。其特征标志为 A0=0, D4=0, D3=0:

A0	D7	D6	D5	D4	D3	D2	D1	D0
0	R	SL	EOI	0	0	L2	L1	L0

R SL EOI: 优先级设置和中断结束

0 0 1 一般的 EOI 命令

0 1 1 特殊的 EOI 命令

1 0 1 循环优先级的一般的 EOI 命令

1 0 0 设置循环 AEOI 方式

0 0 0 复位循环 AEOI 方式

1 1 1 循环优先级的特殊的 EOI 命令

1 1 0 设置优先级命令

0 1 0 无操作

L2 L1 L0: 对应的 IR 编码

- OCW3, 设置中断查询方式、屏蔽方式等。特征标志为 A0=0, D4=0, D3=1:

A0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	ESMM	SMM	0	1	P	RR	RIS

ESMM: 0: 禁止 SMM, 1: 允许 SMM

SMM: 0: 复位特殊屏蔽方式, 1: 设置特殊屏蔽方式

P: 1: 查询命令

RR: 读寄存器。0: 禁止, 1: 允许

RIS: 0: 读 IRR, 1: 读 ISR

5.4 串行通信

串行通信是指数据按位传输的一种通信方式。并行接口虽具有速度快、效率高等特点, 但在远距离数据传输时却受到了限制。而串行方式所用的传输线路少, 适用于远距离通信。计算机和远程终端、终端和终端之间的通信一般均采用串行通信。串行通信主要要解决的问题是, 如何在这些串行传输的信息流中识别位信息, 并进一步识别相关位的集合, 即一个“字”。这就涉及位同步、字同步的问题。此外还有计算机终端和数据传输设备之间握手控制、串行通信协议等诸多问题。

5.4.1 串行通信的特点

由于微处理器内部的数据总线是并行的。串行接口的基本功能之一是进行串行数据和并行数据之间的转换。在串行输出时, 需要把并行数据转换成串行数据, 在一条输出线上顺序逐位的发送出去; 接收时, 从一条输入线逐位接收串行数据, 并把它们转换成并行数据, 然后存入接收数据寄存器。输入输出时, 通过移位寄存器 (shift register) 实现串行与并行的转换功能。异步方式下, 接口电路在发送时自动生成起始位和停止位, 在接收时将其去

除。面向字符的同步方式下, 接口电路在发送数据块前面加同步字符, 在接收时去除同步字符。

串行通信中, 按照通信方向, 可分为三种基本的传送方式: 单工、半双工和全双工方式[1]。根据收发双方是否具有统一的时钟, 又可分为同步 (synchronous) 传输和异步 (asynchronous) 传输两种基本方式。

波特率 (baudrate) 是表示串行通信速度的物理量, 它的单位是每秒数据通信的位数, 单位是 bps。由于通信本身不以字节为单位, 所以通常不应该将波特率除以 8 折算成 B/s, 这种算法只能作为一种粗略的估算。

一般来说, 串行通信要解决以下一些问题:

- 位的判决。这可由提供统一时钟, 或从接收的数据中提取位时钟来解决。
- 字的识别。通常由规范的数据信息格式来获得。
- 串行数据传输距离一般从几米到几千公里, 传输数据的可靠性也是必须解决的问题。常采用差错控制的方式。
- 通信接口中要考虑终端与传输设备的握手关系及网络的需求。这就需要一些通信协议来保障。

5.4.2 串行通信的传输过程

在异步方式下, 收发两端各自有相互独立的位定时时钟。数据的传输速率是双方约定的。收发双方利用数据本身来进行同步。当双方时钟偏差不大时，只要一个同步周期内的位时钟不超过 1 个传输位，就可以满足正确接收的条件。目前常见的一种异步收发器 (**U**niversal **A**synchronous **R**eceiver and **T**ransmitter, UART) 一帧信息由起始位、数据位、奇偶校验位和停止位四部分组成, 如图 5.9 所示。

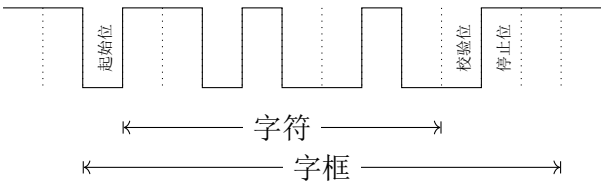


图 5.9: 异步串行通信的字框

- 起始位: 高电平是通信线的休闲状态, 起始位是在高电平后的个一位宽的低电平, 其下降沿表征了同步关系, 通知接收方传输开始。
- 数据位: 5 位至 8 位, 具体由双方通过软件约定。数据位紧跟在起始位后, 是传送的信息的主体。传送时低位在前, 高位在后。
- 校验位: 进行数据的奇偶检查。也可以不设校验位。
- 停止位: 可以是 1 位、1 位半或 2 位, 具体由收发双方约定。停止位是高电平, 标志着一个字符的结束, 也为下一个字做准备。

异步串行传输数据时, 字符作为一个独立的信息单位出现在数据流中。收发两端可以拥有相对独立的局部时钟, 因此接收端必须在发现起始位后的一个字符长度的周期内正确地识别数据位, 即要求接收器对每一位的采样点尽可能靠近位周期的中点。

[1]只能单向传送数据的方式称为单工 (simplex); 通过切换可以在不同时刻完成接收或发送工作的称为半双工 (half duplex) 方式; 可以同时完成接收或发送工作的称为全双工 (full duplex) 方式。

1. 发送过程

   CPU 把数据写入发送数据寄存器, 然后由发送器逻辑对数据进行格式化, 即加上起止位、奇偶检验位和停止位。格式化后的数据送到移位寄存器, 然后按设定的波特率进行串行输出。在数据送出后, 接口设备给出 “发送数据寄存器空” 的信号, 通知 CPU 可以送下一个数据。

2. 接收过程

   串行数据线休闲状态是高电平。一旦它由高变低, 并在半个位宽后仍是低电平时, 接收控制部分便开始工作, 每隔一个位宽的时间就采样数据线, 直至采样到停止位。接着, 差错检测逻辑按格式对数据进行校验, 并根据校验的结果置状态寄存器。如果发现有关错误, 则相应错误标志置位。异步串行通信通常会标示以下错误:

   - 奇偶校验错 (Parity Error)

     接收器按照约定的方式 (“1” 的个数是奇数还是偶数) 进行奇偶校验计算, 然后将奇偶检验的期望值和它的实际值进行比较。如果两者不一致, 便把奇偶错状态置位, 以便查询; 也可以选定在检测到奇偶错时, 产生中断请求, 执行中断服务程序进行处理。该错误通常不禁止接口器件的工作。
   - 帧错 (Framing Error)

     以起始位开头、停止位结束的二进制序列成为一帧。接收时, 当接收器没有接收到足够的停止位, 即为帧错, 相应的状态位置位, 认为字符的同步已经丢失。产生错误的原因很多, 可能是接收器或发送器的问题、传输线路上的干扰或发送器时钟误差超过允许值等等。接收器只能报错, 不能指明原因。
   - 溢出错 (Overrun Error)

     接收数据时, 当接收移位寄存器接收到一个正确的字符数据, 就会把移位寄存器的数据并行装入接收数据寄存器, 微处理器需要及时地取走这个数据。如果微处理器不能及时把数据取走, 后一个数据就会覆盖前一个数据而造成数据丢失。此时就发生了溢出错, 差错检测逻辑就把相应的溢出错标志位置位。

