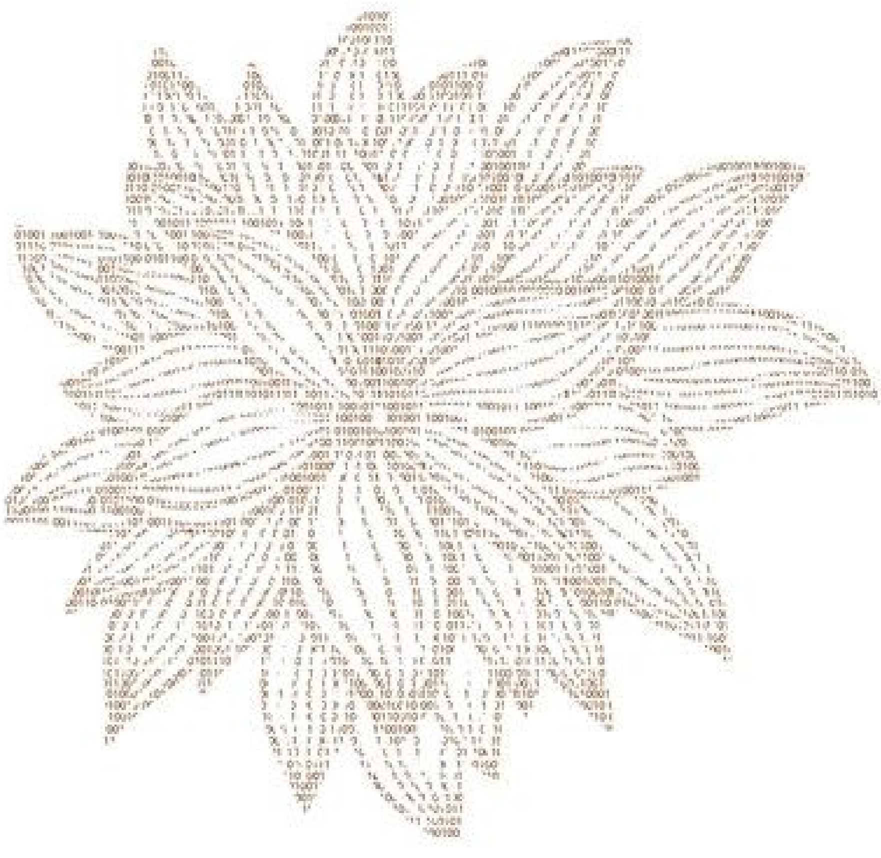
图灵程序设计丛书

Unix内核源码剖析

超级计算机"京"的L1缓存设计者【日】青柳隆宏 著
殷中翔 译 黄炎 周金杰 审



进程 中断 块I/O系统 文件系统 字符I/O系统 启动系统

精读1万行代码，深入理解操作系统原理

人民邮电出版社
POSTS & TELECOM PRESS

3、 25岁前一定要读的25本书

4、 有生之年，你一定要看的25部外国纯文学名著

5、 有生之年，你一定要看的20部中国现当代名著

6、 美国亚马逊编辑推荐的一生必读书单100本

7、 30个领域30本不容错过的入门书

8、 这20本书，是各领域的巅峰之作

9、 这7本书，教你如何高效读书

10、 80万书虫力荐的“给五星都不够”的30本书

版权信息

书名：Unix内核源码剖析

作者：（日） 青柳隆宏

译者：殷中翔

ISBN：978-7-115-34521-9

本书由北京图灵文化发展有限公司发行数字版。

译者序

本书作为学习 UNIX V6 内核源码的参考读物，值得向读者推荐。

国内计算机方面的翻译书籍以欧美作者的著作为主，这与相关的技术多起源于欧美有很大关系。在实际的学习和工作中，我也接触了许多日文书籍，我认为此领域的日文书籍也很有特色。因为日文书籍很擅长从读者的角度出发，配合大量的实例与图表，深入浅出地将一个复杂的问题讲解清楚，非常适合初学者或者工程技术人员阅读。

正如作者在前言中写到的，这本书是基于作者在学习 UNIX V6 内核源码过程中的笔记和博客而写成的，因此对初次接触 UNIX V6 内核的读者而言，无疑是一本很好的参考读物。

但是也应该注意到，正是因为这一点，一些作者已掌握的知识点，书中并未详细阐述。因此，将本书定位于参考书，同时配合附录中的其他参考资料，再借助模拟器 simh 实际动手操作，这样三管齐下才能真正达到理解 UNIX V6 内核源码的目的。

我一直很想将优秀的计算机日文书籍介绍给国内的广大读者。因为工作繁忙，迟迟没有付诸实际行动，直到今年才在图灵公司的大力支持下实现。在这里特别向本书的编辑表示衷心的感谢，他们从译文的逻辑表达和语法等方面提出了宝贵的意见。

同时，也要向原著作者青柳隆宏先生致敬。他的语言表达非常严谨，条理十分鲜明。这使得原著在日本亚马逊网站中获得了 4 星的高分。

最后，我要向家人表示谢意。这本书基本都是利用平时下班后和周末的时间来完成的，没有父母、妻子及两个可爱的孩子的支持，我很难顺利、按时完成翻译工作。

本书的翻译肯定会有许多不足，欢迎广大读者提出宝贵意见或来信交流。我的邮箱地址是 unixv6.yinzx@gmail.com，我会尽可能给与答复。

如果本书能给大家提供一些帮助的话，我会感到十分欣慰。

殷中翔

2013 年 10 月于福冈

前言

本书针对 1975 年由贝尔实验室[1]发布的 UNIX 第 6 版（Sixth Edition Unix，此后简称为 UNIX V6 ）的内核源代码进行解说。面向的读者主要是计算机专业的学生，以及从事计算机相关行业的具有初中级水平的技术人员。

> [1] 由美国的 AT&T 公司和 Western Electric 公司于 1925 年设立的研究开发机构。

考虑到一部分读者会有诸如“我对内核源代码根本不感兴趣”或者“与这种老古董相 比，我喜欢更现代的操作系统”等看法，笔者想先阐述一下阅读内核源代码的引人入胜之处**,**然后再解释 **UNIX V6** 为何适合初次接触内核源代码的读者 。

阅读内核源代码的意义

我们可以将操作系统（OS，Operating System）看做是一种软件（集合），它对包括硬件和软件在内的计算机系统的各个部分进行管理，并为用户提供便于使用的操作界面。内核作为操作系统的核心部分**,**提供计算机系统必备的功能，因此也被称为狭义的操作系统。例如，shell 之类的程序通常不是内核的一部分，而是利用内核提供的功能来实现的。内核以外的程序通常被称为用户空间（userland）程序，或用户程序。

通过阅读并理解内核源代码，我们会有如下收获。

对计算机系统的全貌有更深入的了解

掌握了作为计算机系统核心部分的内核，不仅对操作系统，对计算机的全貌也会有更为深入的认识。对通过大学课程或其他途径学习的各种领域、各个层面的知识之间的关联性也会有更清晰的认识**,**让人有醍醐灌顶的感觉。

让操作计算机成为一种令人愉快的体验

理解了计算机系统的全貌，操作计算机本身也会变得更加令人愉快。比如，在计算机上执行某个程序的时候，如果能够准确把握系统内部所进行的操作，是不是一件很令人兴奋的事情呢？这种体验将加深读者对计算机的兴趣**,**使读者更有动力去提高自己的技术水平 。

加深对知识的理解

阅读代码与否，对知识的理解程度会有云泥之差。如果只学习了概要，既容易遗忘也难以应用。相反，理解代码能够使你对学到的算法和思路举一反三**,**使之成为可以受用一生的财富 。

提升技术人员自身的水平

作为计算机行业的技术人员，阅读并理解了内核源代码有助于在专业领域里将自己提升到一个新的层次。尽管在全球范围内这个领域的从业者不断增加，但是在了解应用层面的同时，对操作系统等底层的知识也有所了解**,**并且能够对系统做出整体优化的技术人员**,**仍是凤毛麟角 。

但是恰恰是具备这种素质的人，才能在第一线发挥不可替代的作用。如果想拉大与竞争对手的差距**,**是必须理解系统内核的 。

为何选择 **UNIX V6**

接下来想说明一下为什么不选择最新的操作系统，而将历史比较悠久的 UNIX V6 作为本书的题材。

代码行数约为 **1** 万行

UNIX V6 的内核源代码包括设备驱动程序在内约有 **1** 万行 ，这个数量的源代码，初学者是能够充分理解的。有一种说法是一个人所能理解的代码量上限为 1 万行，UNIX V6 的内核源代码从数量上看正好在这个

范围之内。看到这里，大家是不是也有“如果只有 1 万行的话没准儿我也能学会”的想法呢？

另一方面，最近的操作系统，例如 Linux 最新版的内核源代码据说超过了 1000 万行 2 。就算不是初学者，想完全理解全部代码基本上也是不可能的。

2 http://www.h-online.com/open/features/Kernel-Log-15-000-000-lines-of-code-3-0-promoted-to-long-term-kernel-1408062.html 。

充实的资料

UNIX V6 的用户手册、相关资料和论文都可以在网上找到 。运行 UNIX V6 所需的处理装置 PDP-11 以及周边设备的设计文档，很大一部分也可以检索到。

另外，有一本关于 **UNIX V6** 的指南已经问世多年 。此书名为《莱昂氏 UNIX 源代码分析》（*Lions' Commentary on UNIX*，机械工业出版社出版），由澳大利亚新南威尔士大学的约翰·莱昂（John Lions）教授撰写，也被称为 Lions 书（Lions Book）[3] 3 。这本书由于 UNIX 的版权问题，据说当时只能在学生之间私下传看，后来才广泛地流传开来，现在可以从网上免费下载 [4] 。Lions 书对本书中基本没有涉及的 PDP-11/40 的汇编器及汇编代码也做了部分说明，与本书配合使用会有更好的效果。4

3 本书中以 [数字] 形式标注的部分在书尾附录中有相对应的内容。

4 同时审校者推荐《UNIX 操作系统教程》，西安电子科技大学出版社于 1985 年 6 月出版，尤晋元主编。这是一本详细剖析 UNIX V6 机制和源码的教材。——审校者注

网罗了操作系统的基本功能

UNIX V6 虽然比较老，但是它实现了构成操作系统的大部分基本功能 。目前最新的操作系统大部分都是以它为基础发展而来的，因此以 **UNIX V6** 为入门教材对我们了解最新的操作系统来说会很有帮助 。

线程、网络、GUI、多核支持、虚拟机等这些 UNIX V6 不具备的功能在近些年的操作系统中得以实现。这些功能当中有很多其实是以 UNIX V6 实现的功能为基础的。

简化的设计

UNIX V6 作为一种早期的操作系统，功能实现比较简单。而最新的操作系统要顾及更多的问题，同时也要考虑到性能的优化，因此实现也更为复杂。如果是首次阅读内核源代码**,**用相对简单的 **UNIX V6** 更合适 。

便于读者对系统有完整的了解

前面已经说过，从代码量上看，通读 UNIX V6 内核源代码对个人来说是可以做到的。如果更进一步，对系统内置的用户程序集（如 shell）的代码或是周边设备的设计文档也有所涉猎的话，就会对包括内核在内的计算机系统整体有更深入细致的了解 。上述系统内置的用户程序集的代码或设计文档与最新产品相比，在实现上更为简单，也更容易理解。

有模拟器可供参考

simh[7] 这款模拟器能够模拟包括 PDP-11 系列的许多处理器，可以用来运行 UNIX V6。因此在阅读源代码时可以随时通过模拟器确认不太明白的地方 。在 simh 上运行 UNIX V6 的方法请参考附录 A.1 的 [8]、[21] 和 [22]。

几个难点

当然，UNIX V6 也存在它特有的问题。UNIX V6 的大部分代码使用了 C 语言编写，而当时的 C 语言还处于初期阶段，在语法规格上与现在的 C 语言有所不同（顺便提一下，C 语言就是因为编写 UNIX 才诞生的）。当时的 C 语言使用 *K&R* 5 之前的语法，因此也被称为 pre K&R。请看下面的例子（代码清单 1 ）。

5 这里指的是 *C Programming Language* 这部讲述 C 语言的名著（Brian W. Kernighan、Dennis M. Ritchie 合著）。根据两位作者姓氏的首字

母，此书也被称为 *K&R*，中文版书名是《C 程序设计语言》，由机械工业出版社出版。——译者注

代码清单 **1**　当时的 **C** 语言的示例

```
struct {
  char high;
  char low;
};

hogehoge(arg) {
  int hoge;
  hoge = arg;
  return hoge->low;
}
```

估计会有读者提出“为什么函数 `hogehoge()` 没有定义返回值和参数”、“为什么对 `int` 型的 `hoge` [6] 可以使用 `hoge->` 这样的写法”等疑问。这是由 C 语言当时的语法决定的。

[6] `hoge` 以及后文中出现的 `fuga` 等是日语中经常使用的伪变量名，类似于英语中的 `foo`、`bar`。——译者注

虽然存在这种语法上的差异，但 pre K&R C 的语法规格说明书 [16] 可以在网上获取， 而且如果是了解当代 C 语言的读者，应该不会很难理解。而比较特殊的 pre K&R C 语法，请参考本书的附录。

另外，和其他很多程序一样，UNIX V6 的一部分代码采用了汇编语言来编写。这对首次接触汇编语言的读者来说可能会有影响。但是，UNIX V6 使用的 PDP-11（参考第 1 章）的汇编语言的规格说明书 [17] 也可以在网上找到，慢慢熟悉就会适应了。

面向的读者

本书主要面向计算机专业的学生，以及从事计算机相关行业的具有初中级水平的技术人员。特别适合那些通过大学课程和其他入门书对操作系统有所了解,但是对具体细节缺乏深入理解 的读者，或是那些对操作系统的具体实现有兴趣 的读者。

需要具备的知识基础

UNIX V6 内核的大部分代码都是用 C 语言写成的，因此读者必须具备 **C** 语言的知识基础 ，也要掌握栈、队列等基本数据结构和算法的知识 。另外，如果能了解计算机的运行原理，比如程序在执行时需要首先加载到内存，然后从程序计数器指定的内存地址读取指令等，就更为理想了。

对那些完全不了解操作系统的朋友来说，本书的难度可能有点大。建议先参考在本书附录部分介绍的入门书籍（附录 A.1 中的 [5] 和 [6]），等掌握了操作系统的基本知识后再来阅读本书，这样可能会达到事半功倍的效果。

本书的结构

第 1 章介绍 UNIX V6 的整体概要。

第 2 章到第 4 章介绍用来管理程序运行的进程（process）。第 2 章概述进程。第 3 章说明进程的控制方法。第 4 章说明以有效利用有限内存空间为目的的交换（swap）处理。

第 5 章和第 6 章介绍因某种事项中断当前进程运行，并转而处理该事项的几种机制。第 5 章首先介绍如何处理周边设备和 CPU 内部发生的中断请求。然后介绍了系统调用（system call），系统调用是利用中断请求来连接用户程序和内核的机制。第 6 章主要介绍信号（signal），信号用于实现进程间通信，会引起处理的中断或使处理内容发生变化。

第 7 章和第 8 章介绍磁盘等设备的 I/O 处理。第 7 章说明块（block）设备子系统，第 8 章说明块设备驱动程序。

第 9 章和第 10 章对文件系统进行说明。文件系统隐藏了块设备的存储细节，向用户提供访问数据的统一方法。第 9 章介绍文件系统的概要，第 10 章对文件的操作加以说明。

第 11 章介绍用来实现进程间数据通信的管道（pipe）。

第 12 章介绍行式打印机的 I/O 处理。

第 13 章介绍终端处理。终端处理使用户能够以会话的方式操作系统。

第 14 章说明系统的启动处理。

内核的整体结构和各章之间的对应关系如图 1 所示。

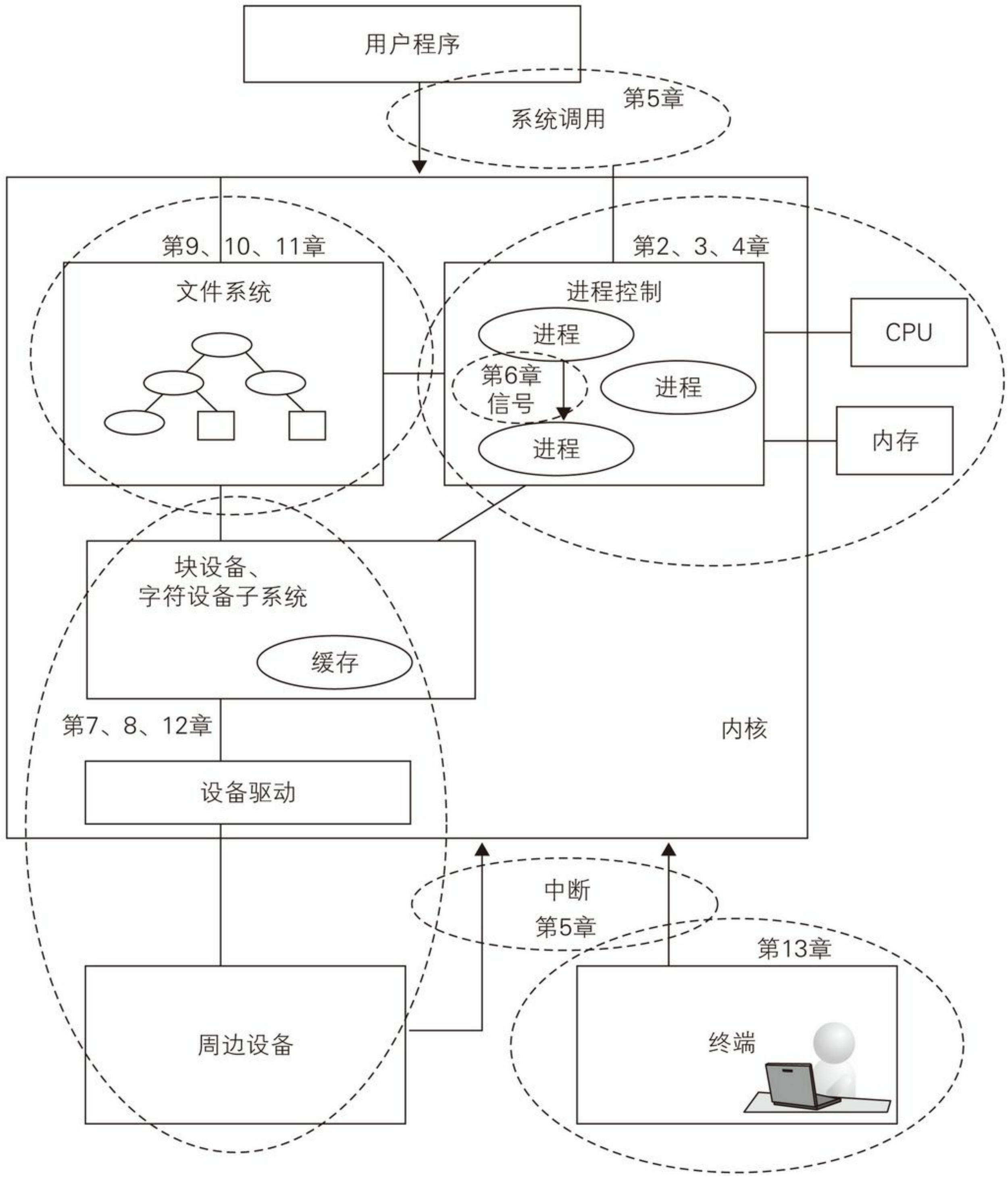


图 **1**　内核整体结

上述章节是根据笔者认为在阅读 UNIX V6 内核源代码时应该遵循的顺序而编排的，将读者可能更感兴趣的有关进程和文件系统的内容尽量安排在前半部分，其他的内容则安排在后半部分。另外，在阅读各章时所需的知识，尽量在前面的章节中加以说明。

关于本书的说明

因为篇幅有限，所以本书以介绍内核的基本处理过程为主要目的，对比较少见的处理和异常处理没有进行特别细致的说明。此外，本书从所有代码中挑选出比较重要的部分（占所有代码的 70%~80%），省略不是很重要的部分或非常简单无需说明的代码。对用汇编语言编写的内容，本书只介绍了启动处理和上下文切换处理。本书中没有介绍的代码希望读者能够自己去分析理解。

另外，考虑到相关内容已经在本书中加以说明，因此在登载的代码中删除了 UNIX V6 开发者加入的注释。但是为了提高可读性，笔者还是保留了一部分原注释，同时也加入了一部分新的注释。

本书旨在成为 **UNIX V6** 内核源代码的阅读指南 。如果想完全理解内核细节，仅阅读本书的说明部分是不够的，还需要深入理解书中涉及的甚至未涉及的代码。

关于代码的说明

本书按照下面的顺序，从抽象程度较高的内容向抽象程度较低的内容进行讲解。

1. 讲解进程或文件系统等功能

2. 讲解实现上述功能的函数或结构体

3. 给出上述函数或结构体的代码并讲解

为了便于读者阅读，本书按照下述格式讲解代码。根据需要，在有些地方调整了代码和表格的讲解顺序。

结构体的讲解

内核中定义了一些重要的结构体，用表格的形式介绍结构体的成员（代码清单 2，表 1 ）。在代码清单标题的括弧中，注明了定义该结构体的文件名。

讲解的过程统一采用“结构体名 . 成员”的形式表示某个构造体内的成员。

代码清单**2**　结构体示例（文件名）

包含结构体的文件名

```
struct hoge {
  int aaa;
  char bbb;
  char ccc;
  char *ddd;
  int eee[3];
};
```

表 **1**　构造体示例

成员	含义
aaa	aaa 的讲解
bbb	bbb 的讲解
ccc	ccc 的讲解
*ddd	ddd 的讲解

eee[]	eee 的讲解

函数的讲解

首先以表格的形式讲解参数（表 2 ），然后给出代码和每行的讲解（代码清单 3 ）。在代码清单标题的括弧中，注明了定义该函数的文件名。虽然此处的例子对每行代码都做了详细（其实没有必要）的说明，但如果代码本身是无需解释的，或事先已做过介绍，那么在代码清单中就不会再做特别说明。

表 **2**　函数参数示例

参数	含义
fuga	在计算中使用

代码清单**3**　函数示例（文件名）

```
1 hogehoge(fuga) {        包含函数的文件名
2   int a, b;
3   a = 10;
4   b = fuga + a;
5   return b;
6 }
```

2　局部变量的定义

3~5　返回参数 fuga 与 10 相加的结果。

对汇编语言代码也以相同的方式说明。因为本书省略了对汇编器规格的说明，所以如果有读者需要简单确认 PDP-11 指令集，请见附录 A.1 的 [9]，需要了解汇编器详细信息的读者详见附录 A.1 的 [20]。

栈的说明

在涉及进程拥有的栈状态时将用表格加以说明（表 3 ）。在第 2 章中会提到，各进程的栈是从地址空间的最高位开始分配，向低位方向增长。但是在表格里位于上方的行表示较低位的地址，也就是说栈在表格里是由下向上增长的。

表 **3**　栈的示例

栈指针（**sp)**	值
->	处于栈顶端的值
	处于栈顶端第 2 位的值

寄存器

在涉及 PDP-11 或周边设备的寄存器时将用表格加以说明（表 4 ）。对未使用的比特位不做介绍。

表 **4**　寄存器示例

比特位	含义
15～5	数据
3～1	状态值
0	错误标志

在讲解中如果需要调用寄存器的某一位，比如说第 0 位的时候，会以“寄存器名 [0]”的形式标注。如果调用了多个比特位，比如说从第 3 位到第 1 位的时候，会以“寄存器名 [3-1]”的形式标注。

数字的表示形式

UNIX V6 中有很多地方采用了八进制数。为了避免混淆基数，在介绍中采用了如下的表示形式。

- 十六进制数 ：0x10
- 十进制数 ：16
- 八进制数 ：020

对二进制数则直接采用 01 或 10 的形式表示。尽管十进制数和八进制数的表示形式比较相似，容易混淆，但相信读者通过上下文还是可以区分的。

另外请注意，PDP-11 的汇编器对不带前缀的数字一律按八进制数处理。

几点有助于增进理解的建议

UNIX V6 内核源代码尽管比较适合初次阅读代码的读者，但是仍然有一定的难度。笔者在此举出几点便于读者增进理解的建议，在阅读代码遇到困难时请酌情参考。

首先通读全书

首先请从头到尾通读本书，掌握内核的全貌。在理解某个模块的代码时，往往需要对相关模块有所了解。前面已经说过，在编排章节时我尽量将相关内容靠前说明，但是这种做法毕竟还是有局限性的。在阅读中遇到难于理解的内容时，可以先跳过它，等到读完全书后再重读这一部分，这可能是一个比较好的办法。

阅读规格说明书和程序员手册

在阅读本书的同时，阅读规格说明书和程序员手册有助于加深理解。不要通过阅读代码去理解程序规格,而应该在理解规格的基础上再去阅读代码 。特别是系统调用部分的规格非常重要。内核实现了最低限度的功能，而这些功能是通过系统调用提供给用户程序的，从这个角度上说，将系统调用的规格看作是内核的规格也并不过分 。

在阅读设备驱动程序时，最好也读一下该设备的规格说明书。设备驱动程序是根据设备规格进行操作的，理解了设备规格后，对理解驱动程序进行的操作无疑会有帮助。

仔细阅读结构体

包括系统内核在内，程序的主要工作就是操作数据。因此，只要把握了作为处理对象的结构体，或是程序中出现的标志变量（flag），就能大致推测出该程序的处理内容。

区分内存空间地址

代码中涉及内存地址的部分，请注意区分该地址指向的是内核的地址空间，还是用户程序的地址空间。

寻找志同道合的朋友

在阅读本书、Lions 书，或是 UNIX V6 内核源代码的时候，组织读书会是一个很好的方法。向他人提出自己的疑问，或是向他人介绍自己理解的内容，这都会加深对内容的理解。在博客上写出自己的想法和心得也是一个很好的方法。

本书的写作原委和谢辞

在写本书之前，笔者对操作系统的大概情况有一定认识，但是完全不了解具体的实现方法，总希望有机会能阅读一下源代码。通过朋友的介绍接触到 *Lions' Commentary on UNIX* 这本书（也就是通称的 Lions 书），在了解到 UNIX V6 的源代码不到 1 万行等信息后，我马上就对此书产生了兴趣。

但是，Lions 书问世已久，写作的时代背景和现在相差迥异，因此对我而言很难理解，阅读速度非常缓慢。此时，又是前面提到的那位朋友向我介绍了一个关于 Lions 书的读书会，我就参加了他们的读书活动。在读书会里，通过请教对内核比较熟悉的成员，或是向他人介绍自己通过预习而掌握的知识，慢慢加深了对 UNIX V6 内核的理解。

理解了系统内核之后，我上学时学到的计算机相关知识之间的联系就在头脑中逐渐清晰起来，同时也切实体会到自己在这个领域的造诣明显地提高了一个层次。

为了和他人分享这种体验，同时也作为学习的备忘录，我开始在博客上记录学习过程。读完了 Lions 书后，博客的连载也告一段落后，我写了一篇总结性的文章，引起了很大反响。

当我确信读者对类似系统内核这种（与应用程序相比较）底层的知识有很大需求之后，便主动向出版社毛遂自荐，商谈以博客文章为基础出版图书的事宜，得到了技术评论社[7]的积极回应。

> [7] 本书原著的出版社。——译者注

本书以笔者一人的微薄之力是难以完成的。如果没有参加读书会，是否能坚持读完 Lions 书都是一个问题。借此机会对读书会的各位成员表示感谢。

读书会成员大野徹先生、小泽广先生、坂项佑树先生、高野了成先生、丰岛隆志先生、七志先生、浜田直树先生、细川生人先生、松泽裕先生和山本英雄先生参与了本书的审阅工作，为笔者提出了很多中肯的意见。一想到如果没有这些意见的话呈现在读者面前的将会是怎样一本书，不免有些后怕。正是因为大家的帮助，本书的品质得到了很大提高，我借此机会表示感谢。

小结

- 阅读内核源代码，可以提高自身的技术水平，加深对计算机的兴趣
- UNIX V6 的历史相对悠久，但是比较适合初次阅读内核源代码的读者
- 在理解规格说明书和手册的基础上，重复阅读本书和内核源代码将会加深理解
- 寻找志同道合的朋友十分重要

第 I 部分　什么是 UNIX V6

在具体介绍代码之前,首先会讲解 UNIX V6 的一些基本概念。

- UNIX V6 内核具有哪些功能
- 内核如何向用户程序提供功能
- 运行 UNIX V6 的系统由怎样的硬件构成

了解了上述概念,对理解内核源代码一定会有帮助。

第 1 章　UNIX V6 的全貌

1.1　什么是 UNIX V6

UNIX V6由肯·汤普逊（Kenneth Lane Thompson）和丹尼斯·里奇（Dennis MacAlistair Ritchie）开发，由贝尔实验室于 1975 年发布。

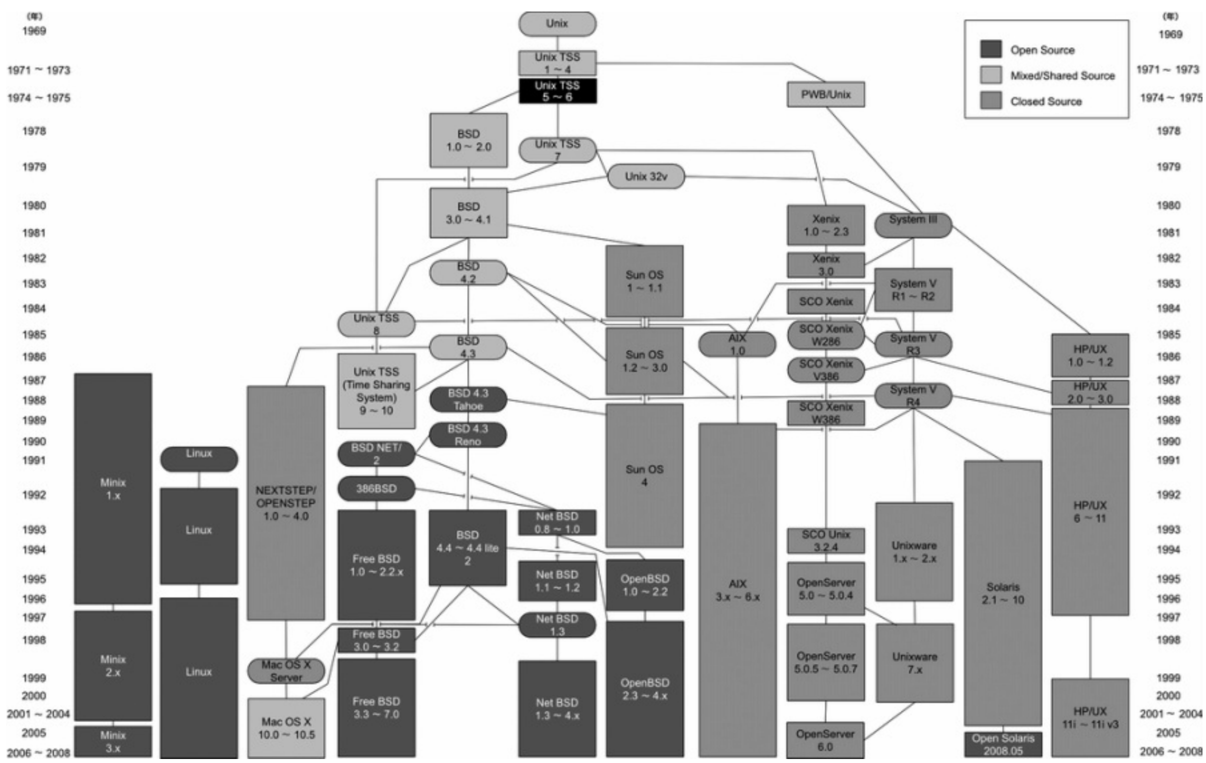
因为大部分代码都是用 C 语言完成的，并且公开了源代码，所以包括大学在内的使用者纷纷将其移植到自己的环境中，UNIX V6 因此得到广泛使用。在移植过程中，有些开发人员加入了独有的功能，其中一些功能后来也被原始版本所采纳。

从 UNIX V6 派生出来的产品还有很多，但是本书以其原始版本**,**即在 **DEC** 公司的 **PDP-11/40** 设备上运行的系统内核作为说明对象 。

1.2　UNIX 的历史

贝尔实验室在发布 UNIX V6 之后，于 1979 年又发布了 UNIX 第 7 版（Seventh Edition Unix，UNIX V7）。加州大学伯克利分校于 1978 年发布了以 UNIX V6 为基础的 BSD 的首个版本。在此之后，UNIX 和 BSD 不断有新版本或派生版本出现。然后又出现了标准化的动向，制定了 POSIX 标准，意在统一各个操作系统所提供的 API。著名的 Linux 也是将 POSIX 标准作为开发目标的。因此，大多数最新的操作系统都和 **UNIX**（**V6** ）有着千丝万缕的联系 。

UNIX 的历史如图 1-1 所示。



图片 ：Eraserhead1, Infinity0 (Own work: CC-BY-SA-3.0, GFDL)

图 **1-1**　**UNIX** 的历史 [1]

[1] http://ja.wikipedia.org/wiki/UNIX

1.3　UNIX V6 内核

UNIX V6 内核提供了这些操作系统所必须具备的如下基本功能：

- 管理运行中的程序（进程）
- 内存管理
- 文件系统
- 文件和周边设备共享 I/O
- 中断
- 支持终端处理

内核向用户提供了经过高度抽象的服务。系统对 CPU 或磁盘的操作细节被内核或设备驱动程序隐藏起来，对用户完全透明。

用户程序通过系统调用 机制访问内核提供的功能。构成计算机系统的软件集合如图 1-2 所示。应用程序利用 UNIX V6 提供的系统内置的用户程序集（处理用户登录的程序，或是守护程序等）、辅助程序（例如 `ls` 、`cat` 等）、程序库等进行处理。系统程序等则利用系统调用（调用内核提供的功能）进行自身的处理。此外，应用程序也可以直接使用系统调用。

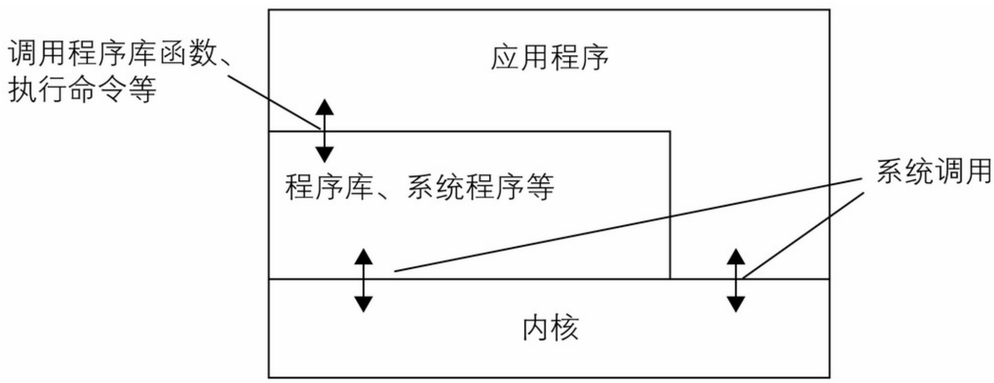


图 **1-2**　构成 **UNIX V6** 的软件集合

内核集中了对系统影响较大的处理，通过限制用户任意执行这些功能，使系统的安 全性和可维护性得以保证。

1.4　构成 UNIX V6 运行环境的硬件

本节对构成 UNIX V6 运行环境的硬件进行介绍。这里说明的内容将贯穿以后的各章，请读者注意理解。

PDP-11

PDP-11 系列是由数字设备公司（Digital Equipment Corporation，DEC）[2] 设计制造的处理装置。本书内容以 PDP-11/40 为对象（图 1-3 ）。

[2] 美国计算机企业。于 1998 年被康柏公司（Compaq）收购，随后康柏公司又与惠普公司（Hewlett Packard）合并。

图 **1-3　PDP-11/40** [3]

[3] http://ja.wikipedia.org/wiki/PDP-11

PDP-11/40 作为一种 16 位计算机，指令和数据基本都是以 16 比特为单位进行处理的。处理器处理数据的单位称为字（word）。对 **PDP-11/40** 而言**,1** 个字的宽度为 **16** 比特 。

PDP-11/40 没有专用的 I/O 总线，而是使用一种名为 Unibus 的总线用于数据的输入输出，Unibus 同时具有 18 比特宽的地址总线 [4] 。PDP-11/40 以及周边设备的寄存器被映射到内存最高位的 8KB 空间，因此可以采用与操作内存相同的方法操作寄存器。这种方式被称为内存映射 **I/O**（**Memory Mapped I/O**）（图 1-4 ）。

[4] Unibus 总线共有 72 条信号线，其中有 18 条用于传输地址（地址总线），另有 16 条用于传输数据（I/O 总线）。——译者注

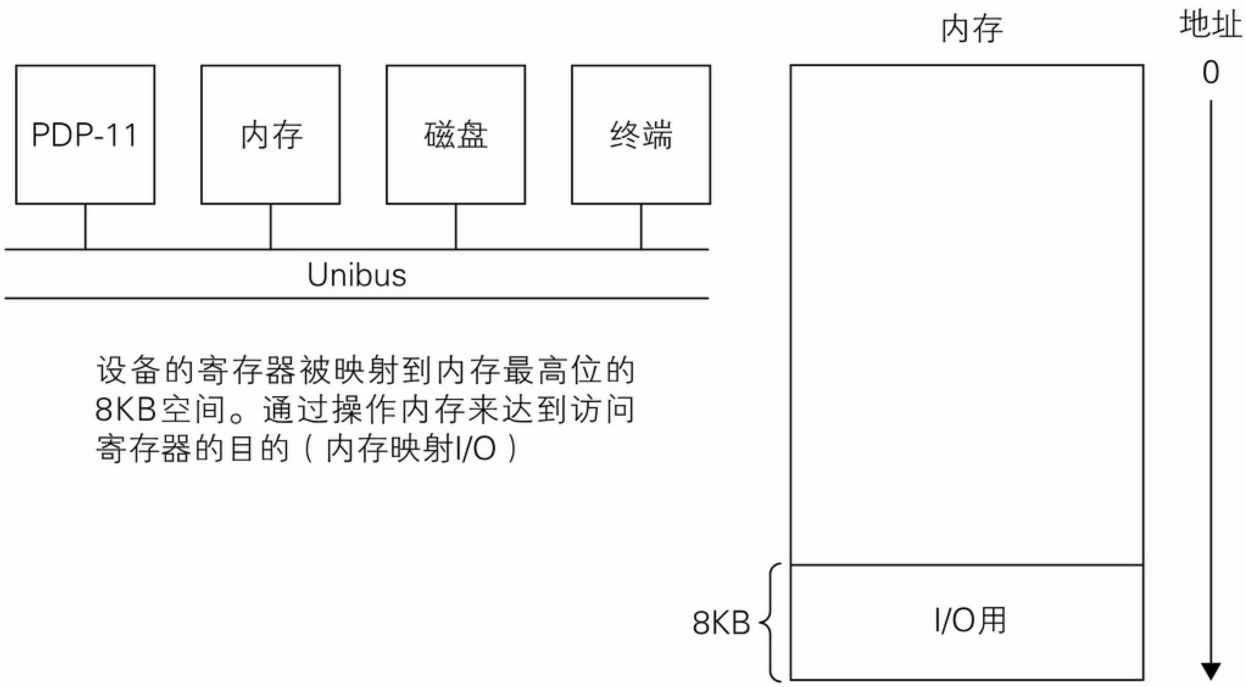


图 **1-4**　通过 **Unibus** 进行连接的设备

利用内存映射 I/O，可以用 C 语言将寄存器的操作写成如代码清单 1-1 所示的形式。integ 的含义请参见本书的附录。

代码清单 **1-1**　寄存器的操作示例

```
/* 寄存器被映射的地址 */
#define REG_ADDRESS 0170000

struct {
  int integ;
};

main() {
    int a;
    a = REG_ADDRESS->integ; /* 从寄存器读取数据 */
    REG_ADDRESS->integ = 0; /* 向寄存器写入数据 */
}
```

PSW

PDP-11/40 拥有一个被称为处理器状态字（Processor Status Word，PSW）的 16 位的寄存器（表 1-1 ）。PSW 表示处理器的状态。

表 **1-1**　**PSW**

比特位	含义
15～14	处理器当前模式（00：内核模式、11：用户模式）

13～12	处理器先前模式（00：内核模式、11：用户模式）
7～5	处理器优先级（7～0）
4	陷入（trap）位
3	N。负位。指令执行结果为负时置 1
2	Z。零位。指令执行结果为 0 时置 1
1	V。溢出位。指令执行中发生溢出时置 1
0	C。借位位。指令执行中发生进位或借位时置 1

处理器模式为 00 时表示内核模式，为 11 时表示用户模式 [5] 。在对系统调用等进行处理时，处理器需要首先从用户模式切换到内核模式。内核模式和用户模式所使用的进程的虚拟地址空间是相互独立的，因此在两种模式间传输数据时，需要了解处理器当前模式和处理器先前模式。

> [5] 在一些资料中，“内核模式”也译为“核心态”，“用户模式”也译为“用户态”。——审校者注

PSW[3-0] 会根据指令的执行结果自动设置。关于处理器优先级和陷入位将在第 5 章做详细说明。

通用寄存器

PDP-11 具有 r0～r7 共 8 个通用寄存器。其中只有 r6 为两个，分别对应用户模式和内核模式。在切换 **PSW** 的当前模式时,**r6** 在硬件上也会自动切换。

UNIX V6 通用寄存器的用途如表 1-2 所示。

表 **1-2**　通用寄存器

寄存器	用途
r0、r1	用于运算、函数的返回值
r2、r3、r4	本地处理
r5	帧指针，环境指针
r6（sp）	栈指针
r7（pc）	程序计数器

上表中的 r5、r6、r7 对理解 UNIX V6 内核尤其重要。

r5 称为帧指针或环境指针。在第 3 章中会做详细介绍。

r6 称为栈指针，它指向各进程所拥有的栈的顶端。

r7 称为程序计数器。处理器从 r7 指示的内存地址读取指令，随后解释并执行该指令。处理完成后 r7 将指向容纳下一条指令的内存地址。

MMU

内存管理单元（MMU，Memory Management Unit）用于地址变换以及访问权限管理。PDP-11/40 以长度为 8KB 的段（segment）或页（page）为单位，对进程所需的内存进行管理。试图访问不具备权限的内存时，MMU 会引发一个陷入异常（参见第 5 章）。MMU 通过称为 APR 的寄存器（页寄存器）对各段进行设定，并将虚拟地址转换为物理地址。APR 和地址变换的详情会在第 2 章中介绍。

PDP-11/40 的 MMU 具有两个状态寄存器（Status Register）SR0 和 SR2。SR0 用于保存出错信息和内存管理的有效标志（表 1-3 ）。SR2 用于保存目标指令的 16 位虚拟地址，可用来确定引起错误的指令（表 1-4 ）。

表 **1-3　SR0**

比特位	含义
15	访问设定错误的页时置 1
14	访问由 PDR（详细请参见第 2 章）规定的页长度以外的区域时置 1
13	试图向只读区域写入数据时置 1
8	维护模式。本书不做详细说明
6~5	出错进程的模式（00 ：内核、11 ：用户）
3~1	页编号。可用于确定引起错误的内存页
0	置 1 时 MMU 对内存的管理有效

表 **1-4　SR2**

比特位	含义
15~0	容纳目标指令的 16 位虚拟地址。当读取指令失败时 SR2 的值不更新。另外，如果 SR0[15-13] 的任意 1 位为 1 时 SR2 的值也不更新

内存

PDP-11/40 使用的内存被称为磁芯内存（Magnetic Core Memory），当时的论文等有时将其称为磁芯（Core），本书为了便于读者理解，仍将其称为内存。

内存以 8 比特（1 字节）为单位赋予地址。地址长度为 18 比特，因此内存容量为 2^{18} =256KB。PDP-11/40 将周边设备的寄存器映射到内存高位 8KB 的地址空间。

处理器利用内存中存储的指令和数据进行运算。

块设备

磁盘设备或磁带设备等块设备可以容纳大量数据。文件系统构筑于块设备之上。对块设备 I/O 处理的介绍参见第 7 章、第 8 章，对文件系统的详细说明参见第 9 章、第 10 章。

块设备的一部分被用做交换空间。为了在有限容量的内存中运行大量程序，内核会把不需要的数据从内存转移到交换空间，或者将需要的数据从交换空间移回内存。这被称为交换处理，在第 4 章中会对此进行详细说明。

行式打印机

行式打印机用于在纸张上打印数据的设备。对行式打印机的说明请见第 12 章。

终端

终端是连接系统与用户的装置。用户可以通过终端对系统进行交互式的操作。终端处理会在第 13 章中说明。

1.5 代码

UNIX V6 源代码对应现在的 BSD 许可证，用户可以在网上获取代码。本书附录中介绍了登载 UNIX V6 源代码的网站 [1] 。

1.6 手册

UNIX V6 向用户提供了名为 UNIX Programmer's Manual（UPM）的手册。该手册包括下述 9 个章节，阅读内核代码时需要特别注意第 **2** 章和第 **4** 章 。如果希望了解可执行程序（a.out）的格式请参考第 5 章。本书在希望读者参考 UPM 的某个章节（比如第 2 章）时，会以“UPM(2)”的形式标注。如果在阅读代码时遇到难以理解的内容，也请参考 UPM。

1. 手册概要

2. 一般命令

3. 系统调用

4. C 语言库函数等

5. 特殊文件 [6] 、设备驱动

> [6] 关于特殊文件的说明请参照 7.1 节。——译者注

6. 文件格式

7. 游戏等

8. 其他

9. 系统管理命令和守护程序

如果想了解内核以外的、系统内置的用户程序集等内容，请参考上述第 1 章、第 3 章、第 5 章、第 8 章。

本书附录中介绍了登载 UNIX V6 手册的网站 [2] 。

1.7 小结

- UNIX V6 由肯·汤普逊和丹尼斯·里奇开发，于 1975 年由贝尔实验室发布
- UNIX V6 可以说是现代操作系统的鼻祖
- UNIX V6 的内核提供了系统运行所必需的基本功能
- 内核向用户隐藏了对硬件的操作，以及保证了对系统具有影响的处理的安全性和可维护性
- 用户程序通过系统调用向内核发出处理请求
- PDP-11/40 及周边设备的寄存器被映射到内存的最高位 8KB 地址空间
- PDP-11/40 具有 r0~r7 的 8 个通用寄存器。其中只有 r6 又分为两个，分别对应内核模式和用户模式。理解内核处理时，r5~r7 尤其重要

第 **II** 部分　进程

内核将执行中的程序作为进程加以管理。内核将内存分配给进程,进程则利用被分配到的内存进行处理。第 II 部分主要对下述内容进行说明。

- 内核如何管理进程
- 内存如何分配给各个进程
- 进程如何访问被分配到的内存
- 如何并行执行多个进程
- 如何有效利用内存空间

阅读这部分内容之后,想必读者对程序内部的运行机制会有更深入的了解。

第 **2** 章　进程

2.1　进程的概要

什么是进程

内核采用进程 的概念对执行中的程序进行管理。一个进程对应一个执行中的程序。

在执行程序时，内核首先将程序（以文件系统中的文件（参见第 9 章）等形式保存在磁盘上）读入内存，然后将此内存区域分配给进程。进程拥有独立的虚拟地址空间，可以使用虚拟地址空间内的地址（虚拟地址），由 MMU 转换为实际的内存地址（物理地址），访问被分配的内存。在使用物理地址管理内存时，本书有时也将内存称为物理内存。

进程具有唯一的进程 ID。由于是以 ID 为识别手段，因此即使同一程序被执行多次生成多个进程，内核也将其识别为不同的进程。此外，由于内存以进程为单位分配，而且进程的虚拟地址空间各自独立，因此各个进程在执行时不会相互影响（图 2-1 ）。由于程序的指令在执行时不会发生变化，因此从节约内存使用量的角度出发，有时也会将同一块内存分配给多个进程的虚拟地址空间。

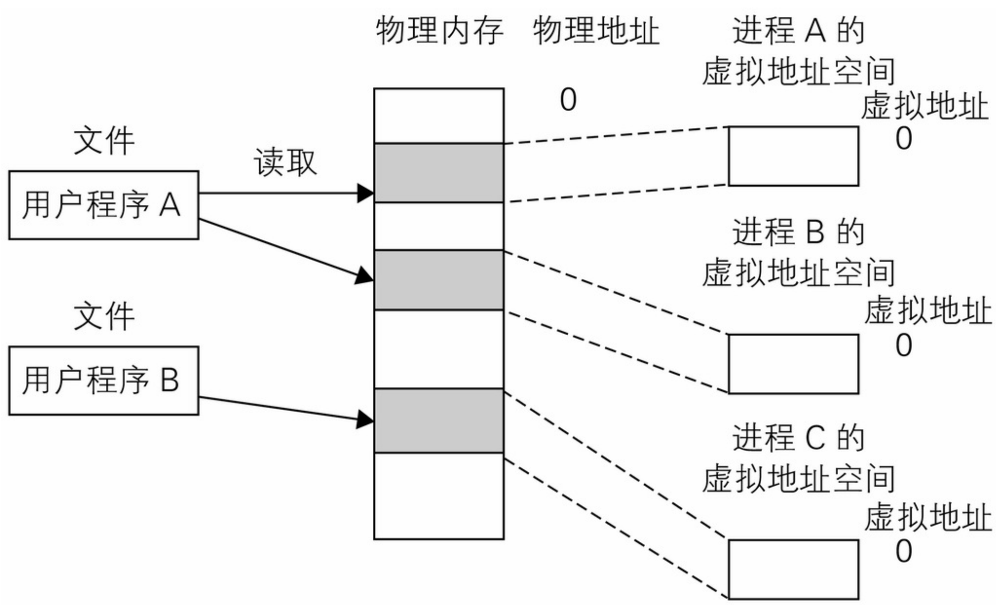


图 **2-1**　相互独立的虚拟地址空间

进程的并行执行

UNIX V6 可供多名用户同时使用。每名用户可同时执行多个程序，因此在任意时刻，系统中都可能存在多个进程（图 2-2 ）。

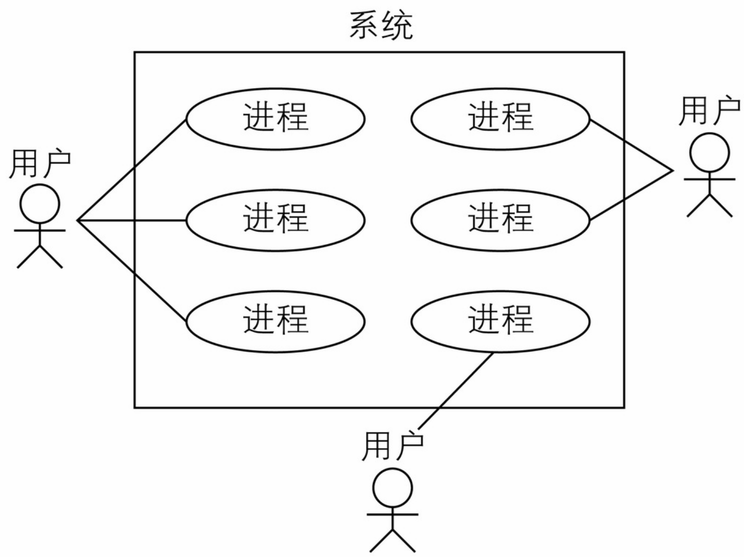


图 **2-2**　系统中存在多个进程

但是，系统中用来处理程序的物理 CPU 只有一个，因此即使存在多个进程，真正处于执行状态的进程仍然只有一个，其他的进程处于待机状态（图 2-3 ）。

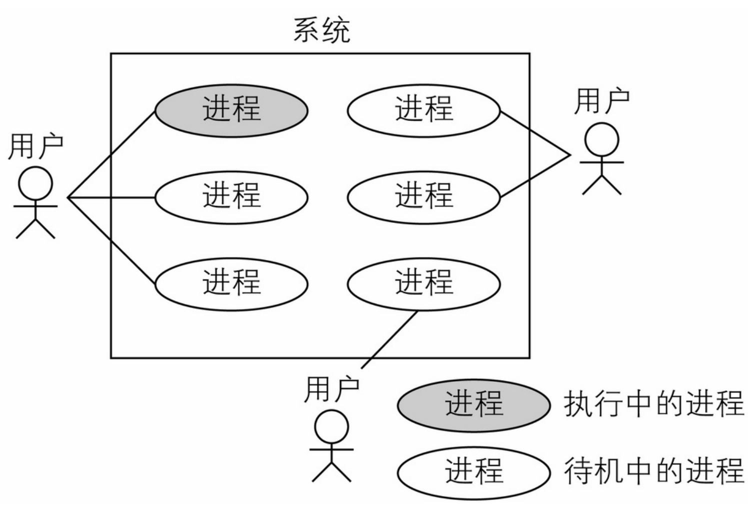


图 **2-3** 可执行的进程只有 **1** 个

内核会随时切换执行中的进程，将时间片分给不同的进程。这样的话，即使只有 **1** 个物理 **CPU,**也可以并行处理多个程序 。采用这种方式的系统被称为分时系统（Time Sharing System，TSS）（图 2-4 ）。