同步传输方式是相对于异步传输方式而言的, 主要是指时钟的同步, 即收发双方采用统一时钟的传输方式。可以利用通过双方的一条时钟信号的信道, 也可以利用独立的同步信号来获取时钟。由于同步传送的位判决来自收发两端采用的统一时钟, 其数据位的准确同步显得更为重要。同步传输不是用起始位来标志开始, 而是用一串特定的二进制序列。一次同步传输的有效数据位可以比异步传输的位数要多得多, 具有较高的传输效率。

5.5 直接存储器存取

普通的数据传输方式都需要 CPU 参与, 存储器和外设之间的数据传输要经过 CPU 的中转。在批量数据传输中, CPU 首先需要从源数据的端口或地址上读入数据, 再将数据写入目的端口或存储单元地址, 然后调整源和目的地址, 计数器加一, 如此反复。传输一个数据常常需要数条指令, CPU 的效率很低。

直接存储器存取 (DMA), 也称数据通道方式。这种方式下, 无需 CPU 干预, 直接由硬件 DMA 控制器控制总线, 完成数据传输。减少了中间环节, 大大提高了传输效率。DMA 主要应用于高速大批量数据传输中, 如磁盘存取、图像传输、高速数据采集系统等。

作为 DMA 数据传输的主要部件 DMA 控制器应具有如下功能:

- 向 CPU 申请 DMA 传输;

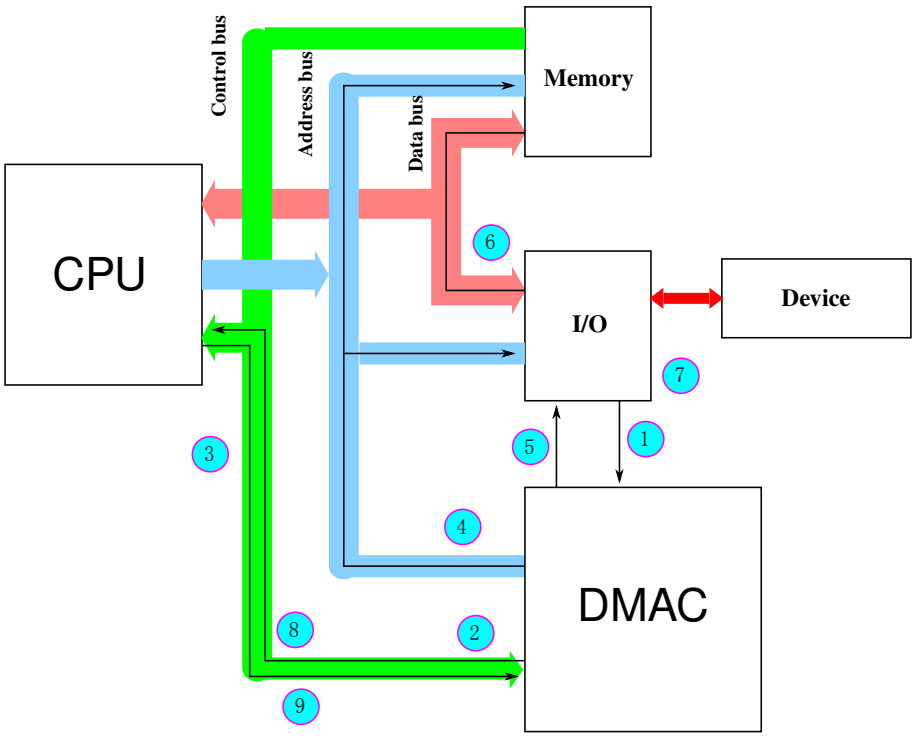


图 5.10: DMA 传输方式的处理流程

- 在 CPU 允许 DMA 方式工作时处理总线控制权的转交, 避免因 DMA 影响 CPU 的正常活动, 或引起总线竞争 (race condition);
- DMA 期间管理系统总线, 控制数据传输;
- 确定数据传输的起始地址和数据长度, 修正传输过程的数据地址;
- 传输结束时给出 DMA 操作完成的信号。

使用 DMA 方式进行输入输出操作, CPU 必须对指定的 DMA 通道设定工作方式, 指出要传输的数据在存储器中的起始地址和需要传输的字节数。完成这些初始化工作后, 才能启动 DMA 进行工作。

图 5.10 是 DMA 传输方式的处理流程示意。

1. 外设首先向 DMA 控制器发出 DMA 请求信号, 要求进行数据传输;
2. DMAC 向 CPU 发出申请信号 HRQ (Hold Request), 以便获得总线控制权进行数据传输;
3. CPU 响应 DMA 请求信号, 发出回答信号 HLDA (Hold Acknowledge), 交出总线控制权, 系统转变为 DMA 工作方式;
4. 由 DMAC 控制存储器和 I/O 设备之间的数据交换操作: 首先向存储器发出存储地址信号, 根据传输方向分别向存储器和 I/O 设备发出响应的读写信号传输一个字节的数据, 每传输一个字节, DMAC 的地址寄存器自动加 1, 字节计数器减 1, 直到传输结束;
5. DMA 结束后, 把总线的控制权交还给 CPU。

■ 本章练习

1. 一般的 I/O 接口电路安排有哪几类寄存器? 各自有什么作用?

2. 为什么要对输出接口中的数据进行锁存? 什么情况下可以不锁存?
3. 异步传输时, 每个字符对应 1 个起始位, 7 个有效数据位, 1 个奇偶校验位和一个停止位。若波特率为 1200, 则每秒钟传输的最大字符数为多少?
4. 基本的输入/输出方式有哪几种? 各有什么特点?
5. 利用中断方式协调 CPU 与外设的工作, CPU 应具有哪些特殊能力?
6. DMA 操作过程中, DMA 控制器将接管 CPU 控制系统总线。根据它的这一任务, 请列出 DMA 控制器必须具有的几项功能。

第 6 章

嵌入式系统概述

以往按照计算机的体系结构、运算速度、结构规模、适用领域，将其分为大型计算机、中型机、小型机和微计算机，并以此来组织学科和产业分工，这种分类沿袭了约 *40* 年。随着计算机技术和产品对其它行业的广泛渗透，以应用为中心的分类方法变得更为切合实际，也就是按计算机的嵌入式应用和非嵌入式应用将其分为嵌入式计算机和通用计算机。

通用计算机具有计算机的标准形态。通过装配不同的应用软件，以类同面目出现并应用在社会的各个方面；而嵌入式计算机则是以嵌入式系统的形式隐藏在各种装置、产品和系统中。

6.1 嵌入式系统简介

用于控制设备的计算机 (也就是嵌入式系统 (embedded systems)), 它的历史几乎和计算机自身的历史一样长。它们最初于六十年代晚期在通讯中被用于控制机电电话交换机。根据 IEE[1]的定义, 嵌入式系统最初的定义是 “控制、监视或者辅助设备、机器和车间的运行装置”[2]。此定义尚不能充分体现嵌入式系统的精髓。目前国内一个普遍认同的定义是: 以应用为中心, 以计算机技术为基础, 软件硬件可裁剪, 适应系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。