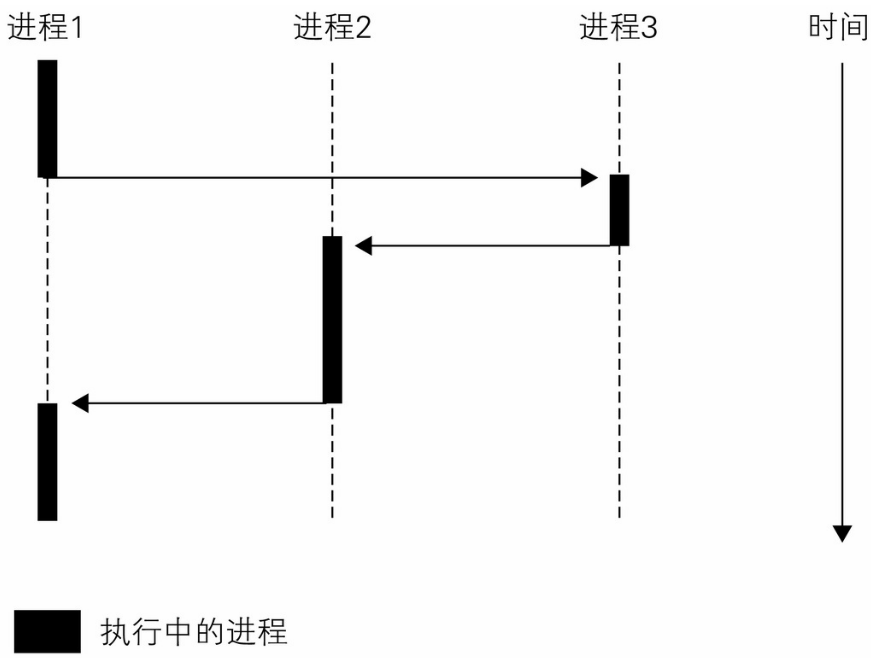


图 **2-4**　**TSS**

尽管在任意时刻都只存在 1 个执行中的进程，但是通过在很短的时间间隔中不断切换执行中进程的方式，使用户产生多个程序在同时运行的错觉。在以后的说明中，将当前执行中的进程称为执行进程。

进程的执行状态

进程的执行状态可以分为两种：可执行状态 和休眠状态 。进程在等待其他资源被释放或等待周边设备处理结束时，会暂时中断自身的处理，进入休眠状态。内核从处于可执行状态的进程中选择 1 个作为执行进程运行（图 2-5 ）。

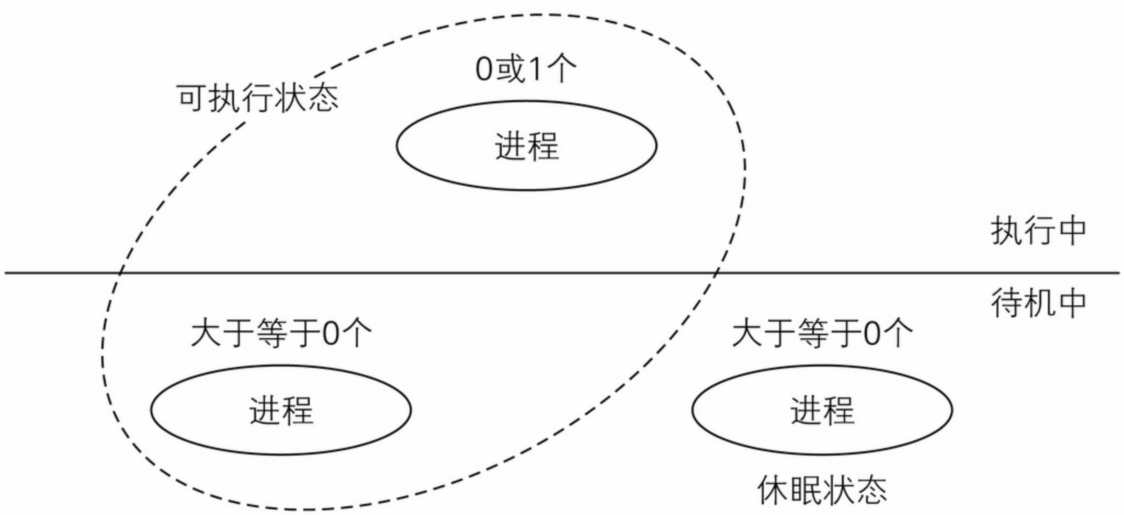


图 **2-5**　进程的执行状态

用户模式和内核模式

如第 1 章所述，处理器具有两种模式：用户模式和内核模式。通过 PSW 能够在两者间进行切换。

切换模式时，映射到虚拟地址的物理内存区域也随之发生变换。虚拟地址在用户模式时映射到用户程序的内存区域，在内核模式时则映射到内核程序的内存区域（图 2-6 ）。内存映射的切换由 MMU 来实现。另外，内核程序在系统启动时被读取到内存中，详细说明请参照第 14 章。

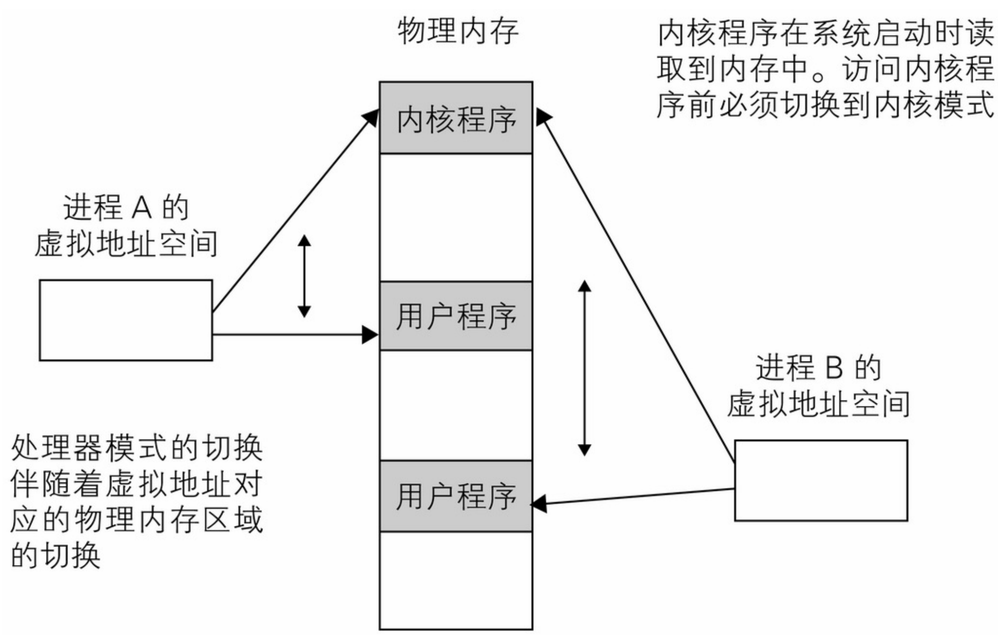


图 **2-6**　用户模式和内核模式

在以后的说明中，将用户模式时的虚拟地址空间称为用户空间，内核模式时的虚拟地址空间称为内核空间。此外，将以用户模式运行的进程称为用户进程，以内核模式运行的进程称为内核进程。

因为用户程序由用户进程处理，所以无法访问加载内核程序的内存区域，也就无法执行由内核实现的功能。为了访问内核功能，用户程序必须通过系统调用 向内核提出访问内核功能的请求。系统调用一旦被执行后，处理器的模式将切换到内核模式，同时虚拟地址空间也会被切换到内核空间，内核功能随之被执行。内核处理结束后，处理器的模式返回到用户模式，继续做一般处理。如上所述，1 个程序（进程）的处理伴随着在用户模式和内核模式之间的多次切换（图 2-7 ）。

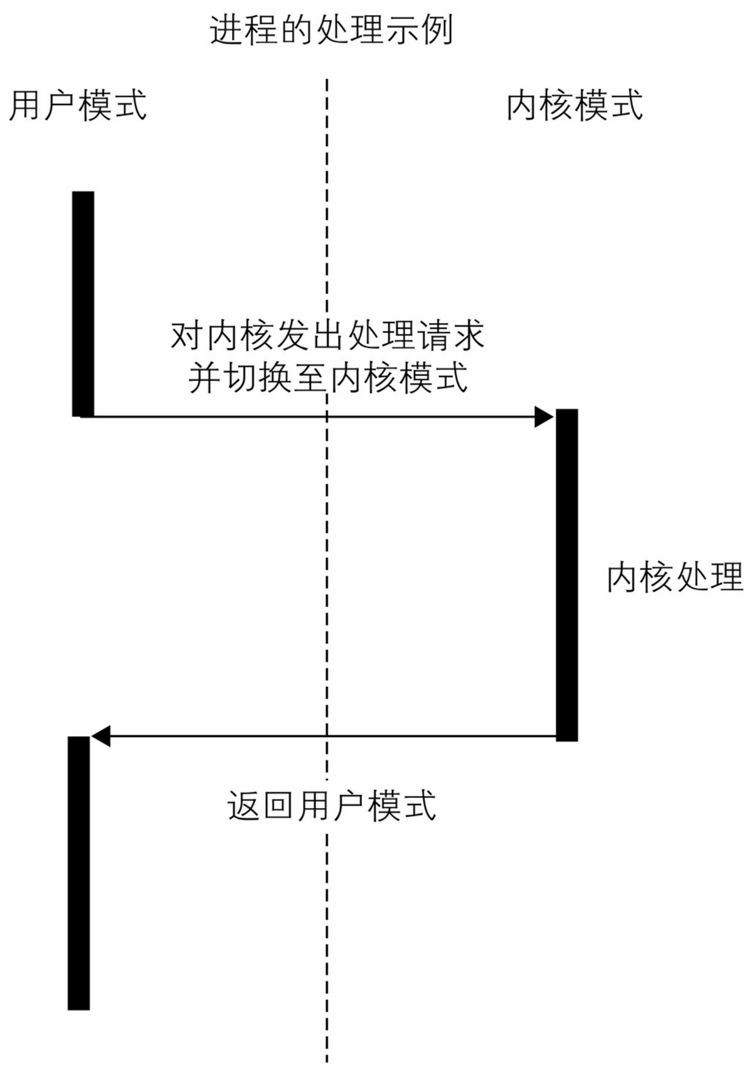


图 **2-7**　进程处理示例

内核提供的功能通常会对系统造成很大影响。如果放任用户程序随意使用内核功能会造成各种各样的问题。通过制定规则强制用户程序使用内核功能时必须提出申请，使系统的安全性得以提高。同时也使用户程序开发人员在编写程序时无需对内核的实现细节有过多了解。

在系统调用等处理中，会出现需要在用户空间和内核空间之间交换数据的情况。为此系统提供了以下用于在用户空间和内核空间之间读写数据的函数：`fubyte()`、`fuibyte()`、`fuword()`、`fuiword()`、`subyte()`、`suibyte()`、`suword()`、`suiword()`。

交换处理

随着进程数量的增多会导致内存容量不足。内核定期将处于休眠状态、重要度较低的进程（所需的数据）会从内存转移到交换空间（swap out，换出），或者将交换空间中已处于可执行状态的进程重新恢复到内存（swap in，换入）。这个处理被称为交换处理，由系统启动时生成的进程执行。

2.2　proc 结构体和 user 结构体

进程的状态信息和控制信息等由 `proc` 结构体和 `user` 结构体管理。每个进程各自会被分配 1 组上述结构体

的实例。proc 结构体常驻内存，而 user 结构体有可能被移至交换空间。

proc 结构体

由 proc 结构体构成的数组 proc[]（代码清单 2-1 ）中的每个元素分别对应一个进程。proc 结构体管理着在进程状态、执行优先级等与进程相关的信息中需要经常被内核访问的那部分信息（表 2-1 ）。

举例来说，内核在（进程切换过程中）选择下一个将被执行的进程时，会首先检查所有进程的状态。这种需要遍历所有进程的情况在其他处理中也会经常出现。由于 proc[] 常驻内存，因此内核可以在很短时间内完成对所有进程状态的检查。假如 proc[] 能够被移至交换空间，内核必须访问交换空间才能取得相应数据，这会导致花费过多时间并引起性能下降。

proc[] 的长度决定了在系统中可以同时存在的进程上限。proc[] 的长度由常量 NPROC 定义，其值为 50（代码清单 2-2 ）。

代码清单 **2-1**　**proc** 结构体（**proc.h)**

```
struct    proc
{
    char    p_stat;
    char    p_flag;
    char    p_pri;
    char    p_sig;
    char    p_uid;
    char    p_time;
    char    p_cpu;
    char    p_nice;
    int     p_ttyp;
    int     p_pid;
    int     p_ppid;
    int     p_addr;
    int     p_size;
    int     p_wchan;
    int     *p_textp;
} proc[NPROC];

/* stat codes */
#define    SSLEEP   1
#define    SWAIT    2
#define    SRUN     3
#define    SIDL     4
#define    SZOMB    5
#define    SSTOP    6

/* flag codes */
#define    SLOAD    01
#define    SSYS     02
#define    SLOCK    04
#define    SSWAP    010
#define    STRC     020
#define    SWTED    040
```

代码清单 **2-2**　**NPROC**（**param.h)**

```
#define    NPROC    50
```

表 **2-1**　**proc** 结构体

成员	含义
p_stat	状态。等于 NULL 时意味着 proc[] 数组中该元素为空。参见表 2-2
p_flag	标志变量。参见表 2-3
p_pri	执行优先级。数值越小优先级越高，下次被执行的可能性也就越大
p_sig	接收到的信号
p_uid	用户 ID（整数）
p_time	在内存或交换空间内存在的时间（秒）
p_cpu	占用 CPU 的累计时间（时钟 tick 数）
p_nice	用来降低执行优先级的补正系数。缺省值是 0，通过系统调用 `nice` 可以设置成用户希望的数值
p_ttyp	正在操作进程的终端
p_pid	进程 ID
p_ppid	父进程 ID
p_addr	数据段的物理地址（单位为 64 字节）[1]
p_size	数据段的长度（单位为 64 字节）
p_wchan	使进程进入休眠状态的原因
*p_textp	使用的代码段（text segment）

[1] 进程图像包括两部分，一部分是常驻内存图像，如 `proc[]`；另一部分是可交换图像（swappable image），如 `PPDA`、数据区域、栈区域等，这一部分可以被交换到磁盘上。p_addr 是指向进程的可交换图像在内存或磁盘上的地址。p_size 是进程可交换图像的大小。——审校者注

表 **2-2**　进程的状态

状态	含义
SSLEEP	高优先级休眠状态。执行优先级为负值
SWAIT	低优先级休眠状态。执行优先级为 0 或正值
SRUN	可执行状态

SIDL	进程生成中
SZOMB	僵尸状态
SSTOP	等待被跟踪（trace）

表 **2-3**　进程的标志常量

标志常量	含义
SLOAD	进程图像处于内存中（未被换出至交换空间）
SSYS	系统进程，不会被换出至交换空间。在 UNIX V6 中只有 `proc[0]` 是系统进程
SLOCK	进程调度锁。不允许进程图像被换出至交换空间
SSWAP	进程图像已被换出至交换空间。由于被换出至交换空间，`user.u_rsav[]` 的值不再有效。必须从 `user.u_ssav[]` 中恢复 r5、r6 的值
STRC	处于被跟踪状态
SWTED	在被跟踪时使用

user 结构体

`user` 结构体（代码清单 2-3、表 2-4 ）用来管理进程打开的文件或目录等信息。由于内核只需要当前执行进程的 `user` 结构体，因此当进程被换出至交换空间时，对应的 `user` 结构体也会被移出内存。

代码清单 **2-3**　**user** 结构体（**user.h)**

```
struct user
{
    int     u_rsav[2];
    int     u_fsav[25];
    char    u_segflg;
    char    u_error;
    char    u_uid;
    char    u_gid;
    char    u_ruid;
    char    u_rgid;
    int     u_procp;
    char    *u_base;
    char    *u_count;
    char    *u_offset[2];
    int     *u_cdir;
    char    u_dbuf[DIRSIZ];
    char    *u_dirp;
    struct {
        int     u_ino;
        char    u_name[DIRSIZ];
    } u_dent;
    int     *u_pdir;
    int     u_uisa[16];
    int     u_uisd[16];
```

```
    int     u_ofile[NOFILE];
    int     u_arg[5];
    int     u_tsize;
    int     u_dsize;
    int     u_ssize;
    int     u_sep;
    int     u_qsav[2];
    int     u_ssav[2];
    int     u_signal[NSIG];
    int     u_utime;
    int     u_stime;
    int     u_cutime[2];
    int     u_cstime[2];
    int     *u_ar0;
    int     u_prof[4];
    char    u_intflg;
} u;

/* u_error codes */
#define    EFAULT     106
#define    EPERM      1
#define    ENOENT     2
#define    ESRCH      3
#define    EINTR      4
#define    EIO        5
#define    ENXIO      6
#define    E2BIG      7
#define    ENOEXEC    8
#define    EBADF      9
#define    ECHILD     10
#define    EAGAIN     11
#define    ENOMEM     12
#define    EACCES     13
#define    ENOTBLK    15
#define    EBUSY      16
#define    EEXIST     17
#define    EXDEV      18
#define    ENODEV     19
#define    ENOTDIR    20
#define    EISDIR     21
#define    EINVAL     22
#define    ENFILE     23
#define    EMFILE     24
#define    ENOTTY     25
#define    ETXTBSY    26
#define    EFBIG      27
#define    ENOSPC     28
#define    ESPIPE     29
#define    EROFS      30
#define    EMLINK     31
#define    EPIPE      32
```

表 **2-4　user** 结构体

成员	含义
u_rsav[]	进程切换时用来保存 r5 和 r6 的当前值
u_fsav[]	处理器为 PDP-11/40 时不使用
u_segflg	读写文件时使用的标志变量

u_error	出错时用来保存错误代码。参见表 2-5
u_uid	实效用户（ effective user）[2] ID
u_gid	实效组（effective group）ID
u_ruid	实际用户（real user）ID
u_rgid	实际组（real group）ID
*u_procp	此 `user` 结构体对应的数组 `proc[]` 的元素
*u_base	读写文件时用于传递参数
*u_count	读写文件时用于传递参数
*u_offset[]	读写文件时用于传递参数
*u_cdir	当前目录对应的数组 `inode[]` 的元素
u_dbuf[]	供函数 `namei()` 使用的临时工作变量，用来存放文件和目录名
*u_dirp	在读取由用户程序或内核程序传来的文件的路径名时使用
u_dent	供函数`namei()` 使用的临时工作变量，用来存放目录数据。`u_ino` 存放inode 编号，`u_name` 存放文件和目录名
*u_pdir	供函数 `namei()` 存放对象文件和目录的父目录
u_uisa[]	用户 PAR 的值
u_uisd[]	用户 PDR 的值
u_ofile[]	由进程打开的文件
u_arg[]	用户程序向系统调用传递参数时使用
u_tsize	代码段的长度（单位为 64 字节）
u_dsize	数据区域的长度（单位为 64 字节）
u_ssize	栈区域的长度（单位为 64 字节）
u_sep	处理器为 PDP-11/40 时此项基本为 0
u_qsav[]	在系统调用处理中处理信号时用来保存 r5 和 r6 的当前值

u_ssav[]	当进程被换出至交换空间，导致 `user.u_rsav[]` 的值不再有效时，用于保存 r5 和 r6 的当前值
u_signal[]	用于设置收到信号后的动作
u_utime	用户模式下占用 CPU 的时间（时钟 tick 数）
u_stime	内核模式下占用 CPU 的时间（时钟 tick 数）
u_cutime[]	子进程在用户模式下占用 CPU 的时间（时钟 tick 数）
u_cstime[]	子进程在内核模式下占用 CPU 的时间（时钟 tick 数）
*u_ar0	系统调用处理中，操作用户进程的通用寄存器或 PSW 时使用
u_prof[]	用于统计，本书不做说明
u_intflg	标志变量，用于判断系统调用处理中是否发生了对信号的处理

[2] 在一些资料中，实效用户、实效组、实际用户、实际组也被称为有效用户、有效组、真实用户、真实组。以实效用户和实际用户为例，登录系统时实际用户就已经确定，比如是 101。若用这个用户运行程序，则实效用户为 101；若用 `su` 命令以超级用户身份运行，则实效用户为 0（超级用户），而实际用户仍为 101。——审校者注

表 **2-5**　错误代码 [3]

[3] 此列表中对错误代码的有些描述并不全面，请谨慎参考。——审校者注

错误代码	含义
EFAULT	在用户空间和内核空间之间传送数据失败等
EPERM	当前用户不是超级用户
ENOENT	指定文件不存在
ESRCH	信号的目标进程不存在，或者已结束
EINTR	系统调用处理中对信号做了处理
EIO	I/O 错误
ENXIO	设备编号所示设备不存在
E2BIG	通过系统调用 `exec` 向待执行程序传送了超过 512 字节的参数
ENOEXEC	无法识别待执行程序的格式（魔术数字，magic number）

EBADF	试图操作未打开的文件，或者试图写入用只读模式打开的文件，或者试图读出用只写模式打开的文件
ECHILD	执行系统调用 `wait` 时，无法找到子进程
EAGAIN	执行系统调用 `fork` 时，无法在数组 `proc[]` 中找到空元素
ENOMEM	试图向进程分配超过可使用容量以上的内存
EACCES	没有对文件或目录的访问权限
ENOTBLK	不是代表块设备的特殊文件
EBUSY	执行系统调用 `mount` 、`umount` 时，作为对象的挂载点仍在使用中
EEXIST	执行系统调用 link 时该文件已经存在
EXDEV	试图对其他设备上的文件创建连接
ENODEV	设备编号所示设备不存在
ENOTDIR	不是目录
EISDIR	试图对目录进行写入操作
EINVAL	参数有误
ENFILE	数组 `file[]` 溢出
EMFILE	数组 `user.u_ofile[]` 溢出
ENOTTY	不是代表终端的特殊文件
ETXTBSY	准备加载至代码段的程序文件曾被其他进程当做数据文件使用。或者对准备加载至代码段的程序文件进行了写入操作
EFBIG	文件尺寸过大
ENOSPC	块设备的容量不足
ESPIPE	对管道文件执行了系统调用 `seek`
EROFS	试图更新只读块设备上的文件或目录
EMLINK	文件连接数过多
EPIPE	损坏的管道文件

内核可以通过全局变量 u 访问执行进程 的 user 结构体（代码清单 2-4 ），理由将在本章最后介绍。

代码清单 **2-4　u（conf/m40.s)**

```
.globl    _u
_u    = 140000
```

2.3　为进程分配的内存

代码段和数据段作为两个连续的物理内存区域被分配给进程。进程通过虚拟地址访问被分配的物理内存区域。

代码段

代码段是只读 的，用来存放作为程序指令的机器代码。某个程序在被同时执行多次时，各进程共享 同一个代码段。代码段通过数组 text[] 进行管理，长度由user.u_tsize 表示。

数据段

数据段用来存放程序使用的变量等数据。数据段与代码段的不同之处是不会被多个进程共享 。如果它允许被共享的话，某个进程的处理就会被别的进程影响。

数据段的物理地址和长度分别由 proc.p_addr 和 proc.p_size 表示。数据段从低位地址开始，依次由下述 3 个部分组成（图 2-8 ）。

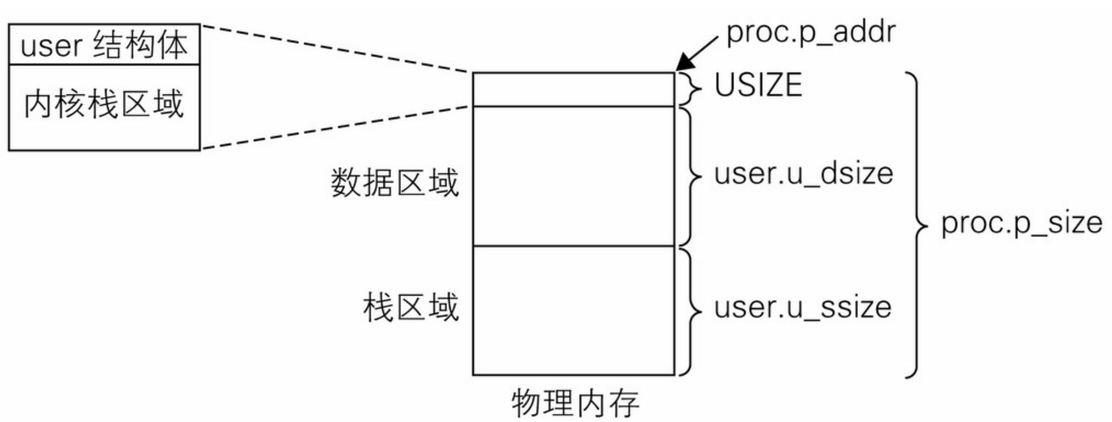


图 **2-8**　数据段

- **PPDA（Per Process Data Area）**
    - user 结构体和内核栈区域构成
    - 长度为 USIZE×64 字节 =1KB（代码清单 2-5）。从用户空间无法访问

    代码清单 **2-5　USIZE（param.h)**

```
#define    USIZE    16
```

- 内核栈区域被用作内核处理时的临时工作区域。每个进程都具有供内核模式使用的工作区域

- 数据区域

  由存放全局变量或 `bss` 等静态变量的区域和进程用来动态管理内存的堆区域构成。扩展堆区域需要通过系统调用来完成。扩展从虚拟地址的低位向高位方向进行，长度由 `user.u_dsize` 表示。

- 栈区域

  用来暂时存放函数的参数或局部数据。长度根据需要自动扩展。栈区域的扩展从虚拟地址的最高位向低位方向进行，长度由 `user.u_ssize` 表示。

虚拟地址空间

进程拥有 64KB 的虚拟地址空间，通过长度为 16 比特的虚拟地址访问物理内存。虚拟地址由 MMU 转换成长度为 18 比特的物理地址（表 2-6 ）。

表 **2-6**　虚拟地址空间和物理地址空间

地址空间	地址	容量
虚拟地址空间	16 比特的虚拟地址	64KB
物理地址空间	18 比特的物理地址	256KB

代码段位于虚拟地址空间最低位的地址，其后为数据区域，数据区域的起始地址以 8KB 为边界对齐。栈区域被分配在虚拟地址空间的最高位地址（图 2-9 ）。

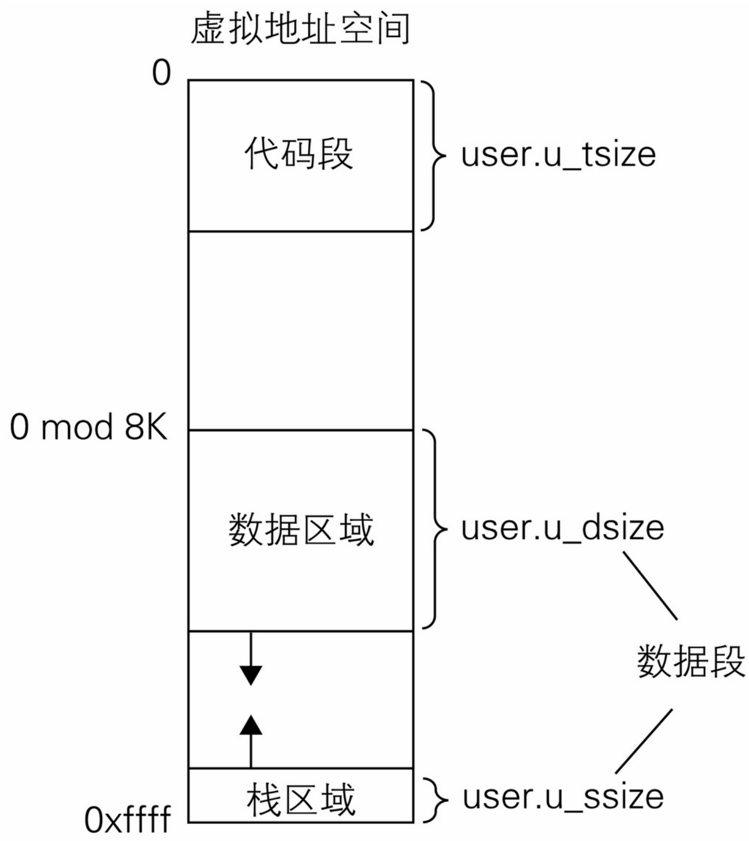


图 **2-9**　进程的虚拟地址空间

每个进程都拥有独立的虚拟地址空间 。举例来说，某个进程的虚拟地址 A 和其他进程的虚拟地址 A 所指向的物理地址是不同的。但是，进程间共享代码段时不受此限制（图 2-10 ）。

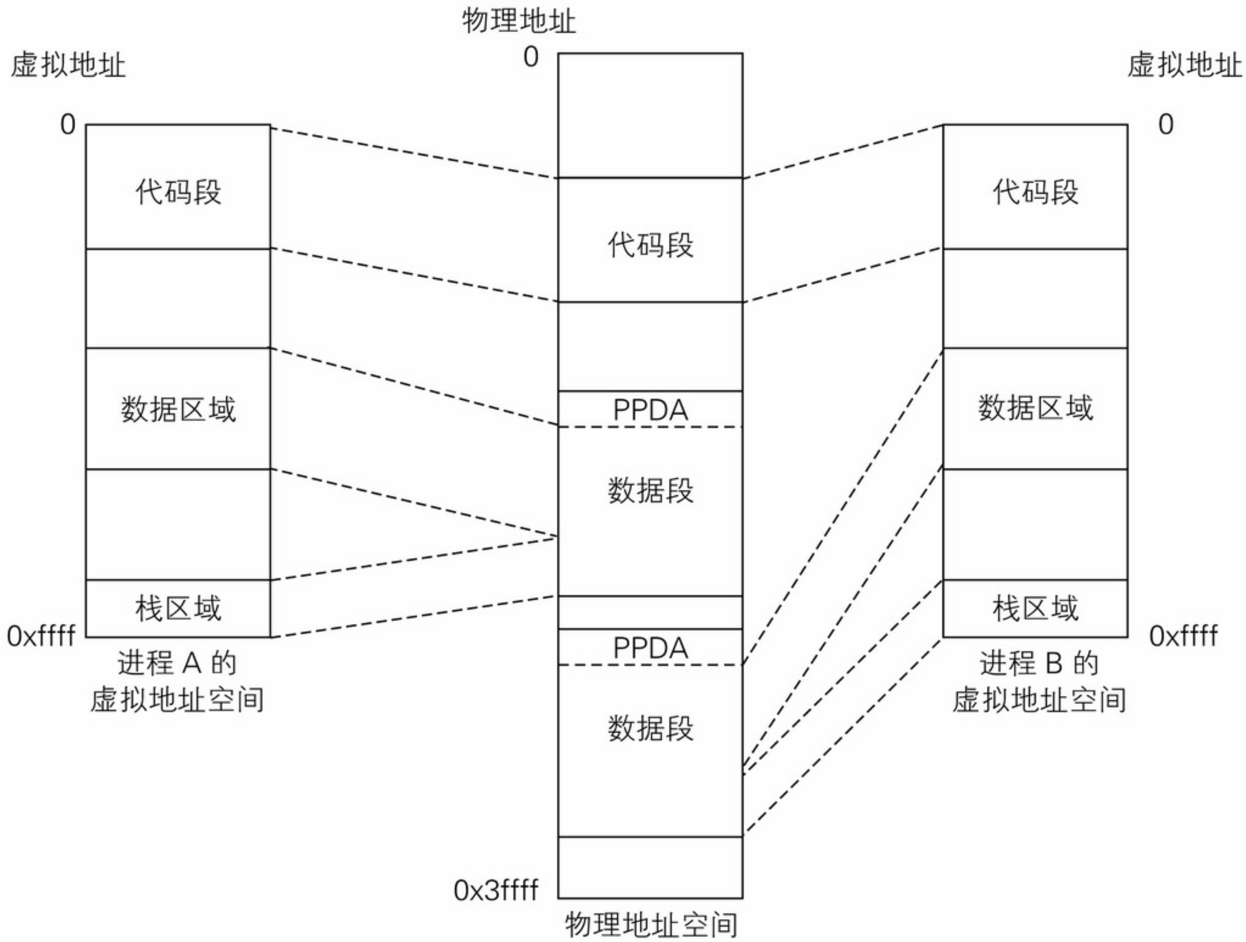


图 **2-10**　物理地址空间和虚拟地址空间

在虚拟地址空间中，数据区域和栈区域可被视作不同的段。此外，正如将在第 3 章中介绍的那样，内核为了使数据区域和栈区域成为不同的段（页），对 APR 进行了设定。因此，在虚拟地址空间中，也可以把数据区域和栈区域分别称为数据段和栈段。本书为了和物理内存中按照连续地址分配的数据段（PPDA+ 数据区域 + 栈区域）加以区分，在以后的说明中还是统一将其称为数据区域和栈区域。

使用虚拟地址具有以下优点。

程序能够使用以任意地址为起点的内存空间

每个程序都可以将以任意地址为起点的内存空间作为自己使用的内存区域，无需考虑该虚拟地址与物理地址的对应关系。

另外，尽管在发生交换处理时分配给进程的物理地址会发生变化，但是由于虚拟地址始终保持不变，因此开发人员在编写程序时无需考虑这个问题。

实现对内存访问的管理

便于实现对内存访问的管理。如果程序能够直接访问物理地址，也就有可能访问其他进程正在使用的物理内存区域。而使用虚拟地址，通过将各个进程的虚拟地址映射到不同的物理地址，就可以保证虚拟地址空间的独立性。

此外，在试图访问未被分配给自己的内存区域，或是访问不具有访问权限的区域时，可以通过 MMU 触发

异常，中止进程的处理。

提高内存的使用效率

在需要确保一定长度的连续的内存区域时，通过将不连续的物理内存区域映射到连续的虚拟内存区域，可以提高物理内存的使用效率。但是，对 UNIX V6 而言，代码段和数据段本身与连续的物理内存区域相对应，系统并未进行将细小的物理内存区域的集合映射为连续的虚拟地址区域的处理。

变换地址

MMU 通过 APR（Active Page Register）寄存器将虚拟地址变换为物理地址（图 2-11 ）。1 个 APR 由 1 个 PAR（Page Address Register）寄存器（表 2-7 ）和 1 个 PDR（Page Description Register）寄存器（表 2-8 ）构成。

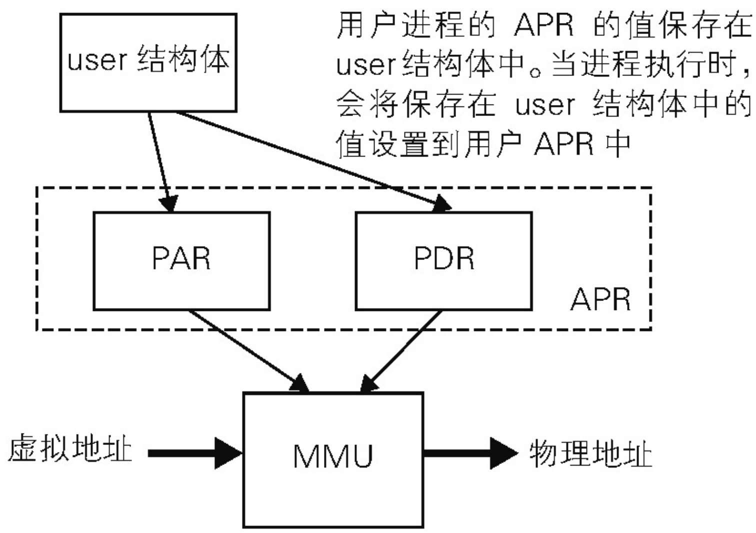


图 **2-11**　变换地址

表 **2-7**　**PAR**

比特位	含义
11~0	基地址（以64字节为单位）

表 **2-8**　**PDR**

比特位	含义
14~8	页的块数
6	更新标志。表示页是否被更新
3	值为 1 时页按高位地址向低位地址的方向进行分配
2~1	页的访问控制方法。00：未分配、01：只读、11：读写

PDP-11/40 拥有分别供内核模式和用户模式使用的两套 **APR**。

通过切换 PSW 的当前模式（PSW[15-14]），可以在硬件上切换 MMU 参照的 APR，从而达到切换虚拟地址空间的目的（图 2-12 ）。

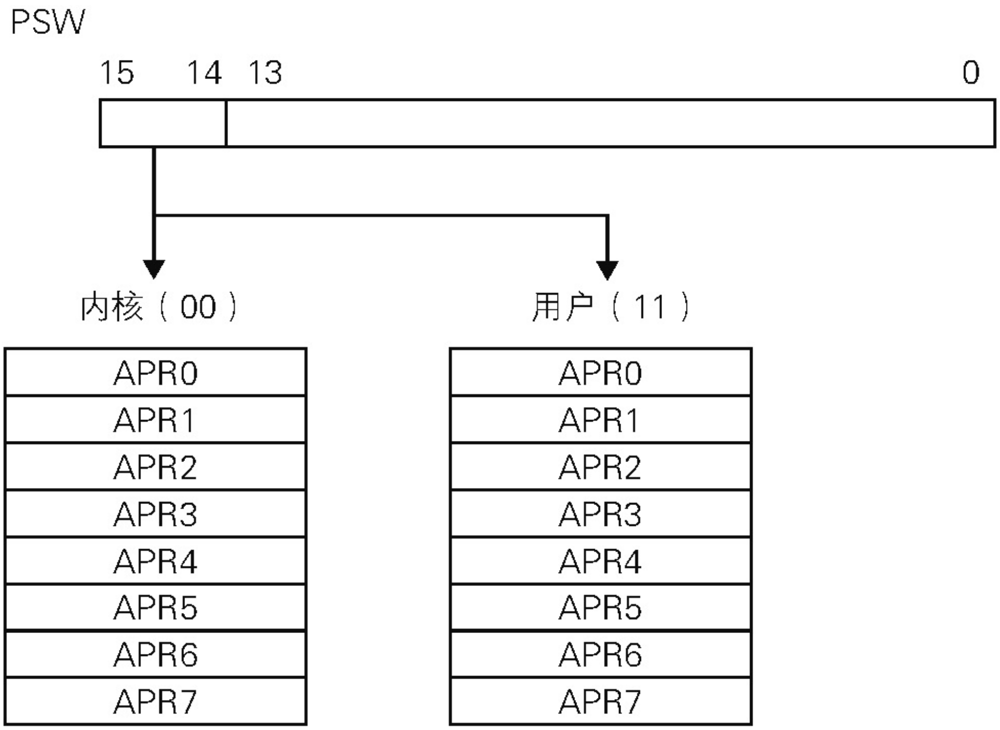


图 **2-12**　**APR** 的切换

内核通过向与执行进程相对应的、供用户进程使用的 APR 设定适当的值，保证各用户进程拥有独立的虚拟地址空间。用户进程用 APR 的值保存在 `user` 结构体中，当该进程成为执行进程时，会将保存在 `user` 结构体中的值设置到用户进程用 APR 中。

APR 共有 **8** 组，编号为 **0** 到 **7** 。进程的虚拟地址空间以页或段为单位进行管理，1 组 APR 对应 1 页。PAR 用来保存与各页物理地址的基地址有关的信息，PDR 用来保存各页的块（以 64 字节为单位）数以及是否允许访问等信息。每一页最多可以被分配 128 个块（8KB）。

虚拟地址的高位3比特决定了对应的页（APR），PAR 的 11~0 比特决定了作为物理地址基地址的块地址，加上虚拟地址的 12~6 比特得到物理内存的块地址，再加上作为块内偏移值的虚拟地址的 5~0 比特，就得到了最后的物理地址（图 2-13 ）。

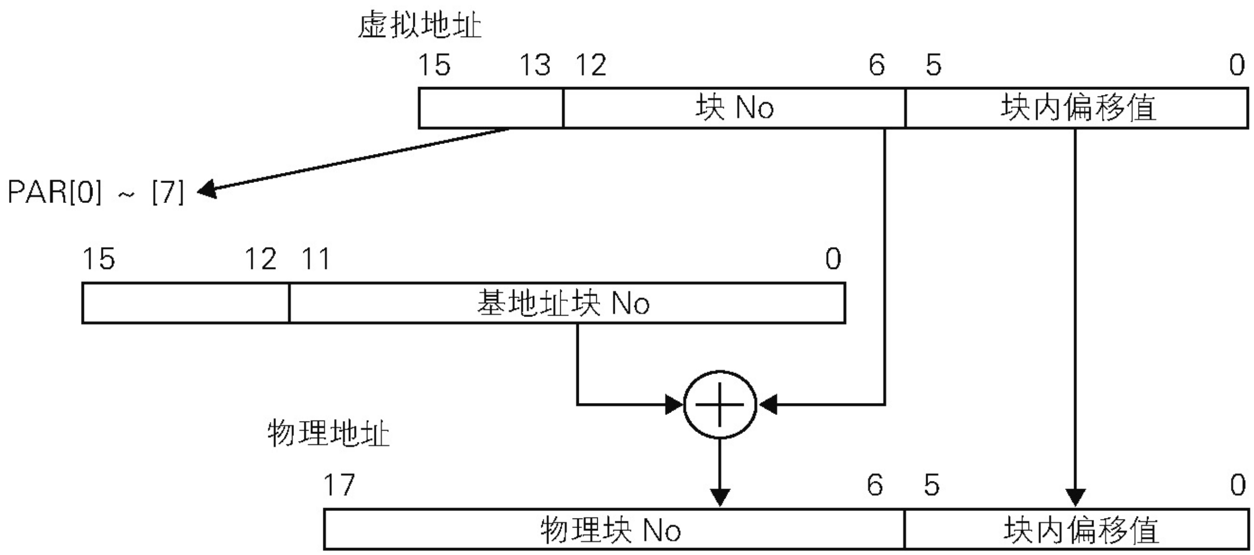


图 **2-13**　物理地址的计算

内核之所以可以通过全局变量 **u** （**0140000** ）访问执行进程的 **user** 结构体**,**是因为内核对供内核模式使用的 **APR** 做了相应的设定 。内核将内核模式使用的编号为 6 的 PAR 设为执行进程的数据段的物理地址（`proc.p_addr` ）（图 2-14 ）。

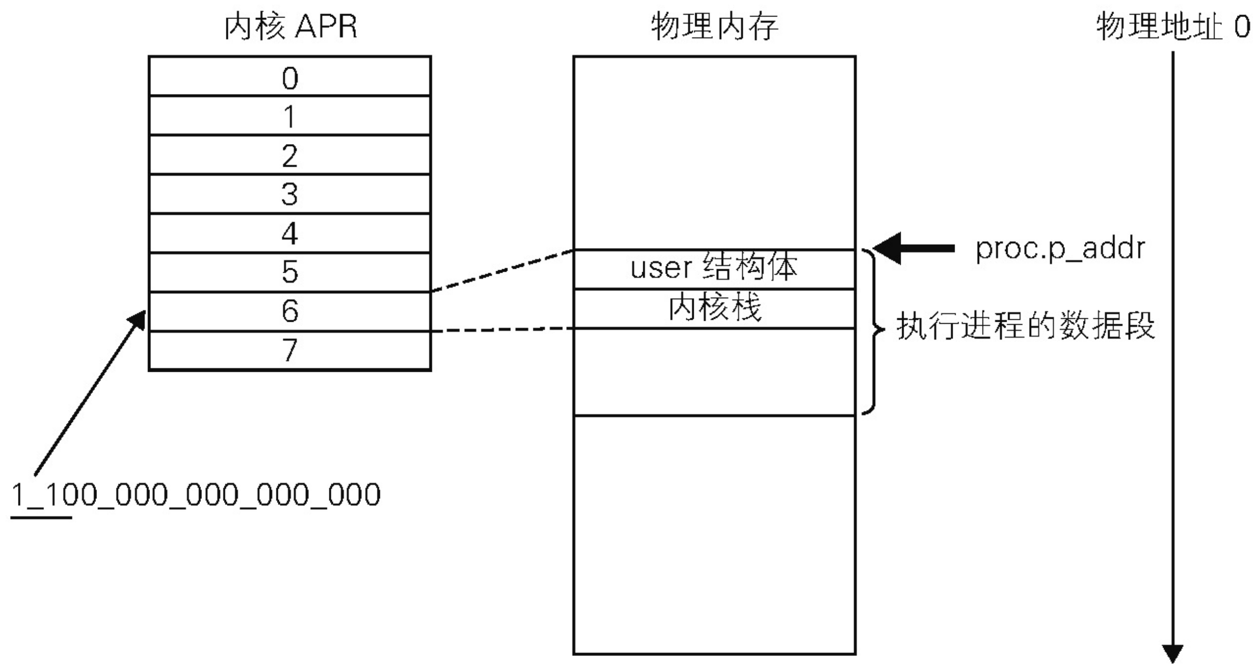


图 **2-14**　通过变量 **u** 访问 **user** 结构体

地址 0140000[4] 的高位 3 比特为 6，这意味着内核模式使用的编号为 6 的 APR 将被选择。由于低位的比特全部为 0，因此 u 指向内核空间第 6 页的起始位置。

[4] 注意，此处为八进制地址。——译者注

在之后的说明中，统一将内核模式使用的 APR 称为“内核 APR”，将用户模式使用的 APR 称为“用户 APR”，将起始编号从 0 开始的编号为 *n* 的 APR 标注为 APR*n* 。此外，为了区分内存的块与块设备的块

（512 字节），在之后的说明中，会将内存的块标注为“以 64 字节为单位的区域”等形式。

2.4 小结

- 执行中的程序以进程为单位进行管理
- 通过不断切换执行中的进程，达到并行执行多个程序的效果
- proc 结构体和 user 结构体用来保存进程的控制和状态信息
- proc 结构体常驻内存，而 user 结构体有可能成为内存交换的对象
- 进程拥有各自独立的虚拟地址空间
- 进程被分配了代码段及数据段的两个连续的物理内存区域
- 数据段由 PPDA、数据区域、栈区域 3 部分组成
- 代码段被映射到虚拟地址空间的最低位地址
- 数据区域被映射到位于代码段之后的虚拟地址空间，起始地址以 8KB 为边界对齐。通过系统调用，数据区域可以向虚拟地址的高位方向扩展
- 栈区域被映射到虚拟地址空间的最高位地址。根据需要向虚拟地址的低位方向自动扩展

第 **3** 章　进程的管理 **I**

3.1　进程的生命周期

进程典型的生命周期如下所示（图 3-1 ）。

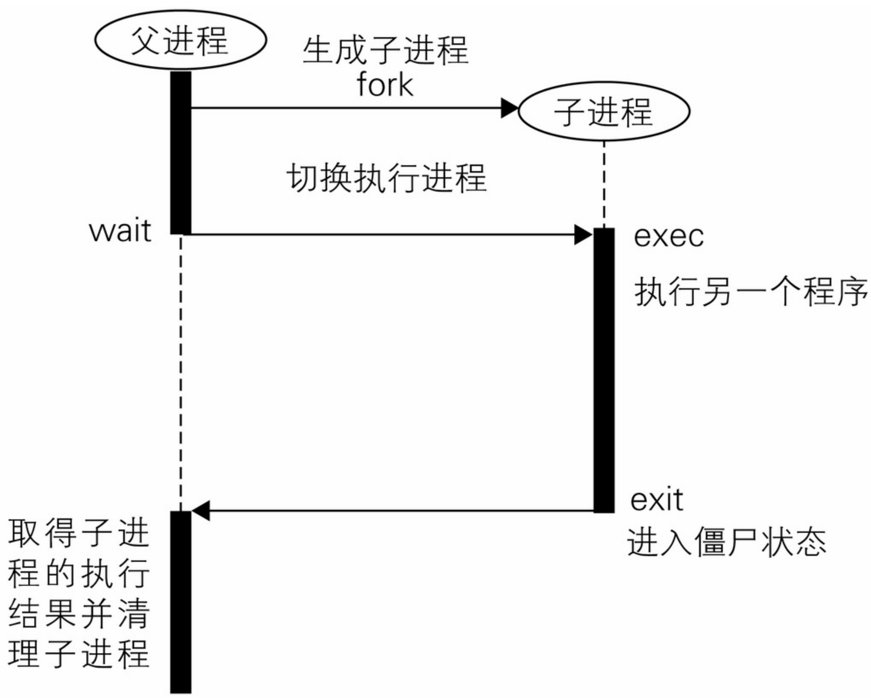


图 **3-1**　进程的生命周期

1. 某个进程通过系统调用 **fork** ，创建一个用于执行程序的进程。生成此进程的进程称为父进程**,**被生成的进程称为子进程。子进程通过复制父进程的数据得以创建 。

2. 父进程执行系统调用**wait** ，进入等待状态直到子进程处理结束。

3. 当控制权转移到子进程后，子进程通过系统调用 **exec** 将程序读取到内存并开始执行。

4. 当程序执行完毕后，子进程通过系统调用 **exit** 结束自身的运行并进入僵尸 状态，控制权交回父进程。

5. 父进程得到子进程的执行结果后清理子进程。

代码清单 3-1 是用 C 语言编写的进行上述处理的例子。C 语言的函数库提供了用来包装系统调用的函数，使用户程序可以像执行普通函数那样访问系统调用。

代码清单 **3-1**　进程的生命周期

```
1 i = fork();
2 if(i == 0) {
3   execv(program_name, argv);
4   exit();
5 }
6 while(wait() != i);
```

1　执行 `fork()` 将生成一个进程。父进程通过 `fork()` 得到子进程的 ID（非 0 的整数）。

6　执行 `wait()` 后进入等待状态直至子进程处理结束。父进程进入休眠状态，（如果此时不存在其他执行优先级更高的进程）控制权将被转交给子进程。

1　子进程从 `fork()` 内部开始进行自身的处理，并从 `fork()` 返回 0。

3　在程序最后，执行由编译器插入的用以访问系统调用 `exit` 的指令以终止进程。

4　此处加入 `exit()` 是为了防止 `execv()` 执行失败。

6　控制权被交回父进程。通过 `wait()` 返回已结束的子进程的 ID。

3.2　创建进程

进程的复制

通过将父进程的数据复制到子进程**,**以此创建新的进程 。由于可以将此过程看做对进程的分支处理，因此创建进程的系统调用被命名为 `fork` 。

被复制的数据如下所示（图 **3-2** ）。

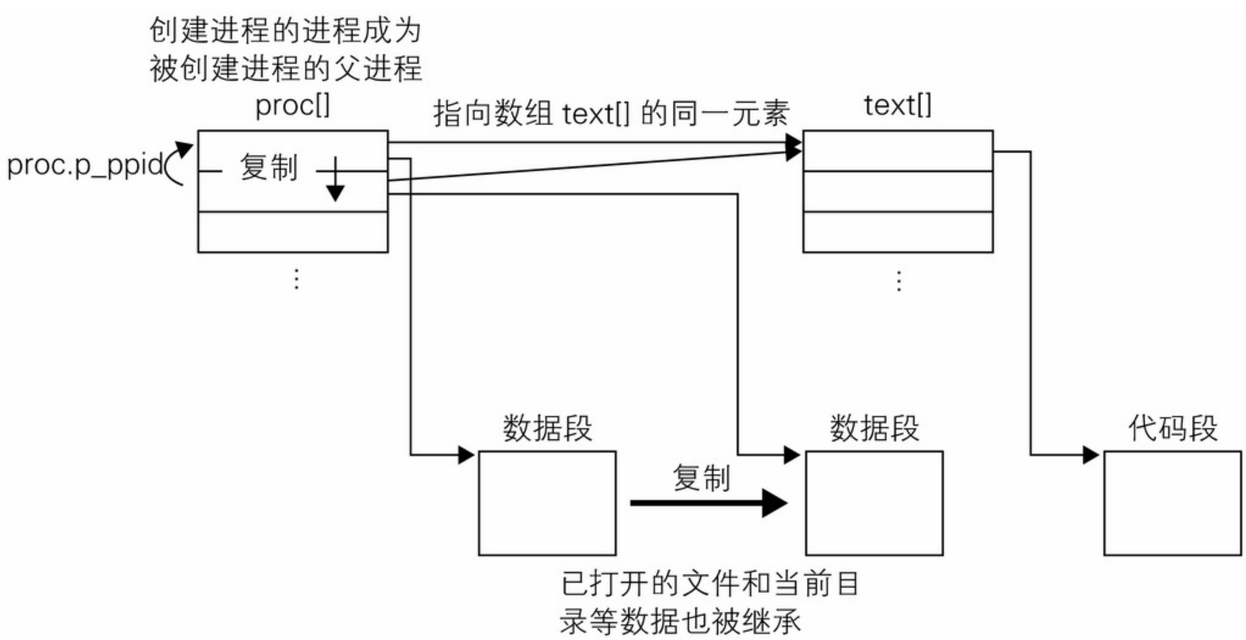


图 **3-2**　进程的生成

- 复制 `proc[]` 数组元素

    - 子进程的 `proc.p_ppid` 指向父进程的 `proc.p_pid` 。

- 复制数据段
    - 复制对象包括 PPDA。子进程继承了已打开的文件和当前目录等数据。
    - 子进程的 `user.p_procp` 指向 `proc[]` 中代表子进程的元素。
    - 代码段不是复制对象。子进程与父进程共享 `text[]` 中的相同元素。`text[]` 用来管理代码段，详细请参考第 4 章。

父进程和子进程

创建进程的进程称为父进程,被其创建的进程称为子进程 。将子进程的 `proc.p_ppid` 设置为父进程的 ID。反之，父进程若想确定自己的子进程，必须遍历 `proc[]` 找到所有 `proc.p_ppid` 指向自己的进程（图 **3-3** ）。