这样的定义可以从以下几个方面来理解:

- 嵌入式系统是面向用户、面向产品、面向应用的。它必须与具体应用相结合才会具有生命力, 才更有优势。它具有很强的专用性, 必须结合实际需要进行合理的裁剪利用。
- 嵌入式系统是将先进的计算机技术、半导体技术和电子技术以及各个行业的具体应用相结合后的产物。这一点决定了它是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。
- 嵌入式系统必须根据应用需求对软硬件进行剪裁, 满足应用系统的功能、可靠性、成本、体积等要求。所以, 如果能建立相对通用的软硬件基础, 然后在此基础上开发出适应各种需要的系统, 是一个比较好的发展模式。目前嵌入式操作系统的核心往往是一个只有几 KB 到几十 KB 的微内核, 需要根据实际使用进行功能扩展或剪裁。由于微内核的存在, 使得这种扩展能够非常顺利地进行。

同时还应该看到, 嵌入式系统本身还是一个外延极广的概念。凡是与产品结合在一起的具有嵌入式特点的控制系统都可以称为嵌入式系统。随着嵌入式应用的性能完善和提高, 还

[1] Institution of Electrical Engineerings, 英国的一家电子、电气、制造和信息技术的专业组织, 2006 年更名为 Institution of Engineering and Technology。

[2] Devices used to control, monitor or assist the operation of equipment, machinery or plants.

需要有操作系统的支持。现在人们谈到嵌入式系统时, 某种程度上是指具有操作系统的嵌入式系统。

一般而言, 嵌入式系统的构架可以分成四个部分: 处理器, 存储器, 输入/输出 (I/O) 和软件 (图6.1)。

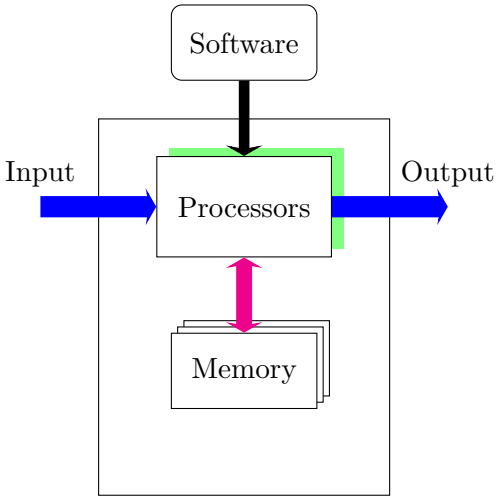


图 6.1: 嵌入式系统的组成

6.2 嵌入式系统的特点

从嵌入式系统的定义可以看出, 嵌入式系统具有以下一些重要特征:

1. 系统内核小。由于嵌入式系统一般应用于小型电子装置, 系统资源有限, 所以内核比传统的操作系统要小得多。
2. 专用性强。嵌入式系统的个性化很强, 软件和硬件结合得非常紧密, 一般要针对硬件进行系统的移植; 即使在同一品牌、同一系列的产品中, 也需要根据系统硬件的变化不断地进行修改。针对不同的任务, 往往需要对系统进行较大地更改。这种修改和通用软件的 “升级” 是完全不同的概念。
3. 系统精简。嵌入式系统一般没有系统软件和应用软件的明显区分, 功能设计和实现不要求过于复杂。这样, 一方面有利于控制系统的成本, 同时也有利于实现系统安全。
4. 高性能的实时操作系统软件。这是嵌入式系统软件的基本要求。同时, 软件要求固化存储, 以提高速度。软件代码要求高质量和高可靠性。
5. 使用多任务的操作系统。嵌入式系统的应用程序可以没有操作系统而直接在芯片上运行。但是为了合理地调度多任务、利用系统资源、系统函数以及 API, 用户必须自行选配 RTOS (**R**eal-**T**ime **O**perating **S**ystem) 开发平台。这样才能保证程序执行的实时性、可靠性, 并缩短开发周期, 保证软件质量。
6. 开发嵌入式系统需要专门的开发工具和开发环境。由于嵌入式系统本身不具备自开发能力。即使设计完成以后, 用户通常也不能对其中的程序功能进行修改, 必须要有一套开发工具和环境才能进行开发。这些工具和环境一般是基于通用计算机上的软硬件设备以及各种逻辑分析仪、信号产生和分析设备等。

6.3 嵌入式系统的分类

嵌入式系统主要由硬件和软件两大部分组成, 所以也可以从硬件和软件进行划分。

6.3.1 嵌入式系统的硬件

各式各样的嵌入式处理器是嵌入式系统硬件中最核心的部分。由于嵌入式系统广阔的发展前景, 很多半导体制造商都开始大规模生产嵌入式处理器, 并且公司自主设计处理器也已成为未来嵌入式领域的一大趋势。从单片机、微处理器到 FPGA, 品种越来越多, 速度越来越快, 性能越来越高, 价格也越来越低。

根据现状, 嵌入式处理器可以分成下面几类: 嵌入式微控制器、嵌入式 DSP 处理器、嵌入式微处理器和嵌入式片上系统。

嵌入式微控制器

嵌入式微控制器又称单片机 (**M**icro **C**ontroller **U**nit, MCU)。顾名思义, 就是将整个计算机系统集成到一块芯片中。嵌入式微控制器一般以某一种微处理器内核为核心, 芯片内部集成了存储器、总线、I/O 接口及各种片内设备, 包括定时器、串行接口、ADC、DAC 等等。为满足不同层次的应用需求, 一般一个系列的单片机具有多种衍生产品, 每种衍生产品的处理器内核都是一样的, 不同的是存储器和外设的配置及封装, 这样可以使单片机最大限度地和应用需求相匹配, 功能不多不少, 从而达到控制成本、降低功耗的目的。

和嵌入式微处理器相比, 微控制器的最大特点是单片化, 体积大大减小, 从而使功耗和成本下降, 可靠性提高。微控制器是目前嵌入式系统工业的主流。此类嵌入式处理器的片上设备资源一般比较丰富, 适合于控制, 因此称微控制器。

目前比较有代表性的通用系列包括 Intel 的 51 系列、TI 的 MSP430 和 C2000 系列、ST 的 STM32 系列等。

嵌入式 DSP

数字信号处理的理论算法在 20 世纪 60 年代就已成形。James Cooley [3]和 John Tukey[4] 发明了快速傅里叶算法, 使这一学科走向成熟。但是由于专门的处理器尚未问世, 大多数理论算法只能通过 MPU 加上分立元件实现, 处理速度无法满足要求。随着大规模集成电路的发展, 1982 年诞生了首枚 DSP (**D**igital **S**ignal **P**rocessor) 芯片, 其运算速度比当时的通用处理器高出一个数量级。随后的发展使 DSP 的存储容量和运算速度都得到了成倍的提高, 应用领域也从开始的语音扩展到图像、通信和计算机等更多方面。

嵌入式 DSP 的代表性产品是德州仪器公司 (Texas Instruments, TI) 的 TMS320 系列、模拟器件公司 (Analog Devices Incoporation, ADI) 的 SHARC 和 Blackfin 系列、Motorola 的 DSP56000 系列等。DSP 是专门用于信号处理方面的处理器, 根据信号处理的需求特点, 它对系统结构和指令进行了特殊设计, 使其适合于执行数字信号处理算法, 编译效率较高, 指令执行速度也较高。