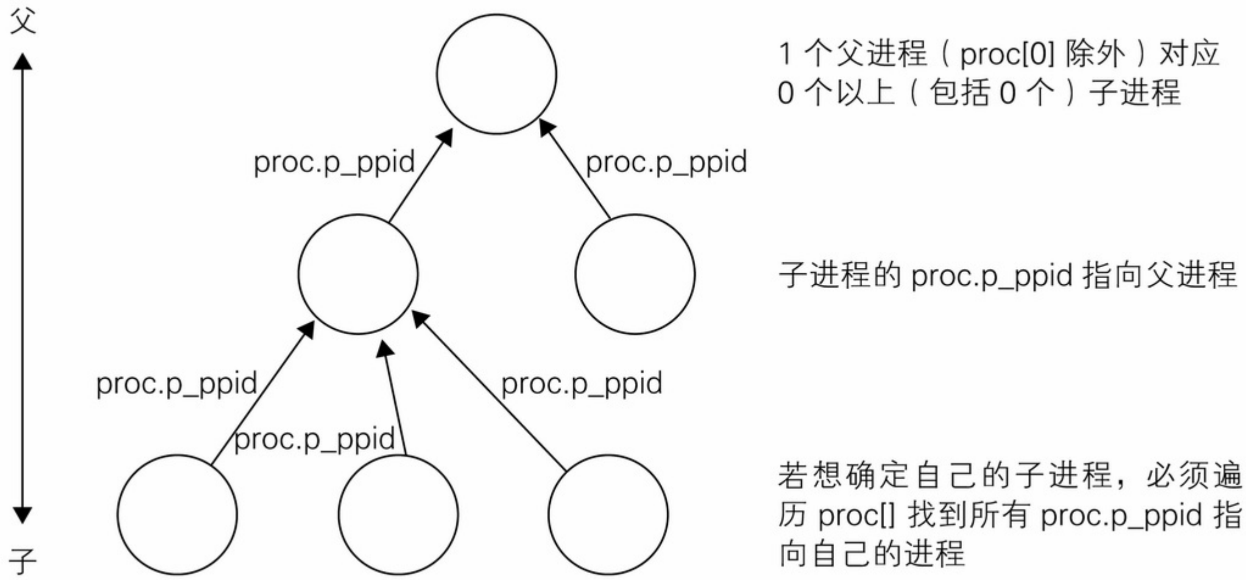


图 **3-3**　父进程和子进程

父进程和子进程具有如下特点。

- 各进程拥有 1 个父进程和 0 个以上（包括 0 个）的子进程
- 父进程在子进程结束时，取其结束状态并释放子进程资源
- 子进程继承了父进程打开的文件和当前目录等数据
- 子进程和父进程共享代码段。但是，如果子进程执行了其他程序，这种共享关系将被解除
- 由于子进程和父进程各自独立，因此他们拥有独立的数据段，`user` 结构体和 `proc[]` 数组元素。在执行时不会相互影响

UNIX V6 提供了使父进程可以介入子进程处理的跟踪机制，以及用于实现父子进程间通信的管道机制。在第 6 章和第 11 章将分别对它们进行介绍。

系统调用 **fork**

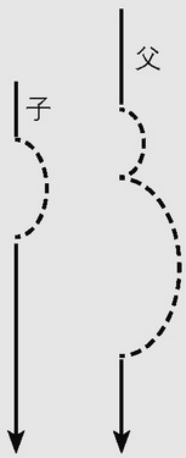
创建新进程时需要使用系统调用 fork。系统调用的详细介绍请参考第 5 章，现在只需要了解通过内核进程处理系统调用即可 。

用 C 语言编写的用户程序在执行 `fork()` 时会首先调用 C 语言的库函数 `fork()` ，在此库函数中再通过执行 `sys fork` 来访问系统调用 fork（代码清单 3-2 ）。

代码清单 **3-2　C** 语言的库函数 **fork() (source/s4/fork.s)**

```
.globl     _fork, cerror, _par_uid

_fork:
    mov    r5,-(sp)
    mov    sp,r5
    sys    fork
        br 1f
    bec    2f
    jmp    cerror
1:
    mov    r0,_par_uid
    clr    r0
2:
    mov    (sp)+,r5
    rts    pc
.bss
_par_uid: .=.+2
```



3　UNIX V6 的 C 编译器将 C 语言的函数转换为起始位置带有下划线“_”的标签。因此，这里的 `_fork:` 就相当于 C 语言的 `fork()` 函数。

`sys` 指令是用来执行系统调用的汇编指令，它的参数是代表执行哪个系统调用的整数。此时，控制权由用户进程交给内核进程，内核的 `fork()` 被执行，使得创建进程的处理也开始执行（代码清单 3-3 ）。

系统调用 `fork` 针对父进程和子进程各返回 1 次 [1] 。父进程和子进程的处理如下所示。

[1] 父进程和子进程此时共用一份代码，父进程从 `fork` 返回处继续运行，而子进程从 `fork` 返回处开始运行。——译者注

代码清单 **3-3　fork() (sys/ken.c)**

```
fork()
{
    register struct proc *p1, *p2;

    p1 = u.u_procp;
    for(p2 = &proc[0]; p2 < &proc[NPROC]; p2++)
        if(p2->p_stat == NULL)
            goto found;
    u.u_error = EAGAIN;
    goto out;

found:
    if(newproc()) {
        u.u_ar0[R0] = p1->p_pid;
        u.u_cstime[0] = 0;
        u.u_cstime[1] = 0;
        u.u_stime = 0;
        u.u_cutime[0] = 0;
```

```
19         u.u_cutime[1] = 0;
20         u.u_utime = 0;
21         return;
22     }
23     u.u_ar0[R0] = p2->p_pid;
24
25 out:
26     u.u_ar0[R7] =+ 2;
27 }
```

父进程的处理

5　将 p1 指向执行进程（父进程）的 proc 结构体。

6~10　从起始位置遍历 proc[] 寻找未使用的元素，找到后将 p2 指向该元素，然后跳转至 found（图 3-4）。

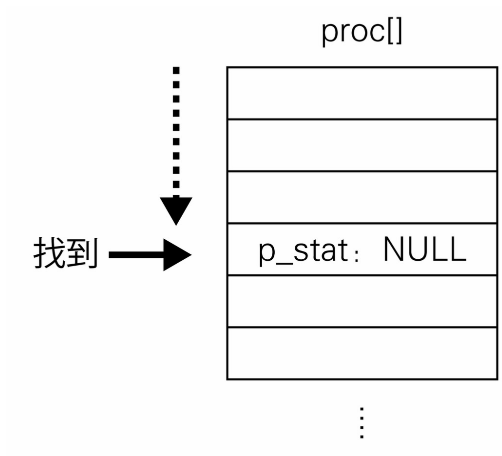


图 **3-4**　寻找未使用的 **roc[]** 数组元素

13　找到未使用的元素后，调用 newproc() 生成新的进程。由于 **newproc()** 向父进程返回 **0**，因此此处 if 条件的值在父进程上为假。

23　将父进程的 **r0** 设为子进程的 **proc.p_pid**，以此作为系统调用 **fork** 对父进程的返回值 。通过 u.u_ar0 可以访问用户进程的寄存器。详细说明请参考第 5 章。

26　将父进程的程序计数器指向下一条指令。

内核的 fork() 处理到此结束，下面来看一下返回到 C 语言库函数 fork() 后的处理（代码清单 3-2 ）。

用户进程的程序计数器已经被修改，此时指向前面给出的汇编代码 fork.s 的第 8 行。

bec 指令用来检查进 / 借位标志（PSW[0]）是否为 0。在系统调用处理中如果出现错误，此标志将被置 1。如果此标志为 0 则跳转到第 13 行，通过 rts 指令返回调用 C 语言库函数 fork() 的位置。返回值为保存在 **r0** 中的子进程的 **ID** 。

子进程的处理

在内核的 fork() 处理中，子进程从 newproc() 返回处开始执行（代码清单 3-3 ）。

13　**newproc()** 对子进程返回**1**，因此if 条件的值在子进程上为真。子进程从newproc() 返回处开

始执行和返回值为 1 的原因将在后文中说明。

14~21　将用户进程的 **r0** 设定为父进程的 **proc.p_pid,** 以此作为系统调用 **fork** 对子进程的返回值。在处理结束前，进程的执行时间被清 0。

内核的 fork() 处理到此结束，下面来看一下返回到 C 语言库函数 fork() 后的处理（代码清单 3-2 ）。

在前面给出的汇编代码 fork.s 里，sys fork 处理结束后将继续执行第 7 行的 br 1f 。br 是无条件分支（转移）指令，1f 表示跳转到下一条标签所在的位置。第 11 行的 mov 指令将父进程的 proc.p_pid 复制到 _par_uid 中，然后将 r0 清 0。调用 C 语言库函数 fork() 的用户程序可以通过 _par_uid 取得父进程的 ID。第 15 行的 rts 指令用来返回调用 C 语言库函数 fork() 的位置，返回值为保存在 **r0** 中的 **0** 。

newproc()

newproc() 是实际创建进程的函数（代码清单 3-4 ）。它负责将 proc[] 中代表执行进程的元素和执行进程的数据段复制到子进程中。代码段不属于复制对象，子进程和执行进程共享相同的代码段。

代码清单 **3-4　newproc() (ken/slp.c)**

```
newproc()
{
    int a1, a2;
    struct proc *p, *up;
    register struct proc *rpp;
    register *rip, n;

    p = NULL;
retry:
    /* 生成ID */
    mpid++;
    if(mpid < 0) {
        mpid = 0;
        goto retry;
    }
    /* 在proc[]中寻找未使用的元素 */
    for(rpp = &proc[0]; rpp < &proc[NPROC]; rpp++) {
        if(rpp->p_stat == NULL && p==NULL)
            p = rpp;
        if (rpp->p_pid==mpid)
            goto retry;
    }
    if ((rpp = p)==NULL)
        panic("no procs");

    /* 生成proc[]元素 */
    rip = u.u_procp;
    up = rip;
    rpp->p_stat = SRUN;
    rpp->p_flag = SLOAD;
    rpp->p_uid = rip->p_uid;
    rpp->p_ttyp = rip->p_ttyp;
    rpp->p_nice = rip->p_nice;
    rpp->p_textp = rip->p_textp;
    rpp->p_pid = mpid;
    rpp->p_ppid = rip->p_pid;
    rpp->p_time = 0;

    /* 将参照计数器加1 */
    for(rip = &u.u_ofile[0]; rip < &u.u_ofile[NOFILE];)
        if((rpp = *rip++) != NULL)
            rpp->f_count++;
    if((rpp=up->p_textp) != NULL) {
        rpp->x_count++;
        rpp->x_ccount++;
    }
```

```
47     u.u_cdir->i_count++;
48
49     /* 复制数据段 */
50     savu(u.u_rsav);
51     rpp = p;
52     u.u_procp = rpp;
53     rip = up;
54     n = rip->p_size;
55     a1 = rip->p_addr;
56     rpp->p_size = n;
57     a2 = malloc(coremap, n);
58     if(a2 == NULL) {
59         rip->p_stat = SIDL;
60         rpp->p_addr = a1;
61         savu(u.u_ssav);
62         xswap(rpp, 0, 0);
63         rpp->p_flag =| SSWAP;
64         rip->p_stat = SRUN;
65     } else {
66         rpp->p_addr = a2;
67         while(n--)
68             copyseg(a1++, a2++);
69     }
70     u.u_procp = rip;
71     return(0);
72 }
```

11~15　mpid 是用来向进程分配 ID 的全局变量（代码清单 3-5）。

代码清单 **3-5**　**mpid (system.h)**

```
1 int    mpid;
```

每次创建一个新的进程，mpid 就会随之加 1。这个 mpid 是分配给子进程的 ID。

mpid 是 int 型的变量，随着值的不断增加，最高位的符号位最终会被置 1，导致出现负数。这时 mpid 会重置为 0 并再次尝试加 1。当时的 C 语言还不存在 unsigned 型的定义。

17~24　从起始位置遍历 proc[] 寻找未使用的元素。如果不存在这样的元素说明 proc[] 的空间已被用尽，此时调用 panic() 停止系统运行。

与此同时，检查 proc[] 的各个元素的成员变量 proc.p_pid，确认是否存在某个进程，其 ID 与即将分配的 ID 相同。如果存在这样的进程则再次生成 ID。

27~28　开始对子进程进行初始设定。从 u.u_procp 取得 proc[] 中代表执行进程（父进程）的元素。

29~37　rpp 中存放了 proc[] 中未被使用的元素。此处对 rpp 进行下述设定。

- 可执行状态（将 proc.p_stat 设定为 SRUN）
- 位于内存中（设置 SLOAD 标志位）
- 分配 ID（将 proc.p_pid 设定为 mpid）
- 执行时间为 0（将 proc.p_time 设定为 0）

- 父进程为执行进程（将 proc.p_ppid 设定为执行进程的 proc.p_pid ）

此外，proc.p_uid 、proc.p_ttyp 、proc.p_nice 和 proc.p_textp 继承了父进程相应的数据。

40~42　由于子进程继承了由父进程打开的文件，因此这些文件的参照计数器都加 1。关于已打开文件的参照计数器，请参见第 10 章。

43~46　因为子进程与父进程指向 text[] 中相同的元素，所以此元素的参照计数器加上 1。关于 text.x_count 和 text.x_ccount 的不同详见第 4 章。

47　由于子进程继承了当前目录的数据，因此 inode[] 中对应此目录的元素的参照计数器加上 1。关于 inode[] 数组元素的说明请参见第 9 章。

50　调用 savu() ，将 r5、r6 的当前值暂存至 user.u_rsav 。savu() 会在下一节中介绍。

51~53　暂时将父进程的 **user.u_procp** 指向 **proc[]** 中代表子进程的元素 。此时复制出来的父进程数据段，其 user.u_procp 将指向 proc[] 中代表子进程的元素，这样等于为子进程构造了一份与父进程完全相同但却属于子进程自身的数据段。至于父进程本身，我们在 rip 中暂存了 proc[] 中代表父进程的元素，用于在第 70 行恢复它的 user.u_procp 。

54~56　将子进程数据段的长度设置为父进程数据段的长度。a1 中保存了父进程数据段的物理地址。

57　调用 malloc() 尝试分配的一块与父进程的数据段相同长度的内存。malloc() 是用 来分配内存或交换空间的函数，如果执行成功则返回被分配区域的物理地址，如果失败则返 回 NULL 。本章的最后将具体说明此函数。

根据 malloc() 执行的成功与否，接下来的处理会有所不同。如果内存分配成功，数据段将被复制到此区域。如果内存没有足够的空间，父进程的数据段会被复制到交换空间（作为子进程的数据段），待数据从交换空间换入内存时再对其分配内存（图 3-5）。

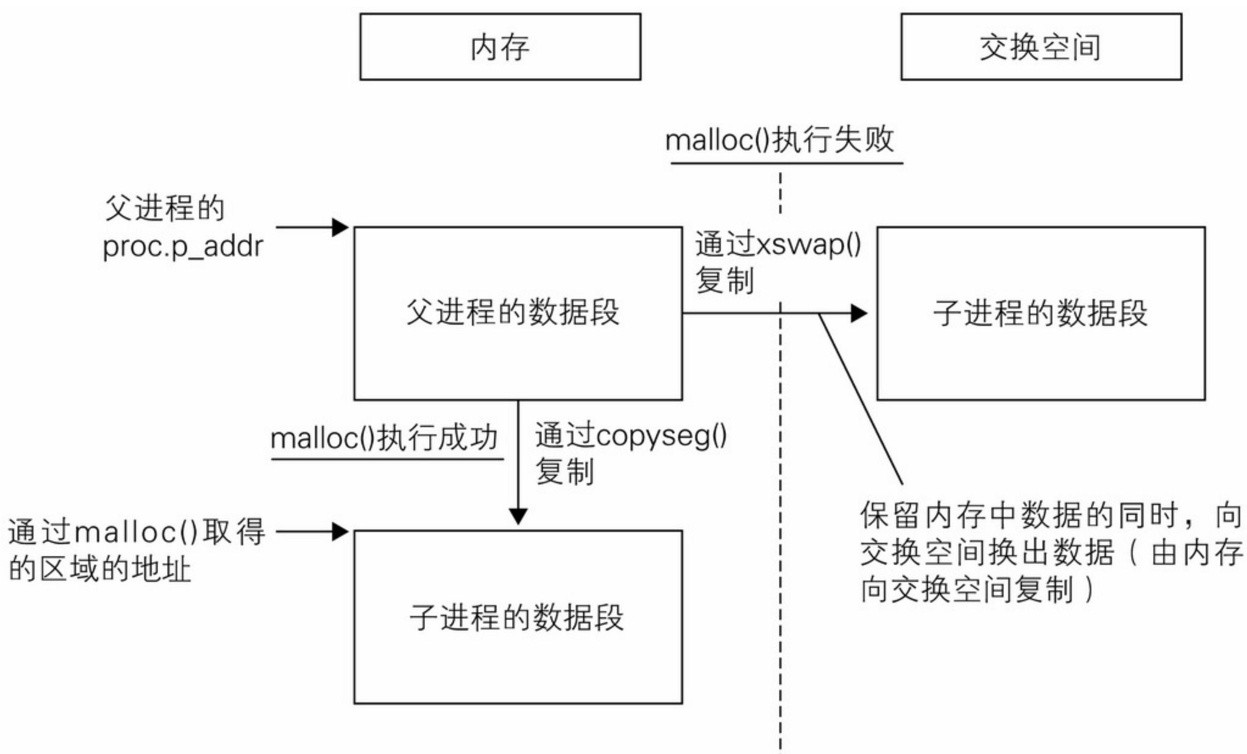


图 **3-5** 数据段的复制

58 以下是 malloc() 执行失败时的处理。

59 将父进程的状态设置为 `SIDL` 。处于 `SIDL` 状态的进程不会被选中成为执行进程，也不会被换出至交换空间。执行第 62 行的 `xswap()` 时，复制父进程的数据段到交换空间的处理被启动。在此处理过程中，父进程将暂时进入休眠状态。将其设置成 `SIDL` 状态是为了防止在复制处理中父进程成为执行进程，或其内存数据被换出导致数据发生变化。

60 将子进程的数据段地址设为与父进程的数据段地址相同。

61 执行 `savu(u.u_ssav)` ，将 r5、r6 的当前值暂存至 `u.u_ssav` 。因为数据段包含 `user` 结构体，所以 `u.u_ssav` 也将被复制到子进程。

62 执行 `xswap()` 将数据从内存换到交换区。由于将 `rpp` 的 `p_addr` 设置为父进程的数据段地址，因此父进程的数据段成为处理对象。由于 `xswap()` 的第二个参数为 0，因此内存中的数据段不会被释放，数据段将从内存复制到交换空间中去。

63 将子进程的 `SSWAP` 标志位置 1。

64 从内存向交换空间的复制处理结束后，将父进程恢复为 `SRUN` 状态。

65 以下是 `malloc()` 执行成功时的处理。

66~68 将子进程的数据段地址设置为通过 `malloc()` 取得的内存区域的物理地址。然后从父进程向子进程复制数据段。

70~71 将父进程的 `user.u_procp` 恢复原状后返回 0。在前面对 **`fork()`** 的说明中也提到过这一点，即 **`newproc()`** 对父进程的返回值为 **0**。

panic()

`panic()` 通过对 `idle()` 进行循环调用以阻止程序继续执行（表 3-1、代码清单 3-6 ）。此函数将在由于资源匮乏等原因导致内核处理无法继续进行的情况下被执行。`idle()` 内部将处理器优先级设置为 0[2]，然后执行汇编指令 `wait` 进入等待中断的状态（代码清单 3-7 ）。

[2] 此处将处理器优先级置 0，使得当前进程可以响应所有优先级的中断。关于中断优先级请参看第 5 章。——审校者注

表 **3-1** **panic()** 的参数

参数	含义
s	错误信息

代码清单 **3-6** **panic() (ken/prf.c)**

```
panic(s)
char *s;
{
    panicstr = s;
    update();
    printf("panic: %s\n", s);
    for(;;)
        idle();
}
```

代码清单 **3-7　idle() (m40.s)**

```
.globl    _idle
_idle:
    mov    PS,-(sp)
    bic    $340,PS
    wait
    mov    (sp)+,PS
    rts    pc
```

3.3　切换执行进程

中断执行进程

执行进程在执行内核函数 `sleep()` 后进入休眠状态并中断当前处理。此后,由于 **`swtch()`** 被调用,执行进程发生切换（图 3-6）。

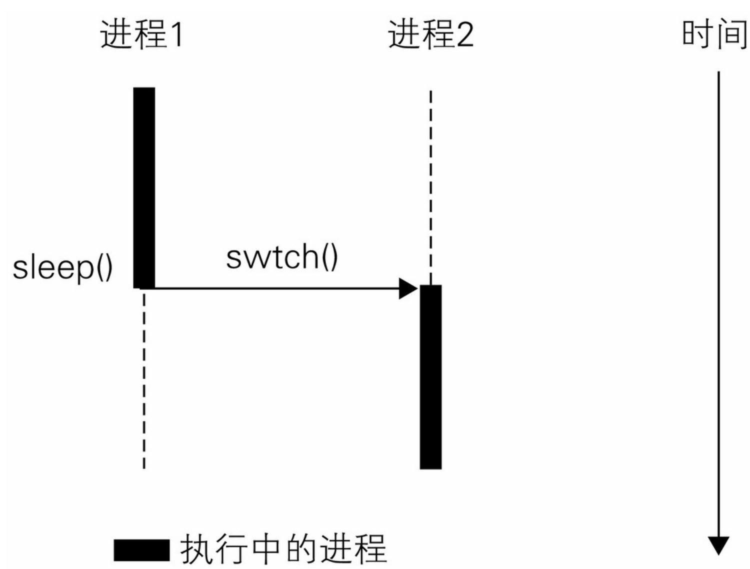


图 **3-6　`sleep()` 和 swtch()**

内核函数 sleep() 一般在下述情况下被调用。

- 用户程序访问了系统调用 `wait`
- 等待周边设备处理完毕
- 等待使用中的资源被释放

只有处于可执行状态的进程才有机会成为执行进程。处于休眠状态的进程除非该状态被解除，否则无法再次被执行。

进程的执行状态

进程的状态由 proc.p_stat 表示。SRUN 表示可执行状态，SSLEEP 和 SWAIT 表示休眠状态。

sleep() 将执行中的进程设置为 SSLEEP 或 SWAIT 状态。wakeup() 将对象进程设置为 SRUN 状态（表 3-2、图 3-7 ）。在之后的说明中会将进入休眠状态的过程用“入睡”来表示，将从休眠状态恢复至执行状态的过程用“被唤醒”来表示。

表 **3-2**　进程的执行状态

状态	含义	函数
SRUN	可执行状态	`wakeup()`
SSLEEP、SWAIT	休眠状态。SSLEEP 和 SWAIT 的区别在于被唤醒后的执行优先级	`sleep()`

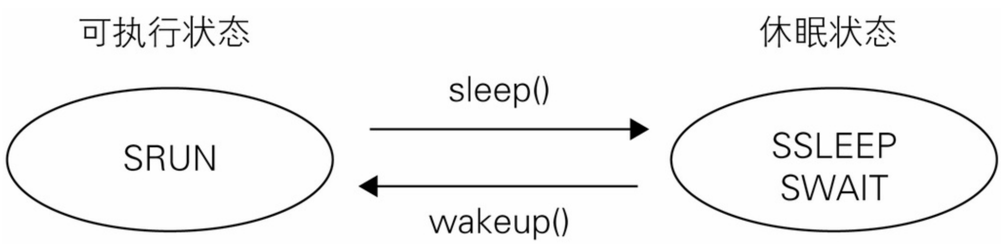


图 **3-7**　状态迁移图

选择执行进程的算法

swtch() 用来选择下一个将被执行的进程。它从起始位置遍历 proc[] ，并将满足下列条件 [3] 的进程选择为执行进程。

> [3] 此处漏了一个条件 ：进程图像在内存中。对于进程图像不在内存中的进程，不能被 swtch() 选择为执行进程，而是经由 0# 进程 sched() 调度进程图像到内存后才能被 swtch() 选择。请参看第 4 章。——审校者注

- 处于可执行状态（SRUN）
- 拥有最高的执行优先级（proc.p_pri 的值最小）

将某个进程明确指定为执行进程是无法做到的。比如，在向子进程转移控制权的时候经常会利用系统调用 wait ，但是如果此时存在执行优先级更高的进程，该进程将被优先执行。

setpri() 函数用来随时调整各进程的执行优先级。占用 CPU 时间越长的进程，其执行优先级将会逐渐降低。由于执行优先级决定了被选择的进程，因此，可以说 **setpri()** 中执行优先级的计算方法对执行进程的选择具有决定性意义 。

setpri() 在下述处理中将被执行。

- 时钟处理（参见第 5 章）
- 信号处理（参见第 6 章）

上下文切换

执行进程被中断时，将当前执行状态保存于 user 结构体中。当被中断的进程再次执行时，通过 user 结构体恢复以前的执行状态。这个处理被称作上下文切换 。

上下文切换处理主要通过以下由汇编语言编写的函数来完成，即 `savu()` 、`retu()` 、`aretu()` 。向 `user` 结构体保存数据的处理由 `savu()` 完成，从 `user` 结构体恢复数据的处理由 `retu()` 和 `aretu()` 完成。

系统调用 **wait**

用户程序执行系统调用 `wait` 后，该进程的执行将被中断，控制权转交给其他进程。

`wait` 具有以下两个功能。中断当前进程等待子进程执行结束，以及清理执行结束的子进程。在 3.5 节中详细介绍 `wait` 。

sleep()

`sleep()` 首先将执行中的进程设置为休眠状态（`SSLEEP` 、`SWAIT` ）（表 3-4、代码清单 3-8 ），然后调用 `swtch()` 切换执行进程。

`sleep()` 接受两个参数：`chan` 和 `pri` 。`chan` 为任意变量的地址，其值将被赋给执行进程的 `proc.p_wchan` 。`wchan` 代表 waiting channel，表示此进程正在等待 **`proc.p_wchan`** 所指向的资源。

`wakeup()` 将进程从休眠状态唤醒至可执行状态。与 `sleep()` 相同，它也接受任意变量地址为参数，然后依次检查各进程的 `proc.p_wchan` ，并将参数一致的进程设置为可执行状态（`SRUN` ）。系统利用 `sleep()` 和 `wakeup()` 实现多个进程等待同一资源时的同步处理（图 3-8 ）。