嵌入式 DSP 处理器有两个发展来源: 一是 DSP 处理器经过单片化、电磁兼容性 (**E**lectro-**M**agnetic **C**ompatibility, EMC) 改造、增加片上外设成为嵌入式 DSP 处理器, TI 的 TMS320 系列等属于此范畴; 二是在通用单片机或 SoC 中增加 DSP 协处理器, 此类见于各种多核异构处理器。

[3] James William Cooley (1926 – 2016.6.29), 美国数学家, 1961 年获哥伦比亚大学应用数学博士学位。

[4] John Wilder Tukey (1915. 6. 16 – 2000. 7. 26), 美国数学家, 1939 年获得普林斯顿大学数学博士学位。

推动嵌入式 DSP 处理器发展的另一个因素是嵌入式系统的智能化, 例如各种带有智能逻辑的消费类产品, 生物信息识别终端, 带有加解密算法的键盘, 实时语音、图像处理系统, 虚拟现实显示等。这类智能化算法一般都是运算量较大, 特别是向量运算、指针线性寻址等较多, 而这些正是 DSP 处理器的长处所在。

嵌入式微处理器

嵌入式微处理器 (**M**icro **P**rocessor **U**nit, MPU) 的基础是通用计算机中的 CPU。在应用中, 将微处理器装配在专门设计的电路板上, 只保留和嵌入式应用有关的母板功能, 这样可以大幅度减小系统体积和功耗。为了满足嵌入式应用的特殊要求, 嵌入式微处理器虽然在功能上和标准微处理器基本是一样的, 但在工作温度、电磁干扰、可靠性等方面一般都做了各种增强。

和工业控制计算机相比, 嵌入式微处理器具有体积小、重量轻、成本低、可靠性高的优点。

嵌入式片上系统

随着 EDA (**E**lectronic **D**esign **A**utomation) 的推广和 VLSI 设计的普及化, 以及半导体工艺的迅速发展, 在一个硅片上实现一个更为复杂的系统的时代已来临, 这就是片上系统 (**S**ystem **o**n **C**hip, SoC)。片上系统是追求产品系统最大包容的集成器件, 是目前嵌入式应用领域的热门话题之一。各种通用处理器内核将作为 SoC 设计公司的标准库, 和许多其它嵌入式系统外设一样, 成为 VLSI 设计中一种标准的器件, 用标准的硬件描述语言, 存储在器件库中。用户只需定义出其整个应用系统, 仿真通过后就可以将设计图交给半导体工厂制作样品。这样, 除个别无法集成的器件以外, 整个嵌入式系统大部分均可集成到一块或几块芯片中去, 应用系统电路板将变得很简洁, 对于减小体积和功耗、提高可靠性非常有利。

6.3.2 嵌入式系统的软件

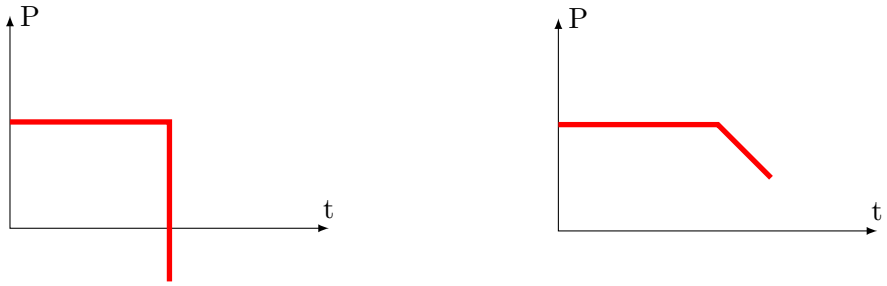
嵌入式系统的软件主要根据操作系统的类型划分。目前嵌入式操作系统的类型主要有两大类: 实时系统 (realtime system) 和分时系统 (time-sharing system)。其中实时系统又分为硬实时系统和软实时系统。

$$\text{嵌入式系统}\begin{cases}\text{实时系统}\begin{cases}\text{硬实时系统}\\ \text{软实时系统}\end{cases}\\ \text{分时系统}\end{cases}$$

实时嵌入式系统是为执行特定功能而设计的, 可以严格地按时序执行功能。其最大特征就是程序的执行具有确定性。实时系统中, 如果在指定的时间未能实现某个确定的任务, 会导致系统的全面失败, 这种系统被称为硬实时系统。软实时系统中, 虽然响应时间同样重要, 但超时不会导致致命的错误。硬实时系统往往需要在硬件上添加专门用于时间和优先级管理的控制芯片, 而软实时系统则主要通过软件编程实现实时管理。

6.4 嵌入式系统的应用领域

嵌入式系统技术具有非常广阔的应用前景, 其应用领域包括以下诸多方面。



(a) 硬实时系统　　(b) 软实时系统

图 6.2: 实时操作系统

1. 工业控制

   基于嵌入式芯片的工业自动化设备具有很大的发展空间。目前已有大量的 8 位、16 位、32 位嵌入式微控制器应用在工业过程控制、数控机床、电力系统、电网安全、电网设备监测、石油化工系统等领域。传统的工业控制产品, 低端的往往采用 8 位单片机。但随着技术的发展, 32 位、64 位的微处理器渐渐成为工业控制设备的核心。
2. 交通管理

   在车辆导航、流量控制、信息监测与汽车服务方面, 嵌入式系统技术已获得了广泛的应用, 运输行业的内嵌 GPS 模块、GSM 模块的移动定位终端已从尖端产品进入寻常领域。
3. 信息家电

   这将是嵌入式系统最大的应用领域。冰箱、空调等的网络化、智能化将引领人们的生活步入一个崭新的空间。即使不在家里, 也可以通过电话、网络进行远程控制。在这些设备中, 嵌入式系统大有用武之地。
4. 智能电器

   水、电、煤气表的远程自动抄表, 安全防火、防盗系统, 其中嵌有的专用控制芯片代替传统的人工方法, 实现更高效和更安全的性能。
5. POS 网络及电子商务

   公共交通的无接触智能卡发行系统、公共电话卡发行系统、自动售货机、各种智能 ATM 终端将全面走进人们的生活。
6. 环境监测

   环境监测包括水文资料实时监测、防洪体系及水土质量监测、堤坝安全、地震监测网、实时气象信息、水源和空气污染监测。在很多环境恶劣、地况复杂的地区, 嵌入式系统将实现无人监测。
7. 机器人

   嵌入式芯片的发展将使机器人在微型化、高智能方面的优势更加明显, 同时会大幅度降低机器人的价格, 使其在工业领域和其它行业获得更广泛的应用。

除以上领域, 嵌入式系统还有其它方面的应用, 在现代社会生活中无处不在。

6.5 嵌入式系统的现状和发展趋势

随着信息化、智能化、网络化的发展, 嵌入式系统技术也将获得广阔的发展空间。目前, 嵌入式技术已成为通信和消费类产品的共同发展方向。在通信领域, 数字技术正在全

面取代模拟技术。软件、集成电路和新型元器件在产业发展中的作用日益重要, 这些都离不开嵌入式系统技术。个人移动数据处理和通信设备也广泛采用了嵌入式技术。由于嵌入式设备具有自然的人–机交互界面, GUI 多媒体界面给人以很大的和移动通信设备、GPS、娱乐相结合, 嵌入式系统同样可以发挥巨大的作用。

硬件方面, 不仅有各大公司的微处理器芯片, 还有用于学习和研发的各种配套开发包。低层系统和硬件平台经过数年的研究, 已相对比较成熟, 实现各种功能的芯片应有尽有。相当多的软件系统 (包括商业的和免费的) 为嵌入式系统开发提供了良好的条件。

未来嵌入式系统的发展趋势体现在下面这些方面:

1. 嵌入式开发是一项系统工程, 要求嵌入式厂商不仅要提供嵌入式硬件系统本身, 还要提供强大的硬件开发工具和软件支持。
2. 网络化、信息化的要求日益提高, 使得以往单一功能的设备如手机、微波炉等的功能不再单一, 结构更加复杂。
3. 为了适应网络发展的要求, 必然要求未来的嵌入式设备提供各种网络通信接口。新一代的嵌入式处理器已开始内嵌网络接口, 除了支持 TCP/IP 协议, 有的还支持 IEEE–1394、USB、CAN、蓝牙等等, 同时也提供相应的协议软件和物理层驱动软件。软件方面, 系统内核支持网络模块, 甚至可以嵌入 Web 浏览器。
4. 精简系统内核、算法, 降低功耗和软硬件成本, 利用最低的资源实现最适当的功能, 这要求开发人员既要有丰富的硬件知识, 又需要开发先进的嵌入式软件技术。
5. 提供友好的多媒体人–机界面。这方面的要求使得嵌入式软件设计者要在图形界面和多媒体技术上多下功夫。这是让嵌入式设备能与用户亲密接触的重要因素。

6.6 嵌入式操作系统

实时嵌入式操作系统种类繁多, 大体上可以分为两种: 商用型和免费型。商用实时操作系统功能稳定可靠, 有完善的技术支持和售后服务。免费型的实时操作系统在价格方面具有优势。

从 20 世纪 80 年代, 国际一些 IT 组织和公司就开始了商用嵌入式操作系统和专用操作系统的研发。经过多年的发展, 目前世界上已有一大批成熟的实时嵌入式操作系统。

6.6.1 商用实时嵌入式操作系统

- VxWorks

  1983 年由美国 WindRiver 公司开发的实时嵌入式操作系统。因其高性能的系统内核和友好的用户开发环境, 成为目前使用最广、市场占有率最高的实时嵌入式操作系统。其突出特点是可靠性、实时性和可裁减性。它支持多种处理器, 如 X86、i960、Sun Sparc、MC68k、MIPS RX、Power PC 等。

- μC/OS-II

  μC/OS-II 是一个可移植、可固化的、可裁剪的、占先式多任务实时内核，适用于多种微处理器，微控制器和数字处理芯片，已经移植到超过 100 种以上的微处理器应用中。同时，该系统源代码开放、整洁、一致，注释详尽，适合系统开发。μC/OS-II 绝大部分源码是用移植性很强的 ANSI C 写的。用汇编语言写的与处理器硬件相关的部分已压到最低限度, 使得它将移植的工作量减到最小: 只要求该处理器有堆栈指针, 有 CPU 寄存器入栈、出栈指令。μC/OS-II 可以在绝大多数 8 位、16 位、32 位甚至 64 位微处理器、微控制器、DSP 上运行。全部 μC/OS-II 的函数调用与服务的执行时间具有确定性, 也

就是说, 全部 μC/OS-II 的函数调用与服务的执行时间是可知的, 它不依赖于应用程序任务的多少。

- LynxOS

  LynxOS 是一个分布式、可扩展规模的实时嵌入式操作系统。LynxOS 支持线程概念, 提供 256 个全局用户线程优先级, 提供一些传统的、非实时系统的服务特征, 包括基于调用需求的虚拟内存、基于 Motif 的图形用户界面、与工业标准兼容的网络系统以及应用开发工具。

- QNX

  QNX 是一个实时的、可扩充的类 UNIX 操作系统。它部分遵循 POSIX 相关标准, 提供了一个很小的微内核以及一些可选的配合进程。内核仅提供四种服务: 进程调度、进程间通信、底层网络通信和中断处理。

  由于 QNX 具有强大的图形界面功能, 因此很适合作为机顶盒、手持设备、GPS 设备的实时操作系统。

6.6.2 开放源码的操作系统

- 嵌入式 Linux

  自由免费软件 Linux 的出现是对目前商用嵌入式操作系统的冲击。作为嵌入式操作系统, Linux 有极其诱人的优势。它可以移植到多个有不同架构的 CPU 和硬件平台上, 具有很好的稳定性, 良好的升级性能, 并且开发更容易。

  由于嵌入式系统越来越追求数字化、网络化和智能化, 原来在某些设备领域中占主导地位的软件系统便难以维继。因为要达到上述要求, 整个系统必须是开放的, 提供标准的 API, 并且能够方便地与众多第三方软件沟通。

  在这些方面, Linux 有得天独厚的优势。首先, Linux 开放源码, 不存在黑箱技术; 遍布全球的众多 Linux 爱好者是 Linux 开发的强大技术后盾; 其次, Linux 内核小、功能强大、运行稳定、效率高; 第三, Linux 是一种开放源码的操作系统, 易于定制剪裁, 在价格上极具竞争力; 第四, Linux 不仅支持 X86 CPU, 还可以支持其它数十种处理器架构和数百种处理器芯片; 第五, 有大量的且不断增加的开发工具, 这些工具为嵌入式系统的开发提供了良好的开发环境; 第六, Linux 沿用了 UNIX 的发展方式, 遵循国际标准, 可以方便地获得众多第三方软件的支持; 最后, Linux 内核的结构在网络方面非常完整, 它提供了对十兆、百兆、千兆以太网、无线网络、令牌网、光纤网等多种联网方式的全面支持。此外, 在图像处理、文件管理及多任务支持等诸多方面, Linux 的表现也都非常出色。因此它不仅可以充当嵌入式系统的开发平台, 其本身也是嵌入式系统应用开发的良好工具。

  针对实时操作系统的要求, RT-Linux 和 Real-time Linux 也受到众多嵌入式应用系统的重视。

- FreeRTOS

  FreeRTOS 是一个广受欢迎的微内核实时操作系统，以 MIT 版权协议发布。

  FreeRTOS 着重在运行的简洁与速度，占用存储器资源极低，支持 Arm、X86、PPC、MSP430 等数十种处理器架构。它既是一个适合操作系统学习的入门教材，也是专业开发人员用于商业产品开发的极具竞争力的选项。

- RTEMS

  RTEMS (**R**eal-**T**ime **E**xecutive for **M**ultiprocessor **S**ystems) 是一个为嵌入式系统而设计的自由的开源实时操作系统。该项目起于 1980 年代晚期，最初被美国国防系统用于

军事目的。其中的字母 “M” 由导弹 (Missile) 到军事 (Military) 演化到现在的 “多处理器” 的概念。它支持包括 POSIX 在内的多种开放 API 标准，支持 Arm、i386、M68K、MIPS、PowerPC、SH 等多种不同的处理器, 移植了包括 NFS 和 FATFS 的多种文件系统和 FreeBSD TCP/IP 协议栈，提供一个单进程、多线程的任务环境。RTEMS 以 GPL 版权协议发布。