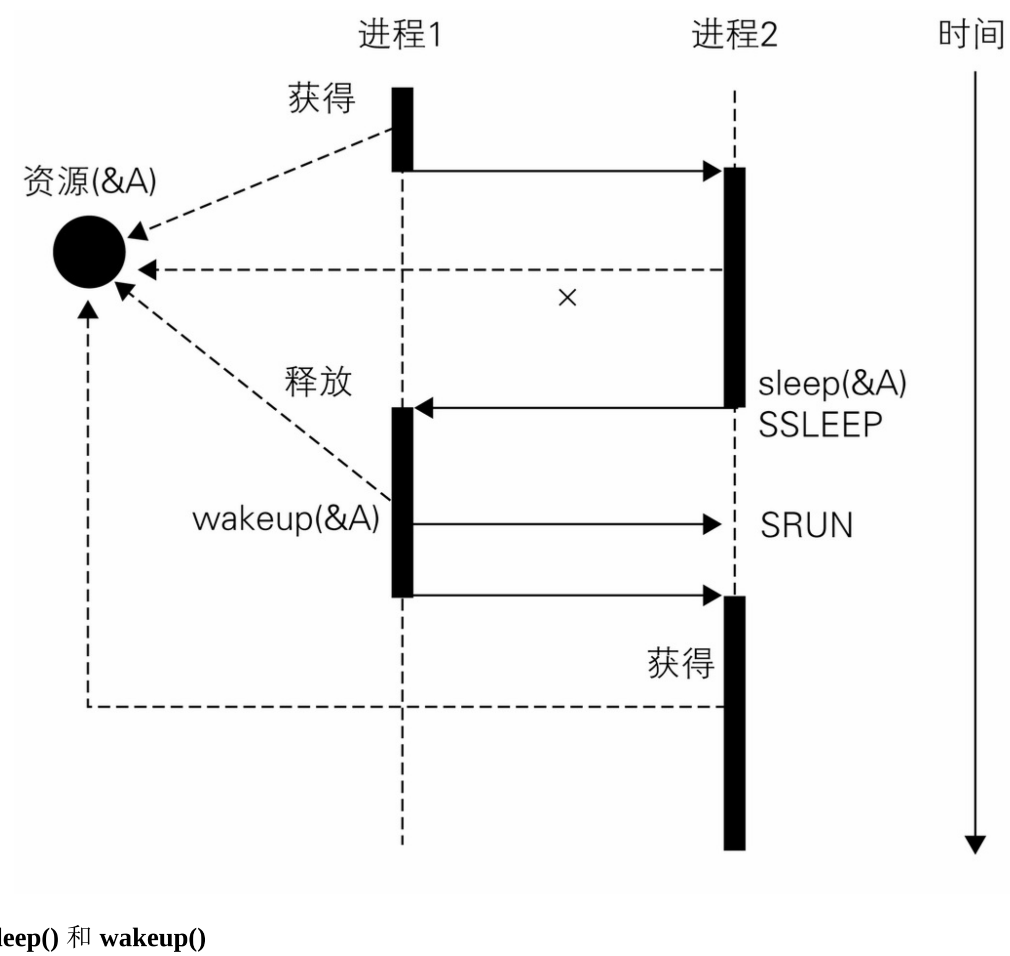


图 **3-8**　**sleep()** 和 **wakeup()**

`pri` 表示执行优先级，将被赋给执行进程的 `proc.p_pri` 。这个值代表进程被唤醒时的执行优先级。

当 pri 的值大于等于 0 时，在执行进程发生切换之前，以及该进程再次被选择为执行进程之后将对信号（参见第 6 章）进行处理，小于 0 时则忽略信号对其不作处理。此外，当 pri 的值大于等于 0 时，进程将被设置为 SWAIT 状态，小于 0 时将被设置为 SSLEEP 状态（表 3-3 ）。这两种不同的状态会对在第 4 章中说明的调度处理产生影响。

表 **3-3　pri**

pri	状态
大于等于 0	设置为 SWAIT 状态。对信号进行处理
小于 0	设置为 SSLEEP 状态。对信号不做处理

表 **3-4　sleep()** 的参数

参数	含义
chan	waiting_channel。所等待资源的地址
pri	执行优先级

代码清单 **3-8　sleep() (ken/slp.c)**

```
sleep(chan, pri)
{
    register *rp, s;

    s = PS->integ;
    rp = u.u_procp;

    if(pri >= 0) {
        if(issig())
            goto psig;
        spl6();
        rp->p_wchan = chan;
        rp->p_stat = SWAIT;
        rp->p_pri = pri;
        spl0();
        if(runin != 0) {
            runin = 0;
            wakeup(&runin);
        }
        swtch();
        if(issig())
            goto psig;
    } else {
        spl6();
        rp->p_wchan = chan;
        rp->p_stat = SSLEEP;
        rp->p_pri = pri;
        spl0();
        swtch();
    }

    PS->integ = s;
    return;
psig:
    aretu(u.u_qsav);
```

```
36 }
```

5 保存 PSW 的当前值，为进程的下次执行做准备。PS 指向 PSW 所映射的地址（代码清单 3-9）。

代码清单 **3-9 PS(param.h)**

```
#define PS 0177776
```

6 将 rp 设置为 proc[] 中代表执行进程的元素。

8~22 如果执行优先级大于等于 0，在入睡和被唤醒时对信号进行处理。首先使用 issig() 判断是否收到信号，如果收到了信号则跳转到 psig 对其进行处理。aretu(u.u_qsav) 的含义在第 6 章进行说明。

设定执行进程的 proc.p_wchan 、proc.p_stat 和 proc.p_pri 。由于在发生中断时被调用的 wakeup() 中也有可能修改这些数据，此处将处理器优先级提升到 6 以防止中断的发生。关于处理器优先级和中断请参见第 5 章的说明。设定结束后再将处理器优先级重置为 0。如果标识变量 runin 为 1（表示不存在需要被换出至交换空间的对象），则启动进程调度器。关于标识变量 runin 和进程调度器请参见第 4 章的说明。最后调用 swtch() 切换执行进程。

23~30 此处为执行优先级小于 0 时的处理。在进行与 11~15 行相同的处理后（但是状态变为 SSLEEP ），调用 swtch() 切换执行进程。

32~33 为了再次执行被中断的进程，恢复在第 5 行被保存的 PSW。

swtch()

swtch() 是用来切换执行进程的函数（代码清单 3-10 ）。它通过遍历 proc[] ，选择处于可执行状态**,**并且执行优先级最高的进程 作为新的执行进程。

内核中有很多处理都是从起始位置遍历 proc[] ，而 swtch() 是从代表当前执行进程的元素位置开始遍历 proc[] 。如果存在 2 个以上具有最高执行优先级的进程，位于对应执行进程的元素后方，并且离它最近的那个元素（进程）将被优先选择（图 3-9 ）。

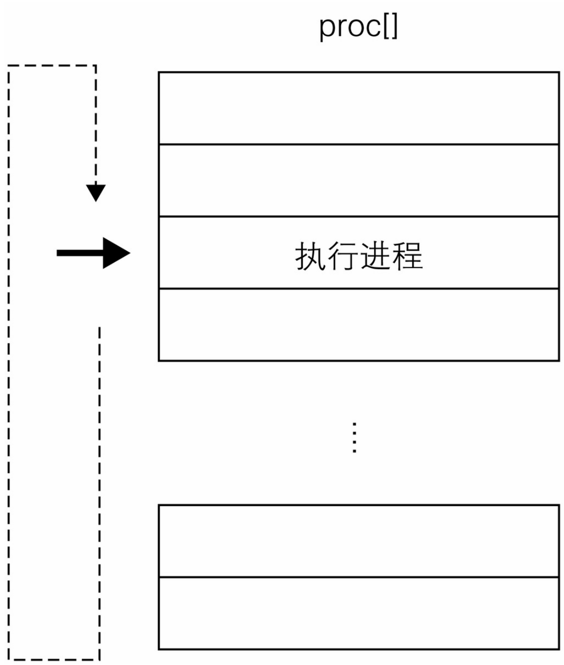


图 **3-9**　以执行进程为起点循环一周

选择进程后，进行下述处理以切换执行进程。

- 修改内核 APR，使得被选择进程的 `user` 结构体可以通过 `u` 进行访问
- 恢复在被选择进程的 `user` 结构体中保存的 r5、r6 的值
- 恢复在被选择进程的 `user` 结构体中保存的用户 APR 的值

`swtch()` 执行结束后，被选择的进程将返回到调用 **`savu()`** 以保存 **r5**、**r6** 的函数自身被调用的位置 [4]，前面提到的**“**利用 **`fork()`** 生成的进程从 **`newproc()`** 返回 **1”**的原因也在于此 [5]。

> [4] 即返回到 `savu()` 的调用者的调用者。——译者注

> [5] 此处逻辑有点跳跃，加以说明：父进程由 `newproc()` 生成子进程后，父进程通过 `newproc()` 的 `return 0` 直接返回，则 `newproc()` 的调用者 `fork()` 获得返回值 0；而子进程通过 `swtch()` 切换上台，`swtch()` 的返回值为 1，返回到 "`savu()` 的调用者的调用者"，即 `fork()`，则 `fork()` 获得返回值 1。——审校者注

代码清单 **3-10　swtch() (ken/slp.c)**

```
swtch()
{
    static struct proc *p;
    register i, n;
    register struct proc *rp;

    if(p == NULL)
        p = &proc[0];

    savu(u.u_rsav);
    retu(proc[0].p_addr);

loop:
    runrun = 0;
    rp = p;
    p = NULL;
```

```
17     n = 128;
18
19     i = NPROC;
20     do {
21         rp++;
22         if(rp >= &proc[NPROC])
23             rp = &proc[0];
24         if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)!=0) {
25             if(rp->p_pri < n) {
26                 p = rp;
27                 n = rp->p_pri;
28             }
29         }
30     } while(--i);
31
32     if(p == NULL) {
33         p = rp;
34         idle();
35         goto loop;
36     }
37     rp = p;
38     curpri = n;
39
40     retu(rp->p_addr);
41     sureg();
42
43     if(rp->p_flag&SSWAP) {
44         rp->p_flag =& ~SSWAP;
45         aretu(u.u_ssav);
46     }
47
48     return(1);
49 }
```

7~8　p 是遍历 proc[] 时所使用的 static 变量。它保存着上次执行 swtch() 时选择的进程（执行进程）所指向的 proc[] 的元素。初次执行 swtch() 时 p 的值为 NULL，指向 proc[] 的起始位置。

10　执行 savu() 将 r5、r6 的当前值保存于待中断进程的 user.u_rsav 之中。等到该进程再次执行时再予以恢复。

11　执行 retu() 切换至调度器进程。proc[0] 是供调度器使用的系统进程，在系统启动时被创建。

14　将标志变量 runrun 设定为 0，该标志变量表示与执行进程相比，存在执行优先级更高的进程（代码清单 3-11）。因为即将切换到拥有最高执行优先级的进程，在此将 runrun 重置为 0。

代码清单 **3-11**　**runrun (system.h)**

```
char    runrun;
```

15~17　初始化工作变量。n 是用来管理进程执行优先级最大值的局部变量。因为最低的执行优先级（proc.p_pri 具有最大值）为 127，在此代入一个更大的值 128 作为初始值。

19~30　遍历 proc[]，寻找满足下述条件的进程。

- 可执行状态（SRUN）
- 存在于内存中（SLOAD）

- 执行优先级最高（proc.p_pri 具有最小值）

如果不存在处于可执行状态的进程，p 保持初始值 NULL 不变。

32~36　如果不存在可执行的进程，将 p 设置为与此前的执行进程相对应的 proc[] 元素。然后调用 idle()，等待由于发生中断而出现的可执行进程。

举例来说，读取块设备的处理结束时将引发中断，使得等待该处理结束的进程被唤醒，因此出现了处于可执行状态的进程。再比如，由时钟引发的中断导致设定在某个时刻启动的进程被唤醒的情况也是如此。中断处理结束后，从 wait 的下一条指令开始继续执行，并从 idle() 返回 swtch()。随后返回到标签 loop 的位置，再次尝试选择执行进程。

37~38　由于执行进程已被决定，将该进程代入 rp，并将该进程的执行优先级保存至 curpri。curpri 为全局变量，用于保存当前被执行进程的执行优先级（代码清单 3-12）。

代码清单 **3-12**　**curpri (system.h)**

```
1 char    curpri;
```

40~41　执行 retu() 恢复被选择进程的 r5、r6，并变更内核 APR 的值使 u 指向 user 结构体。接着执行 sureg() 将保存于被选择进程的 user 结构体中的 APR 的值恢复到硬件的用户 APR，以切换用户空间。至此执行进程的切换过程结束。

43~46　如果 SSWAP 标志位为 1，将其清 0 并从 user.u_ssav 中恢复 r5、r6 的值。之所以这样做，是因为当由于调度器以外的原因导致数据被换出内存时，将 SSWAP 标志位设置为 1。这意味着首先将调用 **xswap()** 进行换出处理,随后执行 **swtch()** 切换执行进程。在 **swtch()** 的第 **10** 行执行 **savu(u.u_rsav)** 时,因为 **user** 结构体已被换出至交换空间,所以此处的 **savu()** 将失去意义。因此，解决办法是在交换处理前首先执行 **savu(u.u_ssav)**,当再次执行进程时再恢复 **user.u_ssav** 中保存的有效值。[1]

48　返回 1。返回位置为执行 **savu()** 的函数（该函数调用 **savu()** 并在 **user** 结构体中保存 **r5**、**r6** 的值）自身被调用的位置。

[1] 这里对savu(u.u _ ssav) 的作用做进一步说明：
1. savu(u.u _ ssav) 用来保存寄存器 r5、r6 的值。之后可以使用 aretu() 来恢复所保存的值，并返回 savu() 的调用者的调用者。
2. swtch() 的第 45 行调用了 aretu()，考虑到在 swtch() 之前执行的 xswap() 进行换出操作，aretu() 所恢复的 r5、r6 的值需要在调用 xswap() 之前通过 savu(u.u _ ssav) 进行保存。
3. 3.2 节给出的 newproc() 的第 62 行执行了 xswap()，并在此之前的第 61 行执行了 savu(u.u _ ssav)。但在 newproc() -> xswap() -> swap() -> sleep() -> 进程下台 -> 经 swtch() 切换该进程再次上台的这个过程中，由于下述两点原因不会引起换出操作。因此此处的 savu(u.u _ ssav) 其实并不需要在 xswap() 之前被执行。
a. 当前进程是生成的子进程，还没有置 SSWAP 标志，不会被换出
b. 父进程被置 SIDL 不能被选择上台
4. 在上述第 3 点中，由于 swtch() 的第 45 行调用了 aretu()，控制权会返回 savu() 的调用者（即 newproc() ） 的调用者。——审校者注

swtch() 的返回位置

函数调用

这部分对“ **swtch()** 的返回位置为执行 **savu()** 的函数自身被调用的位置 ”进行说明。

在进入正题前，首先需要对函数调用时进程栈的行为有所了解。由于使用的编译器等原因，栈的行为会有所不同，此处以示例程序在 UNIX V6 上的运行情况为基础进行说明。

假设有如代码清单 3-13 所示的示例程序。

代码清单 **3-13**　示例程序

```
callee(a, b) {
  int c, d;
  c = a;
  d = b;
  return c + d;
}

caller() {
  int e, f, result;
  e = 1;
  f = 2;
  result = callee(e, f);
  return result;
}
```

在此程序中 `caller` 函数调用了 `callee` 函数。上述代码在 UNIX V6 模拟器上经过编译生成如代码清单 3-14 所示的汇编代码。

代码清单 **3-14**　经过编译的示例程序

```
.globl  _callee
.text
_callee:
~~callee:
~a=4
~b=6
~c=177770
~d=177766
jsr     r5,csv
sub     $4,sp
mov     4(r5),-10(r5)
mov     6(r5),-12(r5)
mov     -10(r5),r0
add     -12(r5),r0
jbr     L1
L1:jmp  cret
.globl  _caller
.text
_caller:
~~caller:
~result=177764
~e=177770
~f=177766
jsr     r5,csv
sub     $6,sp
mov     $1,-10(r5)
mov     $2,-12(r5)
mov     -12(r5),(sp)
mov     -10(r5),-(sp)
jsr     pc,*$_callee
tst     (sp)+
mov     r0,-14(r5)
mov     -14(r5),r0
jbr     L2
L2:jmp  cret
.globl
.data
```

`caller()` 调用 `callee()` 的位置如代码清单 3-15 所示。

代码清单 **3-15**　调用 **callee()** 的位置

```
mov     -12(r5),(sp)
mov     -10(r5),-(sp)
jsr     pc,*$_callee
```

将传递给 `callee()` 的参数压入栈后，通过 `jsr` 指令跳转至 `callee()` 。`jsr` 指令的第一个参数被压入栈。此处被压入栈的 `pc` 的值将成为从 `callee()` 返回的位置。`jsr` 指令的第二个参数为 `pc` 的新值，控制权同时被移交给 `callee()` （表 3-5 ）。

表 **3-5**　控制权被移交给 **callee()** 时栈的状态

sp	值
->	pc（从 `callee()` 返回的位置）
	参数 1
	参数 2

`callee()` 获得控制权后，首先执行 `csv` （代码清单 3-16 ）。

代码清单 **3-16**　**csv**

```
jsr     r5,csv
```

`csv` 函数用于向栈压入 r2、r3 和 r4 的值，采用汇编语言编写（代码清单 3-17 ）。

代码清单 **3-17**　**csv**（**conf/m40.s)**

```
.globl    csv
csv:
    mov    r5,r0
    mov    sp,r5
    mov    r4,-(sp)
    mov    r3,-(sp)
    mov    r2,-(sp)
    jsr    pc,(r0)
```

执行 `csv` 后控制权返回 `callee()` 时的栈状态如表 3-6 所示。

表 **3-6**　执行 **csv** 后的栈状态

sp	值
->	pc(从 `csv` 返回的位置。通常不使用)

	旧的 r2 值
	旧的 r3 值
	旧的 r4 值
r5	旧的 r5 值
	pc (从 `callee()` 返回的位置)
	参数 1
	参数 2

栈中保存了传递给函数的参数，从函数返回时的位置，以及调用函数时 r2、r3、r4、r5 的当前值。r5 的最新值则指向保存在栈中的旧的 r5 值。

在 `callee()` 内使用的局部变量也通过栈进行管理。函数的参数和局部变量通过 r5 进行访问（表 3-7，代码清单 3-18 ）。

代码清单 **3-18**　通过 **r5** 访问 函数的参数和局部变量

```
mov     4(r5),-10(r5)
mov     6(r5),-12(r5)
```

表 **3-7**　局部变量

sp	值-	访问方法
->	局部变量 2	-12(r5)
	局部变量 1	-10(r5)
	pc (从 `csv` 返回的位置)	-8(r5)
	旧的 r2 值	-6(r5)
	旧的 r3 值	-4(r5)
	旧的 r4 值	-2(r5)
r5	旧的 r5 值	(r5)
	pc (从 `callee()` 返回的位置)	2(r5)
	参数 1	4(r5)

	参数 1	4(r5)
	参数 2	6(r5)

随后，`callee()` 的内部处理继续运行，将返回值存放于 r0 之后执行 `cret`（代码清单 3-19 ）。

代码清单 **3-19**　**callee()** 的返回位置

```
mov     -10(r5),r0
add     -12(r5),r0
jbr     L1
L1:jmp  cret
```

`cret` 是恢复保存在栈中的寄存器数据的函数（代码清单 3-20 ）。

代码清单 **3-20**　**cret**（**conf/m40.s)**

```
.globl cret
cret:
    mov    r5,r1
    mov    -(r1),r4
    mov    -(r1),r3
    mov    -(r1),r2
    mov    r5,sp
    mov    (sp)+,r5
    rts    pc
```

恢复保存在栈中的 r5、r4、r3、r2 的值，同时调整 sp 使之指向调用函数时保存的 pc 的值（表 3-8 ）。当 `rts` 指令的参数为 pc 时，将跳转到 sp 所指向的值。

表 **3-8**　**cret** 的 **rts** 指令执行前的栈状态

sp	值
	局部变量 2
	局部变量 1
	pc（从 `csv` 返回的位置）
	旧的 r2 值恢复至 r2
	旧的 r3 值恢复至 r3
	旧的 r4 值恢复至 r4
	旧的 r5 值恢复至 r5

->	pc（从 `callee()` 返回的位置）
	参数 1
	参数 2

至此控制权从 `callee()` 返回 `caller()`，`caller()` 通过 r0 访问 `callee()` 的返回值并继续自身的运行。

此外，栈中保存的位于 sp 所指位置以上的数据不会被删除。这些数据通常不会被再次使用，在使用新的局部变量，或发生函数调用时将被覆盖。

栈上以函数调用为单位进行管理的数据被称为帧，r5 指向与调用者函数相对应的帧内的 r5。通过遍历 **r5,**就可以遍历所有作为调用者的函数**,**因此 **r5** 被称为帧指针或者环境指针 （图 3-10 ）。

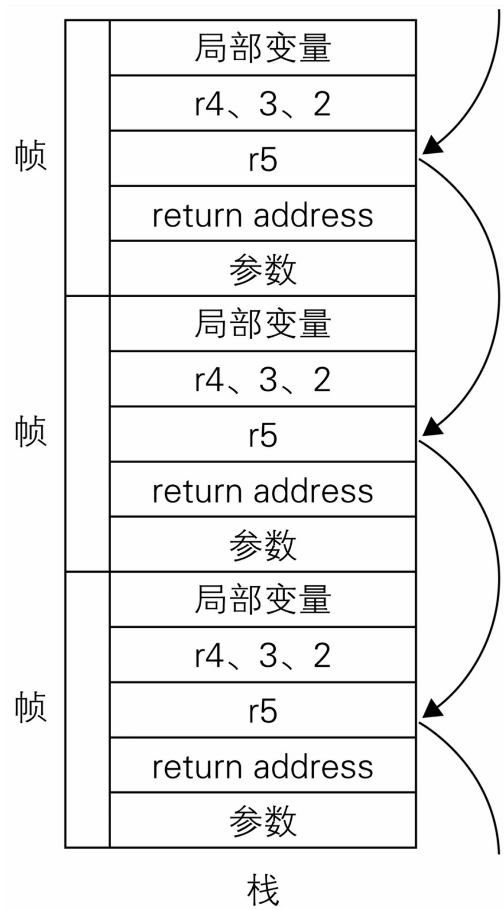


图 **3-10**　帧指针

savu()

根据如上所述的栈行为，再看一下执行 `savu()` （表 3-10，代码清单 3-21 ）时所发生的处理。被中断的进程执行 `savu()` 将 r5 和 r6 的当前值保存于 `user` 结构体内。`savu()` 的参数为 `u.u_rsav[]`，`u.u_ssav[]`，`u.u_qsav[]` 其中之一的地址。r5 和 r6 的当前值将保存于该地址。

`savu()` 获得控制权时的栈状态如表 3-9 所示。

表 **3-9** **savu()** 获得控制权时的栈状态

sp	值
->	pc(从 savu() 返回的位置)
	参数 (u.u_rsav，u.u_ssav，u.u_qsav 其中之一)

表 **3-10** **savu()** 的参数

参数	含义
参数 1	r5 和 r6 的保存位置。为 u.u_rsav 、u.u_ssav 、u.u_qsav 其中之一

代码清单 **3-21** **savu()**（**conf/m40.s)**

```
1 _savu:
2     bis    $340,PS
3     mov    (sp)+,r1
4     mov    (sp),r0
5     mov    sp,(r0)+
6     mov    r5,(r0)+
7     bic    $340,PS
8     jmp    (r1)
```

2 将处理器优先级设置为 7，以防止发生中断。

3~4 将 pc 保存在 r1 中（此处的 pc 指向从 savu() 返回的位置），将参数保存在 r0 中（表 3-11）。

表 **3-11** 保存 **r5**、**r6**

sp	值
	pc -> r1
->	参数 ->r0

5~6 将 r5 和 r6 的当前值保存在作为参数收到的数组 u.u_rsav 、u.u_ssav 或 u.u_qsav 中。

7 将处理器优先级恢复为 0。

8 将 r1 设置为从 savu() 返回的位置，使用 jmp 指令跳转到该位置。

retu()、aretu()

切换进程时将执行 retu()（表 3-13，代码清单 3-22 ）。retu() 操作内核 APR，将 u 指向 user 结构体，随后从 u.u_rsav[] 中恢复 r5 和 r6 的值。

`retu()` 的参数为执行进程的 `proc.p_addr` 。`user` 结构体位于这个地址的起始位置。`retu()` 获得控制权时的栈状态如表 3-12 所示。

表 **3-12　retu()** 获得控制权时的栈状态

sp	值
->	pc（从 `retu()` 返回的位置）
	参数（执行进程的 `proc.p_addr` ）

表 **3-13　retu()** 的参数

参数	含义
参数 1	执行进程的 `proc.p_addr`

代码清单 **3-22　retu()**、**aretu()**（**conf/m40.s)**

```
 1 _aretu:
 2     bis    $340,PS
 3     mov    (sp)+,r1
 4     mov    (sp),r0
 5     br    1f
 6
 7 _retu:
 8     bis    $340,PS
 9     mov    (sp)+,r1
10     mov    (sp),KISA6
11     mov    $_u,r0
12 1:
13     mov     (r0)+,sp
14     mov     (r0)+,r5
15     bic     $340,PS
16     jmp     (r1)
```

8　将处理器优先级设置为 7，以防止发生中断。

9　将返回位置的地址保存在 r1 中。

10　将参数 `proc.p_addr` 保存在 `KISA6` 中。`KISA6` 表示内核 PAR6 的地址（代码清单 3-23）。这样就可以通过 `u` 来访问被选择的进程的 `user` 结构体了。

代码清单 **3-23　KISA6**（**conf/m40.s)**

```
KISA6 =    172354
```

11　将 `user` 结构体的头部保存在 r0 中。`user` 结构体的头部为 `u_rsav` 的定义（代码清单 3-24）。

代码清单 **3-24　user** 结构体的头部

```
struct user
{
    int    u_rsav[2];
```

13~14　将 r6 设置为 `u.u_rsav[0]` 的值，r5 设置为 `u.u_rsav[1]` 的值，以此恢复 r5 和 r6 的数据。

15　将处理器优先级恢复为 0。

16　使用 `jmp` 指令从 `retu()` 返回。

下面来看一下 `aretu()` 的定义（表 3-14）。`aretu()` 的参数为 `u.u_ssav[]` 或 `u.u_qsav[]` 的地址。`aretu()` 从接收到的参数中获取 r5 和 r6 的值并恢复到寄存器中。该函数与 `retu()` 的区别在于不对 APR 进行操作，因此 `u` 指向的 `user` 结构体不会发生切换。

表 **3-14**　**aretu()** 的参数

参数	含义
参数 1	`u.u_ssav` 或 `u.u_qsav` 的地址。从中获取 r5 和 r6 的值用于恢复寄存器

`aretu()` 获得控制权时的栈状态如表 3-15 所示。

表 **3-15**　**aretu()** 获得控制权时的栈状态

sp	值
->	pc（从 `aretu()` 返回的位置）
	参数（`u.u_ssav` 或 `u.u_qsav` 的地址）

2　将处理器优先级设置为 7，以防止发生中断。

3~4　将返回地址保存在 r1，参数保存在 r0 中。之后的处理与 `retu()` 相同，从 `u.u_ssav` 或 `u.u_qsav` 中获取 r5 和 r6 的值并恢复寄存器。

至此，利用 `retu()` 或 `aretu()` 恢复了 r5 和 r6 的值，执行 `savu()` 时的栈状态也得以复原（表 3-16 ）。

表 **3-16**　**retu()**、**aretu()** 执行后的栈状态

sp	值
->	局部变量 2
	局部变量 1
	pc（从 `csv` 返回的位置）

	旧的 r2 值
	旧的 r3 值
	旧的 r4 值
r5	旧的 r5 值
	pc（从调用 `savu()` 的函数返回的位置）
	参数 1
	参数 2

此时执行 `cret` 指令，将恢复栈内保存的 r2~r5 的值，并返回到 `savu()` 的调用者的调用者。

如上所述，在执行**`retu()`** 、**`aretu()`** 后再执行**`return`** （**=`cret`** ）**,** 将从调用 **`savu()`** 的函数中返回。

setpri()

`setpri()` 用于更新进程的执行优先级（`proc.p_pri` ）（表 3-17，代码清单 3-26 ）。在发生时钟中断或处理信号时执行 `setpri()` ，定期更新执行优先级。`setpri()` 的参数为 `proc[]` 的元素，该元素对应更新对象进程。

赋予 `proc.p_pri` 的新值为 `PUSER+(proc.p_cpu/16)+proc.p_nice` ，但最大值为 127。请注意**,** **`proc.p_pri`** 的值越小**,**执行优先级越高。

`PUSER` 的值被定义为 100（代码清单 3-25 ）。

代码清单 **3-25　PUSER**（**param.h)**

```
#define    PUSER    100
```

`proc.p_cpu` 为该进程占有 CPU 的总时间，在进程运行过程中会随着时钟中断而递增。但是，当进程从交换空间换入内存时，该值被重置为 0。**CPU** 的占有时间越长**,**进程的执行优先级越低。

`proc.p_nice` 在用户执行系统调用 `nice` 时被设定，且总为正值。该值用来调低执行优先级。但也有例外，超级用户（参考第 9 章）可以将其设为负值。

表 **3-17　setpri()** 的参数

参数	含义
up	`proc[]` 元素调整该进程的执行优先级

代码清单 **3-26　setpri()**（**ken/slp.c)**

```
setpri(up)
```

```
 2 {
 3     register *pp, p;
 4
 5     pp = up;
 6     p = (pp->p_cpu & 0377)/16;
 7     p =+ PUSER + pp->p_nice;
 8     if(p > 127)
 9         p = 127;
10     if(p > curpri)
11         runrun++;
12     pp->p_pri = p;
13 }
```

6~9　计算执行优先级。

10~11　`curpri` 为当前执行中的进程的执行优先级。标志变量 `runrun` 表示存在执行优先级大于当前进程的其他进程。因为 **`proc.p_pri`** 的值越小执行优先级越高,所以此处的不等号实际上是错误的，这个错误在 UNIX 的下一个版本中已被修正。

wakeup()

`wakeup()` 遍历 `proc[]` 的元素，唤醒因为等待由该函数的参数所指定的资源而处于休眠状态的进程（表 3-18，代码清单 3-27 ）。

表 **3-18**　**wakeup()** 的参数

参数	含义
chan	waiting_channel。进程等待的资源的地址

代码清单 **3-27**　**wakeup()**（**ken/slp.c)**

```
 1 wakeup(chan)
 2 {
 3     register struct proc *p;
 4     register c, i;
 5
 6     c = chan;
 7     p = &proc[0];
 8     i = NPROC;
 9     do {
10         if(p->p_wchan == c) {
11             setrun(p);
12         }
13         p++;
14     } while(--i);
15 }
```

6~14　从起始位置遍历 `proc[]`，对于与 `proc.p_wchan` 参数值相同的进程，可执行 `setrun()` 将其设定为可执行状态。

setrun()

`setrun()` 将由参数指定的进程设定为可执行状态（`SRUN` ）（表 3-19，代码清单 3-28 ）。

表 **3-19　setrun()** 的参数

参数	含义
p	设定对象进程的 proc 结构体

代码清单 **3-28　setrun()**（**ken/slp.c)**

```
1 setrun(p)
2 {
3     register struct proc *rp;
4
5     rp = p;
6     rp->p_wchan = 0;
7     rp->p_stat = SRUN;
8     if(rp->p_pri < curpri)
9         runrun++;
10    if(runout != 0 && (rp->p_flag&SLOAD) == 0) {
11        runout = 0;
12        wakeup(&runout);
13    }
14 }
```

6　由于已从休眠状态被唤醒，因此将 `proc.p_wchan` 重置为 0。

7　将 `proc.p_stat` 置为 `SRUN` 使进程成为可执行状态。

8~9　如果比执行进程的执行优先级高，则设置标志变量 `runrun` 。被唤醒的进程的执行优先级由使之入睡的 `sleep()` 设定。

10~13　标志变量 `runout` 表示当前不存在可由交换空间换入内存的进程。如果已经设置了 `runout`，而被唤醒的进程已位于交换空间，则说明可能允许将其他进程换入内存，因此，此处将标志变量 `runout` 重置为 0，并启动调度器。关于调度器的详细介绍请见第 4 章。

3.4　执行程序

执行程序时，首先需要将该程序的执行文件从块设备读取至内存，并将其配置到执行进程的虚拟地址空间中。

系统调用 `exec` 用于执行程序执行文件的读入处理和分配内存的处理。

系统调用 **`exec`** 不会改变已打开文件和当前目录的数据。父进程的数据按原状保留 。

程序执行文件的格式

系统调用 `exec` 的重要处理之一是读取程序的执行文件。本节将介绍 UNIX V6 程序执行文件（a.out）的格式。

程序执行文件从文件头开始依次由下述部分构成。符号表（symbol table）和重定位比特（relocation bit）在本书中不做介绍。

- 文件头
- 代码（程序的指令）

- 数据（初始值不为 0 的全局变量等）
- 符号表（供调试器等使用）
- 重定位比特

文件头的长度为固定的 16 字节，从起始位置开始依次由表 3-20 所示数据构成。

表 **3-20**　程序执行文件的文件头

字	含义
0	魔术数字（0407、0410、0411 其中之一）
1	代码长度
2	数据长度
3	bss 长度
4	符号表长度
5	条目的起始位置（始终为 0）
6	未使用
7	重定位标志

魔术数字为 **0407** 的程序在存在多个进程时不共享代码段 。程序执行文件的代码和数据一起读取至进程的数据区域（图 3-11 ）。

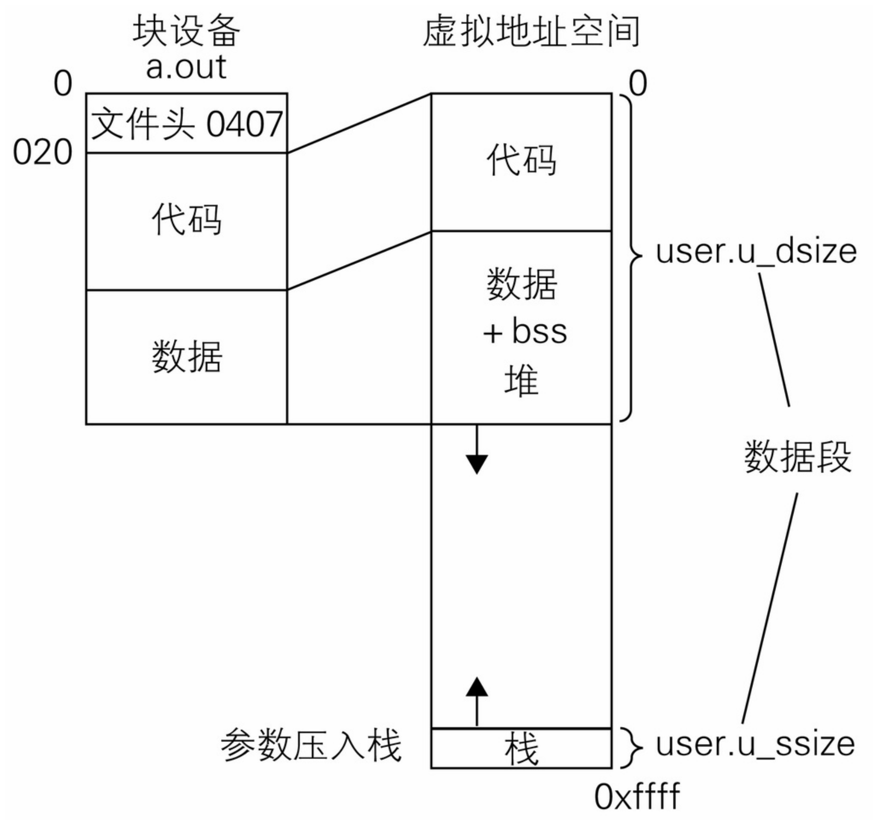


图 **3-11**　魔术数字 **0407**

魔术数字为 **0410** 的程序在存在多个进程时共享代码段 。程序执行文件的代码部分读取到进程的代码段中，数据部分读取到进程的数据区域。读取的数据部分位于代码段之后，起始地址以 8KB 为边界对齐。8KB 恰好等于 APR1 页的长度。代码段和数据区域分别对应不同的 APR，因此在虚拟地址空间里，数据区域的起始地址也需要以 8KB 为边界对齐（图 3-12 ）。

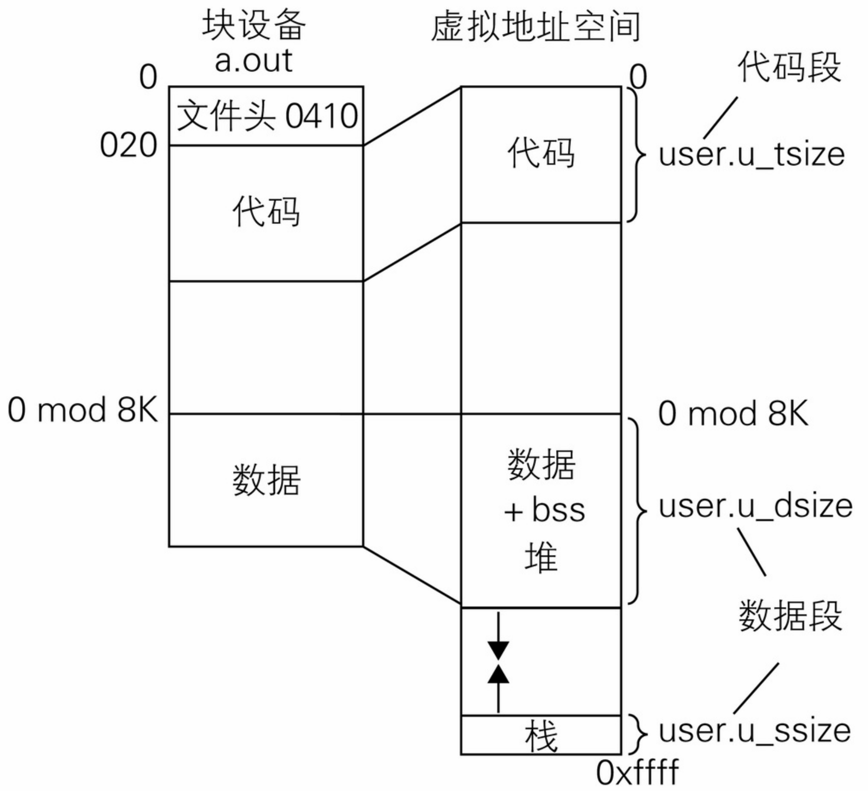


图 **3-12**　魔术数字 **0410**

魔术数字为 0411 的程序在 PDP-11/40 的环境中未投入使用，因此本书中也直接省略。

bss 代表初始值为 0 的数据。因为已知初始值为 0，程序执行文件中并没有为其分配相应的空间。只有当文件被读取到内存时才会分配供 bss 使用的空间。

用户程序可以使用 C 函数库的 `malloc()` 函数来扩展数据区域。

栈区域位于数据段，被赋予地址空间的最高位地址。执行系统调用 `exec` 时将生成长度为 20×64 字节的栈区域。栈区域根据实际情况自动扩展。关于栈区域的扩展将在第 5 章中介绍。栈区域也用于保存传给程序的参数。

系统调用 **exec**

系统调用 `exec` 进行的处理不仅与进程管理相关，还与文件系统、块设备 I/O 等多种要素关系。若想完全理解这一概念，必须了解 UNIX V6 内核整体。建议在阅读本书后复习相关内容。

用户程序调用 `exec` 后将执行下述处理。

- 将传递给执行程序的参数存入缓冲区
- 将程序从块设备读取到内存
- 更新进程的虚拟地址空间
- 将存入缓冲区的参数压入栈
- 对 SUID、SGID（参见第 9 章）进行处理

- 对寄存器和信号进行初始化

exec 的第 1 个参数为一个字符串的起始地址，该字符串代表程序执行文件路径，以 0（NULL ）结尾。第 2 个参数为传递给程序的参数序列的起始地址。参数序列中各参数的起始地址依次排列，以 0 结尾。各个参数的值也同样以 0 结尾。执行 exec 汇编代码的例子如代码清单 3-29 所示。

代码清单 **3-29**　执行系统调用 **exec**

```
sys exec; name; args
...
name: ...\0
...
args: arg0; arg1; ...; \0
arg0: ...\0
arg1: ...\0
```

在更新进程的虚拟地址空间之后，传递给程序的参数压入进程的栈中。从栈区域的底部（高位地址）开始依次是各参数的值、-1、各参数的地址、参数的数量。各参数的值的结尾字符为 0（NULL ）。参数的数量、各参数地址的起始位置分别对应 C 语言程序 main(int argc, char**argv) 的两个参数（图 3-13 ）。

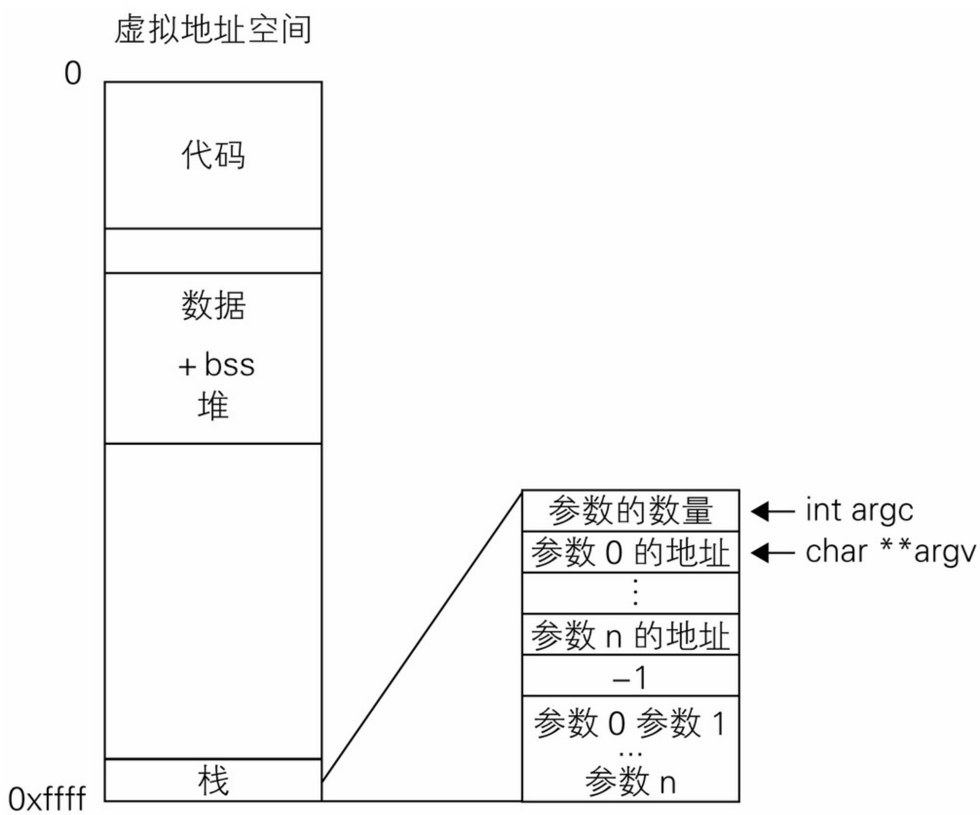


图 **3-13**　传递给程序执行文件的参数

可以同时执行 exec 的进程数量被限制为 NEXEC （=3 ）（代码清单 3-30 ）。

代码清单 **3-30**　**NEXEC**（**param.h)**

```
#define    NEXEC    3
```

向执行程序传递参数、将程序执行文件从块设备读取至内存，这些处理都会大量使用块设备的缓冲区（参见第 7 章）。此外，在等待块设备处理完毕的过程中切换执行进程的可能性很大，考虑到新的执行进程有可能再次执行 exec，从而导致块设备的缓冲区空间不足，因此有必要加以限制。

exec() 为系统调用 exec 的处理函数（表 3-21，代码清单 3-31 ）。向系统调用处理函数传递参数的方法详见见第 5 章。

表 **3-21**　系统调用 **exec** 的参数

参数	含义
u.u_arg[0]	字符串的起始地址，该字符串代表程序执行文件路径
u.u_arg[1]	传递给程序的参数序列的起始地址

代码清单 **3-31**　**exec()**（**ken/sys1.c)**

```
exec()
{
    int ap, na, nc, *bp;
    int ts, ds, sep;
    register c, *ip;
    register char *cp;
    extern uchar;

    /* 正当性检查 */
    ip = namei(&uchar, 0);
    if(ip == NULL)
        return;
    while(execnt >= NEXEC)
        sleep(&execnt, EXPRI);
    execnt++;
    bp = getblk(NODEV);
    if(access(ip, IEXEC) || (ip->i_mode&IFMT)!=0)
        goto bad;

    /* 将参数存入缓冲区 */
    cp = bp->b_addr;
    na = 0;
    nc = 0;
    while(ap = fuword(u.u_arg[1])) {
        na++;
        if(ap == -1)
            goto bad;
        u.u_arg[1] =+ 2;
        for(;;) {
            c = fubyte(ap++);
            if(c == -1)
                goto bad;
            *cp++ = c;
            nc++;
            if(nc > 510) {
                u.u_error = E2BIG;
                goto bad;
            }
            if(c == 0)
                break;
```

```
        }
    }
    if((nc&1) != 0) {
        *cp++ = 0;
        nc++;
    }

    /* 读取程序执行文件的文件头 */
    u.u_base = &u.u_arg[0];
    u.u_count = 8;
    u.u_offset[1] = 0;
    u.u_offset[0] = 0;
    u.u_segflg = 1;
    readi(ip);
    u.u_segflg = 0;
    if(u.u_error)
        goto bad;
    sep = 0;
    if(u.u_arg[0] == 0407) {
        u.u_arg[2] =+ u.u_arg[1];
        u.u_arg[1] = 0;
    } else
    if(u.u_arg[0] == 0411)
        sep++; else
    if(u.u_arg[0] != 0410) {
        u.u_error = ENOEXEC;
        goto bad;
    }
    if(u.u_arg[1]!=0 && (ip->i_flag&ITEXT)==0 && ip->i_count!=1) {
        u.u_error = ETXTBSY;
        goto bad;
    }

    ts = ((u.u_arg[1]+63)>>6) & 01777;
    ds = ((u.u_arg[2]+u.u_arg[3]+63)>>6) & 01777;
    if(estabur(ts, ds, SSIZE, sep))
        goto bad;

    /* 读取程序执行文件的代码部分 */
    u.u_prof[3] = 0;
    xfree();
    expand(USIZE);
    xalloc(ip);
    c = USIZE+ds+SSIZE;
    expand(c);
    while(--c >= USIZE)
        clearseg(u.u_procp->p_addr+c);

    /* 读取程序执行文件的数据部分 */
    estabur(0, ds, 0, 0);
    u.u_base = 0;
    u.u_offset[1] = 020+u.u_arg[1];
    u.u_count = u.u_arg[2];
    readi(ip);

    u.u_tsize = ts;
    u.u_dsize = ds;
    u.u_ssize = SSIZE;
    u.u_sep = sep;
    estabur(u.u_tsize, u.u_dsize, u.u_ssize, u.u_sep);

    /* 将参数压入栈 */
    cp = bp->b_addr;
    ap = -nc - na*2 - 4;
    u.u_ar0[R6] = ap;
    suword(ap, na);
    c = -nc;
    while(na--) {
        suword(ap=+2, c);
        do
            subyte(c++, *cp);
        while(*cp++);
```

```
113     }
114     suword(ap+2, -1);
115
116     /* 设置SUID、SGID */
117     if ((u.u_procp->p_flag&STRC)==0) {
118         if(ip->i_mode&ISUID)
119             if(u.u_uid != 0) {
120                 u.u_uid = ip->i_uid;
121             u.u_procp->p_uid = ip->i_uid;
122         }
123     if(ip->i_mode&ISGID)
124         u.u_gid = ip->i_gid;
125 }
126
127     /* 清空信号和寄存器 */
128     c = ip;
129     for(ip = &u.u_signal[0]; ip < &u.u_signal[NSIG]; ip++)
130         if((*ip & 1) == 0)
131             *ip = 0;
132     for(cp = &regloc[0]; cp < &regloc[6];)
133         u.u_ar0[*cp++] = 0;
134     u.u_ar0[R7] = 0;
135     for(ip = &u.u_fsav[0]; ip < &u.u_fsav[25];)
136         *ip++ = 0;
137     ip = c;
138
139 bad:
140     iput(ip);
141     brelse(bp);
142     if(execnt >= NEXEC)
143         wakeup(&execnt);
144     execnt--;
145 }
```

10~12　通过用户作为参数指定的程序执行文件的路径，取得 inode[] 中对应的该文件的元素。如果用户指定了不存在的文件路径，或是不具备访问该文件的权限时，namei() 将返回 NULL 。

13~15　确认正在执行的 exec() 的进程数小于 NEXEC 。如果大于或等于 NEXEC 则调用 sleep() 进入睡眠状态，直到正在执行的 exec() 的进程数小于或等于 NEXEC 。通过此处的检查后递增 execnt 的值，表示正在执行 exec() 的进程数又增加了 1 个。execnt 为全局变量（代码清单 3-32）。

代码清单 **3-32**　**execnt**（**system.h)**

```
int    execnt;
```

16　执行 getblk() 取得尚未被分配给其他块设备的块设备缓冲区，并将其作为处理参数时的工作内存。大概开发人员认为与通过 malloc() 和 mfree() 分配内存相比，这种做法更简单。

17~18　执行 access() 检查程序文件的执行权限，同时确认该文件不是特殊文件。

21~23　将传递给程序的参数存入刚才取得的缓冲区。首先将 cp 设定为缓冲区数据区域的地址，然后将 na （参数的数量）和 nc （参数的总字节数）清 0。

24　依次处理用户传递给程序的参数。fuword() 是从上一模式所示的虚拟地址空间向当前模式所示的虚拟地址空间复制的 1 个字长的数据。u.u_arg[1] 此时为系统调用 exec 的 第 2 个参数的地址。

25　每处理 1 个参数，na 都会加 1。当这个 while 处理结束时，na 的值等于参数的数量。

26~27　fuword() 处理失败时返回 -1，此时跳转到 bad 进行错误处理。

28　将 u.u_arg[1] 增加 1 个字以便访问下一个参数。

29~41　将参数以字节为单位依次存入缓冲区，将 ap 设定为指向该参数的指针。fubyte() 与 fuword() 相似，但是处理单位为 1 个字节。参数以字节为单位存入缓冲区。fubyte() 发生错误时返回 -1，此时跳转到 bad 进行错误处理。

每处理 1 个字节，nc 都会加 1。当这个 while 处理结束时，nc 的值等于参数的总字节数。因为缓冲区的长度只有 512 字节，当 nc 大于 510 时将认为参数的长度过长，此时跳转到 bad 进 行错误处理。

由于各参数都以 0（NULL ）结尾，因此当 fubyte() 取得的值为 0 时，转而处理下一个参数。当存入缓冲区的处理结束后，缓冲区中保存的数据形式为“参数 0 参数 1... 参数 n”。

43~46　当 nc 的值为奇数时，向缓冲区中追加 1 个字节使数据长度成为偶数。因为系统希望以字为单位进行处理。

49~53　读取程序文件的文件头。readi() 是用于读取文件内容的函数，其参数由 user 结构体指定（表 3-22）。

表 **3-22**　**readi()** 的参数

参数	含义	值
u.u_base	用于保存读取数据的地址	&u.u_arg[0]
u.u_count	读写长度	8 字节（4 个字）
u.u_offset	读写偏移量	0
u.u_segflag	读取到内核空间（1），或是读取到用户空间（0）	1

56~57　如果 readi() 在执行中发生错误，将错误代码保存至 u.u_error ，并跳转到 bad 进行错误处理。

通过 readi() 读取的数据，以表 3-23 所示的形式保存于 u.u_arg[] 中。

表 **3-23**　**readi()** 的执行结果

u.u_arg[n]	值
u.u_arg[0]	魔术数字
u.u_arg[1]	代码长度
u.u_arg[2]	数据长度
u.u_arg[3]	bss 长度

58　将 sep 设定为 0。

59~68　根据魔术数字进行分支处理。魔术数字为 0407 时不使用代码段，而是将代码和数据一并读取至数据区域。将 u.u_arg[2] 设置为代码长度和数据长度之和，将 u.u_arg[1] 清 0。

如果魔术数字为 0407、0410、0411 以外的值则认为不是程序的执行文件，并跳转到 bad 进行错误处理。

69~72　如果代码的长度（u.u_arg[1] ）不为 0，该程序文件又被作为数据文件打开，则进行错误处理。

74~75　计算代码段和数据区域的长度。段的长度以 64 字节为单位进行管理。

76~77　执行 estabur() 更新用户 APR 和用户空间 [6] 。SSIZE 表示栈区域的初始长度，被定义为 20（×64 字节）（代码清单 3-33）。

[6] 此处调用 estabur() 主要是通过设置 APR 来检查各个段长度是否合法，之后会再次调用 estabur() 正式设置 APR。——审校者注

代码清单 **3-33**　**SSIZE（param.h)**

```
1 #define    SSIZE    20
```

80　从此处开始进行代码段和数据段的初始化处理。首先将用于统计的变量清 0。

81　执行 xfree() 释放当前使用的代码段。如果在此之前依次执行了 fork 和 exec ，那么到此为止就终于告别了父进程所使用的代码段。

82　执行 expand() 将数据段缩小为与 user 结构体相同的长度。

83　执行 xalloc() 分配代码段使用的内存。如果未使用代码段，则在 xalloc() 中不做任何处理。

84~87　执行 expand() 确保数据区域。由于位于 user 结构体之后的区域留有过去的数据，因此需要将其清 0。

90　执行 estabur() ，将进程的数据区域的起始位置变更至虚拟地址为 0 的位置 [7] 。

91~94　执行 readi() ，将程序读取至进程的数据区域。魔术数字为 0407 时读取代码和数据，魔术数字为 0410 时只读取数据。偏移量 020 表示程序文件头的长度（表 3-24）。

[7] 此处调用 estabur() 是将数据段的虚拟地址设为 0（物理地址并没有变），原因是之后调用 readi() 的参数要求读取进程数据区到虚拟地址为 0 的位置。也可以将数据段物理地址映射到其他虚拟地址上，然后将这一虚拟地址作为参数传给 readi() 。不过用 0 更为方便。——审校者注