- Fuchsia

  Fuchsia (中文译名 “灯笼海棠”) 是 Google 基于微内核 Zircon 开发的面向嵌入式应用的操作系统，主要使用 C 语言和 C++ 编写。Fuchsia 的设计目标之一是可运行在众多的设备上，包括移动电话、平板和个人计算机。Fuchsia 的用户界面和应用软件使用跨平台开发工具 Flutter 开发。

  Fuchsia OS 的开发者网站 Fuchsia.dev 上线，但到目前为止，尚无正式的商业产品上市。

6.7 嵌入式系统的选型原则

6.7.1 硬件平台的选择

嵌入式硬件平台的选择主要是指嵌入式处理器的选择。使用什么样的嵌入式处理器内核主要取决于应用领域、用户需求、成本、开发的难易程度等因素。

挑选最佳硬件的过程会相当复杂, 需要从多方面考虑总的产品成本 (企业策略、信息渠道、货源稳定性等等), 而不仅仅是 CPU 本身。有时, 一旦把 CPU 使用其它外围设备所必需的总线逻辑和延迟时间考虑在内, 快速而廉价的 CPU 也可能变得昂贵。

确定了使用哪种嵌入式处理器内核以后, 接下来就是综合考虑系统外围设备的需求情况以选择一款合适的处理器。下面列出考虑系统外围设备的一些因素:

- 总线的需求, 通用并行总线 PCI、串行通信总线 USB、I2C、SPI 等等;
- 通用串行接口;
- 以太网和无线通信接口;
- ADC 或 DAC 及音频 IIS 总线;
- 外设接口及 I/O 控制接口。

另外, 还要考虑处理器的寻址空间, 是否需要片上 FLASH, 处理器是否容易调试和仿真以及调试工具的成本和易用性等相关的信息。在实际过程中, 挑选硬件是一项很复杂的工作, 充满着各种顾忌和干扰, 包括其它工程的影响以及缺乏完整或准确的信息等。

6.7.2 嵌入式操作系统的选择

实时嵌入式系统的种类繁多, 大体上可分为商用型和免费型两种。商用型的实时操作系统功能稳定、可靠, 有完善的技术支持和售后服务, 但往往价格昂贵; 免费型的实时操作系统在价格方面具有优势。不管选用什么样的系统, 都要考虑以下几点:

- 能否满足应用需求
- 操作系统的硬件支持
- 开发工具的支持程度

由此可见, 选择一款既能满足应用需求、性价比又可达到最佳的实时操作系统, 对开发工作的顺利开展意义重大。

6.8 嵌入式系统的实时性

嵌入式系统的硬件核心是嵌入式处理器。目前的嵌入式应用中还广泛采用了操作系统支持。根据操作系统的工作特性, 有实时操作系统和分时操作系统之分。其中实时操作系统是目前嵌入式系统最主要的组成部分。

嵌入式实时系统中采用的操作系统称为嵌入式实时操作系统, 它既是嵌入式操作系统, 又是实时操作系统。作为一种嵌入式操作系统, 它具有嵌入式软件共有的可裁剪、低资源占用、低功耗等特点; 而作为一种实时操作系统, 它与通用操作系统 (如 Windows、Linux 等) 相比也有一些重要的差别。

实时是指物理进程的真实时间。实时操作系统具有实时性, 能从硬件方面支持实时, 控制系统工作的操作系统, 其中的实时性是第一要求, 需要调度一切可利用的资源完成实时控制任务。其次才着眼于提高计算机系统的使用效率, 重要特点是满足对时间的限制和要求。

通用操作系统体现的是分时操作系统的特性, 它们大部分都支持多用户和多进程, 负责管理众多的进程并为它们分配系统资源。软件的执行在时间上的要求并不严格, 而是侧重于系统整体性能。时间上的延误, 一般不会造成灾难性的后果。分时操作系统的基本设计原则是: 尽量缩短系统的平均响应时间并提高系统的吞吐率, 在单位时间内为尽可能多的用户请求提供服务。从系统的角度看, 分时操作系统注重系统的整体表现性能, 而不关心单个具体任务的响应时间; 对于某个任务来说, 注重每次执行的平均响应时间而不关心某次特定执行的响应时间。通用操作系统中采用的很多策略和技巧都体现出了这种设计原则, 如虚存管理机制中, 由于采用 LRU 等页替换算法, 使得大部分的访存需求能够快速地通过物理内存完成, 只有很小一部分的访存需求需要通过调页完成, 但从总体上来看, 平均访存时间与不采用虚拟存储技术相比没有很大的提高, 同时又获得了虚拟空间可以远大于物理内存容量等好处, 因此虚拟存储技术在通用操作系统中得到了十分广泛的应用。

而对于实时操作系统, 它除了要满足应用的功能需求以外, 更重要的是还要满足应用提出的实时性要求。不同的应用、同一系统中不同的实时任务对于实时性的要求是各不相同的, 此外实时任务之间可能还存在一些复杂的关联和同步关系, 如执行顺序限制、共享资源的互斥访问要求等, 这就为系统的实时性设计带来了很大的困难。因此, 实时操作系统所遵循的最重要的设计原则是: 采用各种算法和策略, 始终保证系统行为的可预测性。可预测性是指在系统运行的任何时刻, 在任何情况下, 实时操作系统的资源调配策略都能为共享资源 (包括 CPU、内存、外设等) 的多个实时任务合理地分配资源, 使每个实时任务的实时性要求都能得到满足。此时操作系统注重的不再是系统的平均表现, 而是要求每个实时任务在最坏情况下都要满足其实时性要求。

6.8.1 实时操作系统的特点

由于实时操作系统与通用操作系统的基本设计原则差别很大, 因此在很多资源调度策略的选择上以及操作系统实现的方法上两者都具有较大的差异, 这些差异主要体现在以下几点:

1. 任务调度策略

   通用操作系统中的任务调度策略一般采用基于优先级的抢先式调度策略, 对于优先级相同的进程则采用时间片轮转调度方式, 用户进程可以通过系统调用动态地调整自己的优先级, 操作系统可根据情况调整某些进程的优先级。

   实时操作系统中的任务调度策略目前使用最广泛的主要可分为两种, 一种是静态表驱动方式, 另一种是固定优先级抢先式调度方式。

静态表驱动方式是指在系统运行前工程师根据各任务的实时要求用手工的方式或在辅助工具的帮助下生成一张任务的运行时间表, 这张时间表与列车的运行时刻表类似, 指明了各任务的起始运行时间以及运行长度。运行时间表一旦生成就不再变化了, 在运行时调度器只需根据这张表在指定的时刻启动相应的任务管理即可。静态表驱动方式的主要优点是:

- 运行时间表是在系统运行前生成的, 因此可以采用较复杂的搜索算法找到较优的调度方案;
- 运行时调度器开销较小;
- 系统具有非常好的可预测性, 实时性验证也比较方便。

这种方式主要用于航空航天、军事等对系统的实时性要求十分严格的领域。其主要缺点是不灵活。需求一旦发生变化, 就要重新生成整个运行时量间表。

固定优先级抢先式调度方式则与通用操作系统中采用的基于优先级的调度方式基本类似, 但在固定优先级抢先式调度方式中, 进程的优先级是固定不变的, 并且该优先级是在运行前通过某种优先级分配策略来指定的。这种方式的优缺点与静态表驱动方式的优缺点正好完全相反, 它主要应用于一些较简单、较独立的嵌入式系统, 但随着调度理论的不断成熟和完善, 这种方式也会逐渐在一些对实时性要求十分严格的领域中得到应用。目前市场上大部分的实时操作系统采用的都是这种调度方式。

2. 内存管理

关于虚存管理机制在上面已经进行了一些分析。为解决虚存给系统带来的不可预测性, 实时操作系统一般采用如下两种方式:

- 在原有虚存管理机制的基础上增加页面锁功能, 用户可将关键页面锁定在内存中, 从而不会被 swap 程序将该页面交换出内存。这种方式的优点是既得到了虚拟存储管理机制为软件开发带来的好处, 又提高了系统的可预测性。缺点是由于 TLB 等机制的也是按照注重平均表现的原则进行的, 因此系统的可预测性并不能完全得到保障;
- 采用静态内存划分的方式, 为每个实时任务划分固定的内存区域。这种方式的优点是系统具有较好的可预测性, 缺点是灵活性不够好, 任务对存储器的需求一旦有变化就需要重新对内存进行划分, 此外虚拟存储管理机制所带来的好处也丧失了。

目前市场上的实时操作系统一般都采用第一种管理方式。

3. 高速缓存

高速缓存 (cache) 的主要作用是用少量的高速存储器件来弥补高性能 CPU 与相对来说性能较低的存储器之间的性能差异, 从而提高存储设备的性价比。由于它可以使系统的平均表现性能得到大幅提高, 因此在硬件设计中得到了极为广泛的应用。

但实时操作系统注重的不是平均表现性能, 而是个体最坏情况表现, 因此在对系统进行实时性验证时必须考虑实时任务运行的最坏情况, 即每次访问内存都没有命中 cache 情况下的运行时间, 所以在利用辅助工具估算实时任务在最坏情况下的执行时间, 应将系统中所有的 cache 功能暂时关闭, 在系统实际运行时再将 cache 功能激活。除此以外, 另一种较极端的做法则是在硬件设计中完全不采用 cache 技术。

4. 中断处理

在通用操作系统中, 大部分外部中断都是开启的, 中断处理一般由设备驱动程序来完成。由于通用操作系统中的用户进程一般都没有实时性要求, 而中断处理程序直接跟硬件设备交互, 可能有实时性要求, 因此中断处理程序的优先级被设定为高于任何用户进程。

但对于实时操作系统, 采用上述的中断处理机制则是不合适的。首先, 外部中断是环境向实时操作系统进行的输入, 它的频度是与环境变化的速率相关的, 而与实时操作系统无关。如果外部中断产生的频度不可预测, 则一个实时任务在运行时被中断处理程序阻塞的时间开销也是不可预测的, 从而使任务的实时性得不到保证; 如果外部中断产生的频度是可预测的, 一旦某外部中断产生的频度超出其预测值 (如硬件故障产生的虚假中断信号或预测值本身有误) 就可能会破坏整个系统的可预测性。其次, 实时控制系统中的各用户进程一般都有实时性要求, 因此中断处理程序优先级高于所有用户进程的优先级分配方式是不合适的。

一种较适合实时操作系统的中断处理方式为: 除时钟中断外, 屏蔽所有其它中断, 中断处理程序变为周期性的轮询操作。采用这种方式的主要好处是充分保证了系统的可预测性, 主要缺点是对环境变化的响应可能不如上述中断处理方式快。另外, 轮询操作在一定程度上降低了 CPU 的有效利用率。另一种可行的方式是: 对于采用轮询方式无法满足需求的外部事件, 采用中断方式, 其它时间仍然采用轮询方式。但此时中断处理程序与所以其它任务一样拥有优先级, 调度器根据优先级对处于就绪态的任务和中断处理程序统一进行处理器调度。这种方式使外部事件的响应速度加快, 避免了上述中断方式带来的第二个问题, 但第一个问题依然存在。

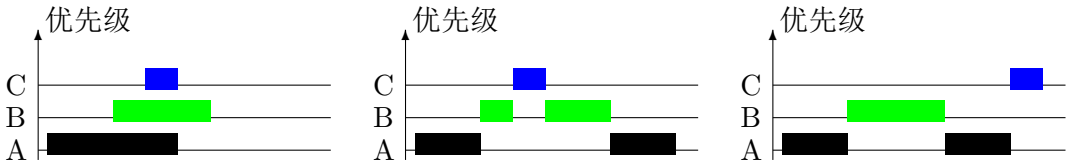
此外为提高时钟中断响应时间的可预测性, 实时操作系统应尽可能少地屏蔽中断。

5. 共享资源的互斥访问

通用操作系统一般采用信号量机制来解决共享资源的互斥访问问题。

对于实时操作系统, 如果任务调度采用静态表驱动方式, 共享资源的互斥访问问题在生成运行时间表时已经考虑到了, 在运行时无需再考虑。如果任务调度采用基于优先级的方式, 则传统的信号量机制在系统运行时很容易造成优先级倒置问题, 即当一个高优先级任务通过信号量机制访问共享资源时, 该信号量已被一低优先级任务占有, 而这个低优先级任务在访问共享资源时可能又被其它一些中等优先级的任务抢先, 因此造成高优先级任务被许多具有较低优先级的任务阻塞, 实时性难以得到保证。

例如我们考虑图 6.3 中三个任务的调度完成情况。



(a) 任务顺序 (b) 理想情况 (c) 实际完成情况

图 6.3: 多任务调度的优先级倒置

在没有共享资源冲突的情况下, 系统根据优先级策略, 按预期方案完成了每个任务。但如果发生共享资源冲突, 比如任务 A 和任务 C 需要使用共同的资源, 但由于任务 A 尚未结束, 导致高优先级的任务 C 无法开始, 而优先级高于 A、低于 C 的任务 B 因为没有资源冲突问题, 得到优先运行权, 从而发生 B、C 优先级倒置 (priority inversion)。

解决优先级倒置的问题可以采用以下两种方案:

优先级继承 (priority inheritance) 协议 当低优先级的任务获取共享资源控制权时, 系统按原定的优先级方案运行; 如此时有高优先级的任务申请该资源, 则将

低优先级的任务的优先级抬高至高优先级, 使正在运行的任务全速运行, 尽早释放共享资源。

优先级顶置 (priority ceiling) 协议 为共享资源设置一个顶置优先级, 该优先级高于所有使用此共享资源任务的优先级。一旦某任务和共享资源发生关系, 则不允许其它任务抢占。

6. 系统调用以及系统内部操作的时间开销

   进程通过系统调用得到操作系统提供的服务, 操作系统通过内部操作 (加上下文切换等) 来完成一些内部管理工作。为保证系统的可预测性, 实时操作系统中的所有系统调用以及系统内部操作的时间开销都应是有界的, 并且该界限是一个具体的量化数值。而在通用操作系统中对这些时间开销则未做如此限制。

7. 系统的可重入性

   在通用操作系统中, 核心态系统调用往往是不可重入的, 当一低优先级任务调用核心态系统调用时, 在该时间段内到达的高优先级任务必须等到低优先级的系统调用完成才能获得 CPU, 这就降低了系统的可预测性。

   考虑如下两个程序片:

```
int temp;
swap(int *a, int *b)
{
    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

   两段程序唯一不同点就在于一个使用了全局变量 `temp`, 而另一个使用的是局部变量。编译器将全局变量分配在公共数据区, 而将局部变量分配在堆栈区。在多任务环境下, 如果发生不同任务对函数 `swap()` 的重复调用, 则全局变量 `temp` 的值就会受到影响, 即不可重入性。实时操作系统中的核心态系统调用往往设计为可重入的。

8. DMA

   DMA 是一种数据交换协议, 主要作用是在无需 CPU 参与的情况下将数据在内存与其它外部设备间进行交换。

   DMA 最常用的一种实现方式被称为周期窃取 (Cycle Stealing) 方式, 即首先通过总线仲裁协议与 CPU 竞争总线控制权, 在获得控制权后再根据用户预设的操作指令进行数据交换。由于这种周期窃取方式会给用户任务带来不可预测的额外阻塞开销, 所以实时操作系统往往要求系统设计时不采用 DMA, 或采取一些可预测性更好的 DMA 实现方式。

6.8.2 实时系统中的其它概念

- 多任务 (Multi-Task)。在嵌入式应用领域, 多任务是一个普遍的要求。多任务运行的实现, 实际上是依靠 CPU 为每个任务分配相应的时间片、通过调度机制轮番服务于一系列任务中的某一个而实现的。多任务运行很像前后台系统, 但后台任务有多个。多任务运行使 CPU 的利用率得到最大的发挥, 并使应用程序模块化。在实时应用中, 多任务的

最大特点是, 允许开发人员将复杂的应用程序层次化。多任务使得应用程序的设计和维护变得更加容易。

- 任务的优先级 (Priority)。每个任务都有其优先级, 重要的任务应赋予更高的优先级。应用程序执行过程中, 诸任务的优先级不变, 称为静态优先级。在静态优先级系统中, 诸任务以及它们的时间约束在程序编译时是已知的。

  如果任务的优先级在应用程序的执行过程中是可变的, 则称为动态优先级。
- 内核 (Kernel)。多任务系统中, 内核负责管理各个任务, 为每个任务分配 CPU 的时间, 并负责任务之间的通信。内核提供的基本服务是任务切换。实时内核可以大大地简化应用系统的设计, 因为实时内核可以将应用分成若干个任务并进行有效的管理。
- 调度 (Scheduler/Dispatcher)。调度是内核的主要职责之一。多数实时内核是基于优先级的调度法则。CPU 总是让处于就绪状态的优先级最高的任务先运行。究竟何时让高优先级的任务获得 CPU 使用权, 有两种不同的情况。要看内核的类型是属于非先占式的还是先占式的。
- 系统响应时间 (System Response Time)。系统发出处理要求到系统给出应答的时间。
- 任务切换时间 (Context-Switch Time)。任务之间切换而使用的时间。
- 中断延迟 (Interrupt Latency)。计算机接收到中断信号到操作系统做出响应、并完成切换, 转入中断服务程序的时间。

6.9 Arm 处理器

Arm (**A**dvanced **R**ISC **M**achine) 处理器是目前世界上应用最为广泛的嵌入式内核。Arm 公司是著名的知识产权供应商, 本身不生产芯片, 靠转让设计许可、由合作伙伴公司来生产各具特色的芯片。Arm 公司在全世界有上百个合作伙伴, 其中包括半导体工业的著名公司, 从而产生了大量的开发工具和丰富的第三方资源, 它们共同保证了基于 Arm 处理器内核的设计可以迅速投入市场。

最初的 Arm 处理器是 32 位设计。大多数 Arm 处理器支持超过一种指令集: 32 位指令集 Arm 和 16 位指令集 Thumb。2012 年发布了 64 位指令集架构 Aarch64。Thumb 指令集允许软件以 16 位编码，具有更好的代码密度 (相比 32 位的 Arm 代码，性能有所降低)。

Arm 内核包括经典的 Arm 系列 (Arm7~Arm11、StrongArm、XScale 等) 和 Cortex 系列。目前 Cortex 系列包含以下三个子系列:

- Cortex-A: 高性能应用型处理器 (Application)。
- Cortex-R: 实时应用型处理器 (Real-time)。
- Cortex-M: 面向嵌入式应用的微控制器 (Microcontroller)。

6.9.1 工作模式

Arm 架构具有一组寄存器, 提供处理器内部的数据存储, 它们是 R0–R12 通用寄存器、作为程序计数器使用的 R15、链接寄存器 R14 (用于存放函数或中断的返回地址)、堆栈指针 R13, 此外还有包含 ALU 标志位与其它执行状态信息的程序状态寄存器等等。这些寄存器中很多是分组的, 除非是在特定处理器模式, 否则处理器不能访问。Arm 架构提供 9 种处理器模式, 每个处理器模式都有不同的寄存器组。寄存器分组在处理器异常和特权操作时可以得到快速的上下文切换。处理器模式切换可通过指令向 CPSR (**C**urrent **P**rogramm **S**tatus **R**egisger) 寄存器模式位写入相应的编码, 也可能是因发生异常事件而自动进入。

表 6.1: Arm 处理器模式

模式	编码	功能
USR	10000	用户模式，运行大多数应用程序
FIQ	10001	快速中断模式
IRQ	10010	中断模式
SVC	10011	复位或执行超级调用指令
MON	10110	监控模式，安全扩展
ABT	10111	存储器访问异常
HYP	11010	虚拟扩展
UND	11011	执行未定义指令时进入此模式
SYS	11111	特权模式

6.9.2 Arm 指令特色

Arm 指令码的最高 4 位是条件码，每条指令都是可以成为有条件执行的指令, 这可以简化程序分支结构。例如一个标准的求解最大公约数的欧几里得算法:

```
int gcd (int i, int j)
{
    while (i != j) {
        if (i > j)
            i -= j;
        else
            j -= i;
    }
    return i;
}
```

采用 Arm 指令集编译的结果甚至比 C 语言还简短：

```
loop:   CMP     R3, R4          ; 比较R3和R4的大小
        SUBGT   R3, R3, R4      ; 若R3大，则R3=R3-R4
        SUBLT   R4, R4, R3      ; 若R3小，则R4=R4-R3
        BNE     loop            ; 若相减不为0，则继续循环
```

Arm 指令的另一个别具特色的地方是可以将移位/旋转操作和数据运算合并在一条指令中，这使得像

```
   x += (y << 3);
```

这样的指令只需要一个周期即可完成，可以让程序更加紧凑，减少对存储器的访问。

表 6.2: Arm 寄存器分组

USR	SYS	SVC	ABT	UND	IRQ	FIQ
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_FIQ
R9						R9_FIQ
R10						R10_FIQ
R11						R11_FIQ
R12						R12_FIQ
R13	R13_SVC	R13_ABT	R13_UND	R13_IRQ	R13_FIQ	
R14	R14_SVC	R14_ABT	R14_UND	R14_IRQ	R14_FIQ	
R15						
CPSR						
	SPSR_SVC	SPSR_ABT	SPSR_UND	SPSR_IRQ	SPSR_FIQ	

■ 本章练习

索引