表 **3-24**　**readi()** 的参数

参数	值
u.u_base	0。将程序读取至虚拟地址空间地址为 0 的位置
u.u_offset	0407 时为代码的起始位置，0410 时为数据的起始位置
u.u_count	0407 时为代码和数据长度之和，0410 时为代码的长度

u.u_segflag	0

96~100　执行 estabur() 最终确定用户空间。魔术数字为 0410 时，通过 estabur() 设定用户 APR，对虚拟地址空间进行如下变更：将代码段起始地址调整至虚拟地址为 0 处，数据区域起始地址以 8KB 为边界对齐。

103　从这里开始将保存于缓冲区的参数转存至栈区域。首先将 cp 设定为缓冲区的数据区域的地址。

104~105　计算栈指针的地址，并设定用户进程的 sp。请注意，因为栈区域的起始地址为 0xffff，换算成 int 型将为负数（图 3-14）。

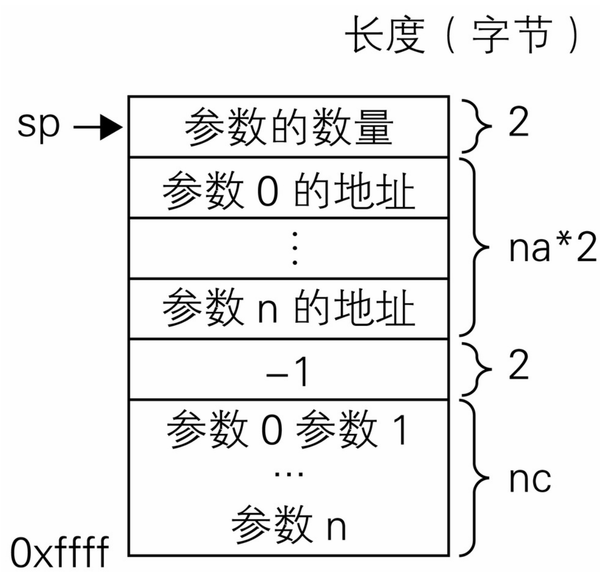


图 **3-14**　计算栈指针的地址

106　将 na（参数的数量）的值设定至栈指针指向的用户空间地址。suword() 是从当前模式的虚拟地址空间向前一模式的虚拟地址空间复制的数据。

107　将 c 设定为保存参数的用户空间地址。

108~113　将参数及地址存入栈区域。

114　将参数地址的下一地址设置为 -1。

117　如果没有处于跟踪状态（详见第 6 章），则从此处开始设置 SUID 和 SGID。

118~122　如果 inode[] 中代表该程序文件的元素的标志位 ISUID 已被设置，且当前用户并非超级用户（u.u_uid 不为 0），则将该元素的成员 inode.i_uid 设定至 u.u_uid。

123~124　如果 inode[] 中代表该程序文件的元素的标志位 ISGID 已被设置，则将该元素的成员 inode.i_gid 设定至 u.u_gid。

128　从此处开始对信号和寄存器进行初始化。

129~131　如果 u.u_signal[n] 的值为偶数则清 0，如果为奇数则保留原有数据。如果在此之前依次执行了 fork 和 exec，u.u_signal[n] 的值则为从父进程继承的数据。关于 u.u_signal[n] 的数据

含义将在第 6 章中说明。

132~134　将用户进程的 r0~r5、r7 的值清 0。作为栈指针的 r6 已在第 105 行设定完毕。

135~136　将进行浮动小数点运算的寄存器清 0。此处理与 PDP-11/40 无关。

140　将 `inode[]` 中代表该程序执行文件的元素的参照计数器减 1。

141　释放用于参数处理的缓冲区。

142~145　如果正在执行的 `exec()` 的进程数小于或等于 `NEXEC`，则唤醒其他等待进入 `exec()` 处理的进程。同时递减 `execnt`，表示正在执行的 `exec()` 的进程数减少了 1 个。

estabur()

`estabur()` 用于更新 `user` 结构体中的用户 APR 数据 `user.u_uisa[]` 和 `user.u_uisd[]`（表 3-25、代码清单 3-34），并在最后执行 `sureg()`，将更新的数据反映到硬件的用户 APR，更新用户空间。

`estabur()` 接受 4 个参数。PPDA 和数据区域的总长度（`nd`）与栈区域的长度（`ns`）之和为数据段的长度。由于 PDP-11/40 对代码段和数据段采用相同的 APR 进行管理（`sep` 为 0），因此本书对 `sep` 为 1 时的处理不进行说明。

`estabur()` 对与用户 PAR、PDR 相对应的 `user.u_uisa[]`、`user.u_uisd[]` 进行设定。首先以 128×64 字节（8KB）为单位分配供代码段使用的内存，余数作为最后一页分配给代码段，并将 PDR 设定为只读。然后以 128×64 字节为单位分配供数据区域使用的内存，在数据区域起始位置的 PAR 附加与 `user` 结构体相同长度的内存，余数作为最后一页分配给数据区域，并将 PDR 设定为允许读写。最后分配供栈使用的页。由于栈向低位地址方向扩展，因此从最高位的页开始向低位进行分配，余数作为最后一页分配给栈区域。除了将 PDR 设定为允许读写之外，还设置 `ED` 标志位表示向低位方向扩展。

对代码段和数据段的 PAR，分别设定从 0 开始的相对值。当代码段、数据区域、栈区域的长度均为 192×64 字节时，用户 APR 的状态如图 3-15 所示。

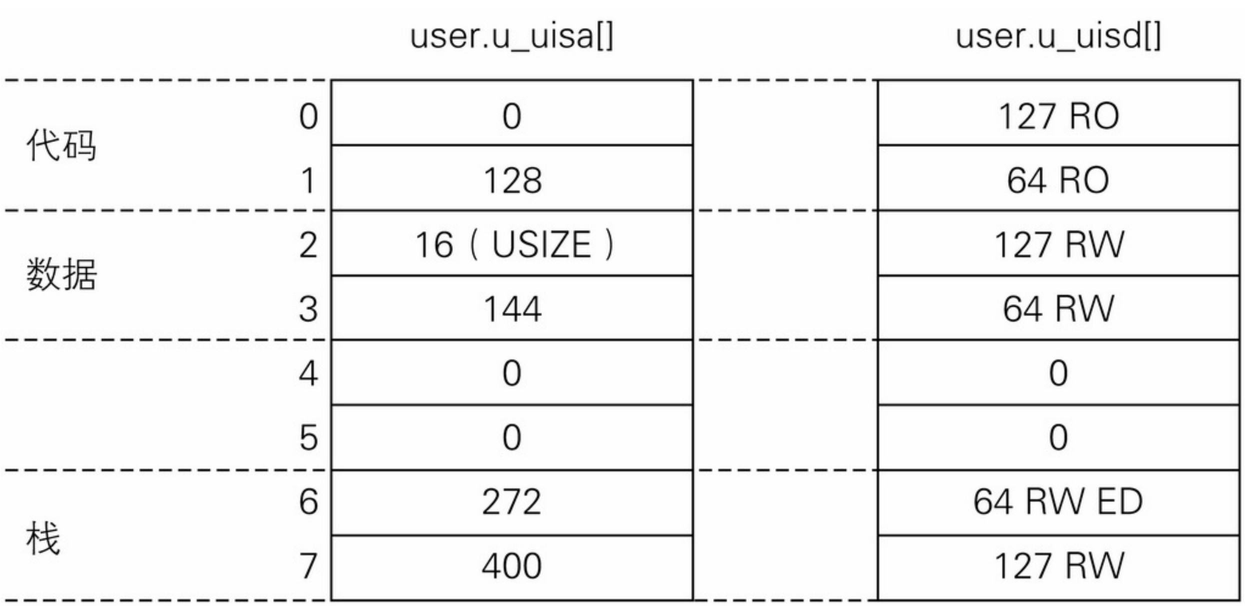


图 **3-15**　由 **estabur()** 设定的用户 **APR** 的示例

表 **3-25**　**estabur()** 的参数

参数	含义
nt	代码段的长度（单位为 64 字节）
nd	PPDA 和数据区域的长度之和（单位为 64 字节）
ns	栈区域的长度（单位为 64 字节）
sep	是否对代码段和数据段的 APR 进行分别管理。PDP-11/40 的环境下为 0

代码清单 **3-34　estabur()**（**ken/main.c)**

```
estabur(nt, nd, ns, sep）
{
    register a, *ap, *dp;

    /* 正当性检查 */
    if(sep) {
        if(cputype == 40)
            goto err;
        if(nseg(nt) > 8 || nseg(nd)+nseg(ns) > 8)
            goto err;
    } else
        if(nseg(nt)+nseg(nd)+nseg(ns) > 8)
            goto err;
    if(nt+nd+ns+USIZE > maxmem)
        goto err;

    /* 分配代码段 */
    a = 0;
    ap = &u.u_uisa[0];
    dp = &u.u_uisd[0];
    while(nt >= 128) {
        *dp++ = (127<<8) | RO;
        *ap++ = a;
        a =+ 128;
        nt =- 128;
    }
    if(nt) {
        *dp++ = ((nt-1)<<8) | RO;
        *ap++ = a;
    }
    if(sep)
    while(ap < &u.u_uisa[8]) {
        *ap++ = 0;
        *dp++ = 0;
    }

    /* 分配数据区域 */
    a = USIZE;
    while(nd >= 128) {
        *dp++ = (127<<8) | RW;
        *ap++ = a;
        a =+ 128;
        nd =- 128;
    }
    if(nd) {
        *dp++ = ((nd-1)<<8) | RW;
        *ap++ = a;
        a =+ nd;
    }
    while(ap < &u.u_uisa[8]) {
        *dp++ = 0;
```

```
52          *ap++ = 0;
53      }
54      if(sep)
55      while(ap < &u.u_uisa[16]) {
56          *dp++ = 0;
57          *ap++ = 0;
58      }
59
60      /* 分配栈区域 */
61      a =+ ns;
62      while(ns >= 128) {
63          a =- 128;
64          ns =- 128;
65          *--dp = (127<<8) | RW;
66          *--ap = a;
67      }
68      if(ns) {
69          *--dp = ((128-ns)<<8) | RW | ED;
70          *--ap = a-128;
71      }
72      if(!sep) {
73          ap = &u.u_uisa[0];
74          dp = &u.u_uisa[8];
75          while(ap < &u.u_uisa[8])
76              *dp++ = *ap++;
77          ap = &u.u_uisd[0];
78          dp = &u.u_uisd[8];
79          while(ap < &u.u_uisd[8])
80              *dp++ = *ap++;
81      }
82      sureg();
83      return(0);
84
85  err:
86      u.u_error = ENOMEM;
87      return(-1);
88  }
```

12~13　`nseg()` 是将以 64 字节为单位的块数转换为页数的函数。如果总页数大于 8 则出错。

14~15　如果需要的块数（以 64 字节为单位）大于能够使用的内存容量上限则出错。`maxmem` 表示能够使用的物理内存容量上限，在系统启动时设定。

18~30　对代码段使用的 `user` 结构体中的 APR 进行设定。

38~53　对数据区域使用的 `user` 结构体中的 APR 进行设定。因为数据段起始位置的 16×64 字节被分配给 PPDA，因此需要将 PAR 与 `USIZE`（16）相加。当数据区域分配完毕时，到 APR6 为止的区域被清 0。

61~71　分配栈区域使用的页。

82　执行 `sureg()`，将 `user` 结构体中的 APR 反映到硬件的用户 APR，以更新用户空间。

83　`estabur()` 执行成功时返回 0。

sureg()

`sureg()` 将执行进程的 `user` 结构体中保存的 APR 数据反映到硬件的用户 APR，以更新用户空间（代码清单 3-35）。`sureg()` 除了在 `estabur()` 中更新 `user` 结构体的 APR 数据时被调用外，在切换执行进程时也会被调用。

因为 `user` 结构体中保存的 APR 数据采用相对地址，在转存至硬件的用户 APR 时，需要根据分配给进程的内存区域的物理地址进行补正。代码段需要将 `text.x_caddr` （参照第 4 章）、数据段需要将 `proc.p_addr` 分别与补正值相加（图 3-16 ）。

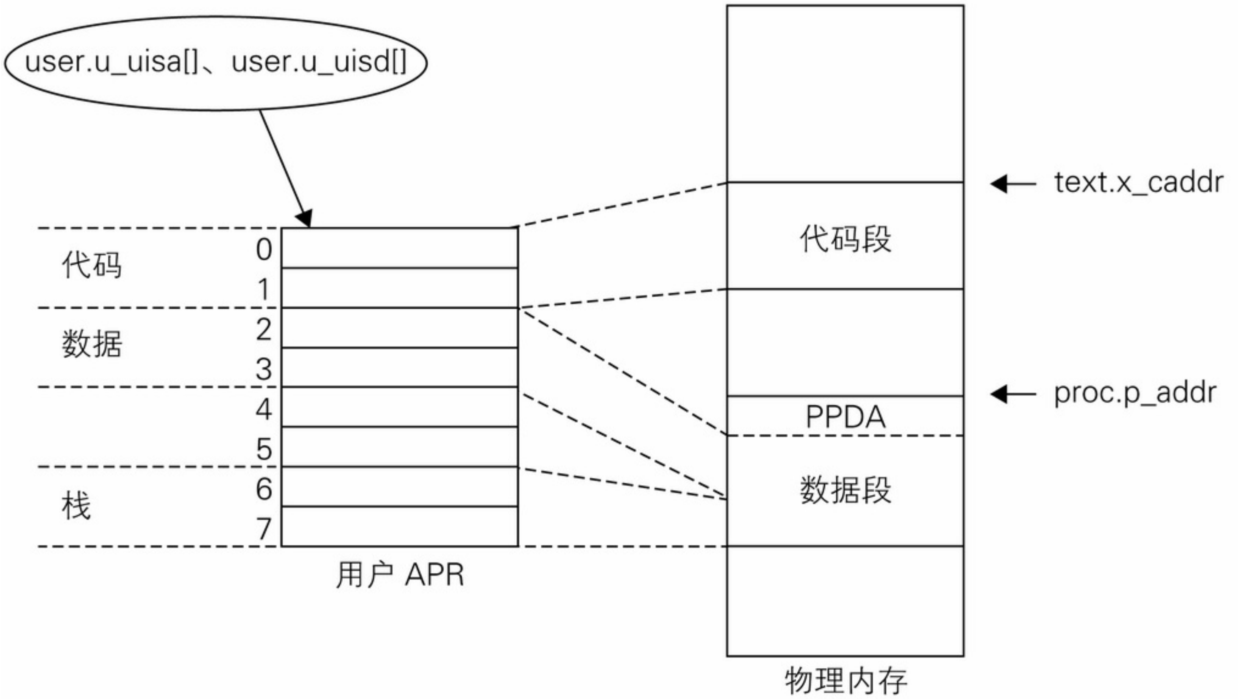


图 **3-16　sureg()**

代码清单 **3-35　sureg()**（**ken/main.c)**

```
sureg()
{
    register *up, *rp, a;

    /* 设定PAR */
    a = u.u_procp->p_addr;
    up = &u.u_uisa[16];
    rp = &UISA->r[16];
    if(cputype == 40) {
        up =- 8;
        rp =- 8;
    }

    while(rp > &UISA->r[0])
        *--rp = *--up + a;
    if((up=u.u_procp->p_textp) != NULL)
        a =- up->x_caddr;

    /* 设定PDR */
    up = &u.u_uisd[16];
    rp = &UISD->r[16];
    if(cputype == 40) {
        up =- 8;
        rp =- 8;
    }
    while(rp > &UISD->r[0]) {
        *--rp = *--up;
        if((*rp & WO) == 0)
            rp[(UISA-UISD)/2] =- a;
    }
```

```
31 }
```

6~15　根据数据段的物理地址对 `user` 结构体的 APR 数据进行补正，并用补正后的值更新用户 PAR。`UISA` 为用户 PAR0 的地址（代码清单 3-36）。

代码清单 **3-36　UISA**（**seg.h)**

```
#define    UISA    0177640
```

16~17　如果执行进程使用代码段，则根据代码段的物理地址确定补正值，并将其赋予 `a` 。

20~30　使用 `user` 结构体的 APR 数据更新 PDR。在设定读取专用代码段的 PDR 时，将对应的 PAR 寄存器的值与第 17 行设定的补正值相加。`UDSA` 为用户 PDR0 的地址（代码清单 3-37）。

代码清单 **3-37　UDSA**（**seg.h)**

```
#define    UDSA    0177660
```

expand()

`expand()` 用来扩展或缩小数据段（表 3-26，代码清单 3-38 ）。缩小数据段时调用 `mfree()` 释放不需要的内存。扩展数据段时调用 `malloc()` 重新分配与数据段全体长度相同的内存，并将其指向用户空间（图 3-17 ）。如果出现内存不足的情况，则将进程从内存换出至交换空间，直到可以分配足够的内存（图 3-18 ）。`expand()` 接受为数据段指定的新的长度作为参数。

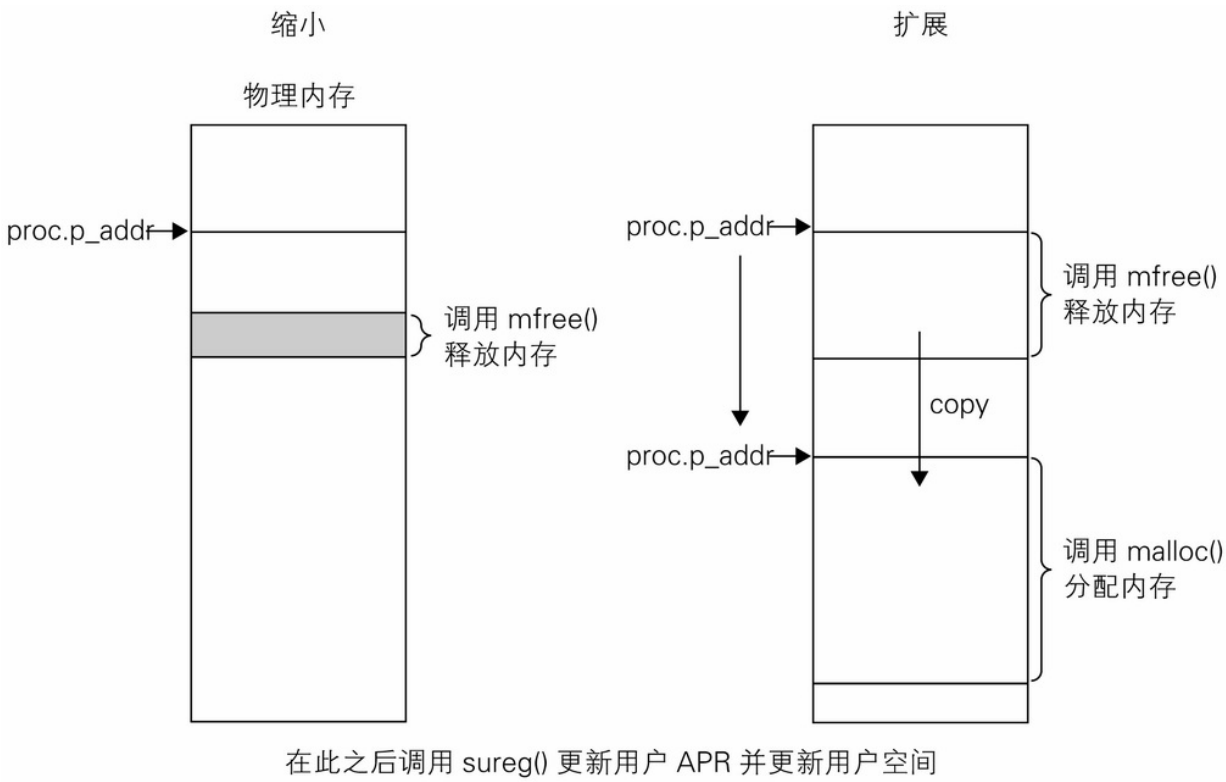

缩小
扩展
物理内存
proc.p_addr
调用 mfree()
释放内存
proc.p_addr
调用 mfree()
释放内存
copy
proc.p_addr
调用 malloc()
分配内存
在此之后调用 sureg() 更新用户 APR 并更新用户空间

图 **3-17　expand()**

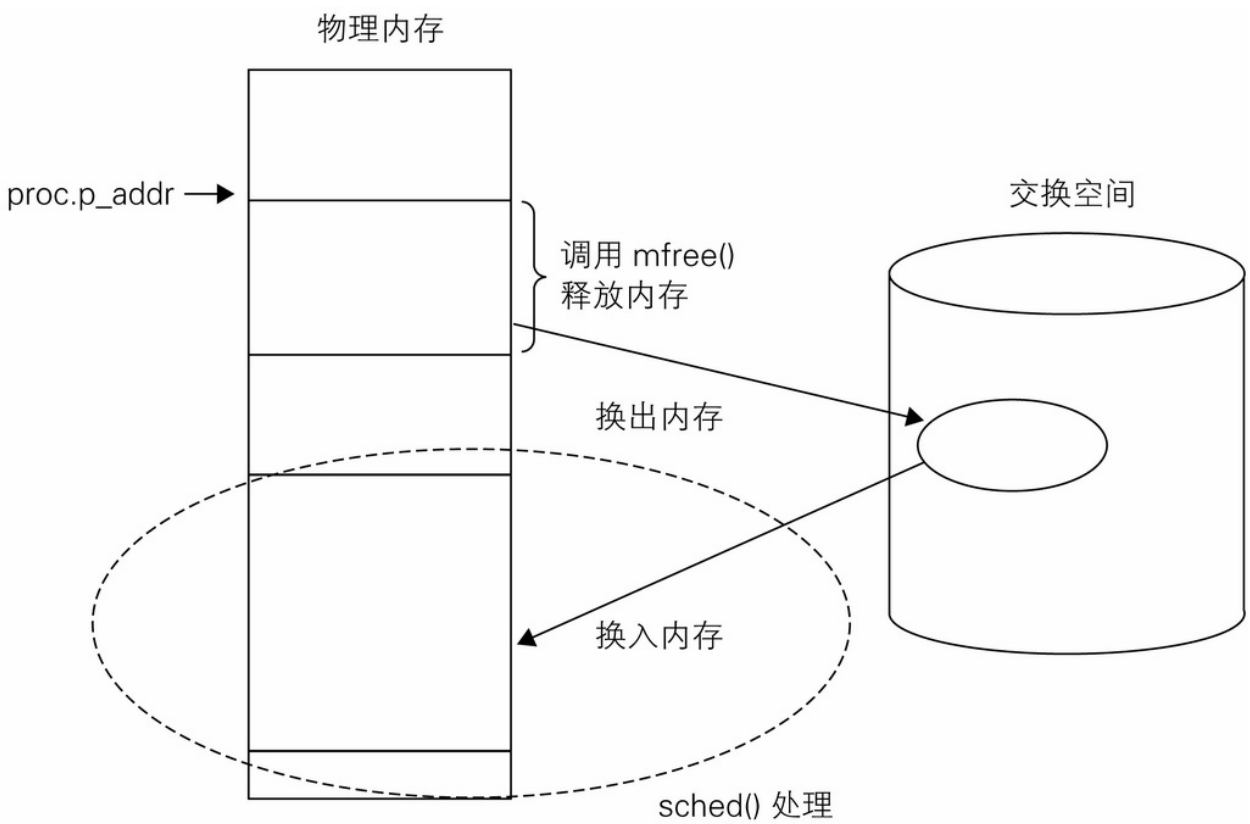

物理内存
proc.p_addr
调用 mfree()
释放内存
交换空间
换出内存
换入内存
sched() 处理

图 **3-18**　暂时将数据换出内存

表 **3-26**　**expand()** 的参数

参数	含义
`newsize`	数据段的新的长度（以 64 字节为单位）

代码清单 **3-38**　**expand()**（**ken/slp.c)**

```
 1 expand(newsize)
 2 {
 3     int i, n;
 4     register *p, a1, a2;
 5
 6     p = u.u_procp;
 7     n = p->p_size;
 8     p->p_size = newsize;
 9     a1 = p->p_addr;
10     if(n >= newsize) {
11         mfree(coremap, n-newsize, a1+newsize);
12         return;
13     }
14     savu(u.u_rsav);
15     a2 = malloc(coremap, newsize);
16     if(a2 == NULL) {
17         savu(u.u_ssav);
18         xswap(p, 1, n);
19         p->p_flag =| SSWAP;
20         swtch();
21         /* 从退出 expand() 的地方继续运行 */
22     }
23     p->p_addr = a2;
24     for(i=0; i<n; i++)
25         copyseg(a1+i, a2++);
26     mfree(coremap, n, a1);
27     retu(p->p_addr);
28     sureg();
29 }
```

8　更新表示数据段长度的 `proc.p_size` 。

10~13　需要缩小数据段长度，可调用 `mfree()` 释放不再需要的内存并返回。

14~15　如果需要扩展，则调用 `malloc()` 分配供数据段使用的新内存。因为在第 27 行将执行 `retu()` ，所以在此处执行 `savu()` 以提前保存 r5 和 r6。

16~22　如果出现内存不足的情况，则将进程从内存换出至交换空间。当进程被换入内存之后再次尝试分配内存，然后执行 `swtch()` 切换执行进程。当该进程再次成为执行进程时，从退出 `expand()` 的地方继续运行。

23~26　将数据段的内容复制到新分配的内存区域。将 `proc.p_addr` 设置为新分配的内存区域的地址，并调用 `mfree()` 释放原有的数据段。

27~28　因为数据段的地址已经更改，所以执行 `retu()` 和 `sureg()` 以更新用户空间。

3.5　进程的终止

当程序执行完毕之后，该进程的运行将被终止，被其占用的资源也会释放。进程的终止处理包括一下两个步骤。

1. 用户程序执行系统调用 `exit`，使进程进入僵尸状态（`SZOMB`）。将 `user` 结构体换出至交换空间，同时释放被其占用的内存。

2. 父进程通过执行系统调用 `wait` 取得子进程的完结状态，并负责清理处于僵尸状态的子进程。

系统调用 **exit**

系统调用 `exit` 的主要功能如下所示。

- 关闭打开的文件
- 将当前目录的参照计数器减 1
- 释放代码段
- 将 `user` 结构体复制到交换空间
- 释放数据段
- 使进程进入僵尸状态（`SZOMB`）
- 唤醒父进程和 `init` 进程
- 如果当前进程存在子进程，则将其设置为 `init` 进程的子进程

被复制到交换空间的 `user` 结构体将在父进程清理子进程时投入使用。

如果由于程序自身的错误，使得进程在进入僵尸状态时没有正确清理属于自己的的子进程，那么这些子进程将处于无人管理的状态。因此，当被终止的进程存在子进程时，将委托 `init` 进程（`proc[1]`）对子进程进行清理。`init` 进程在系统启动时被创建，该进程负责清理终止的进程、初始化终端等工作（图 3-19）。

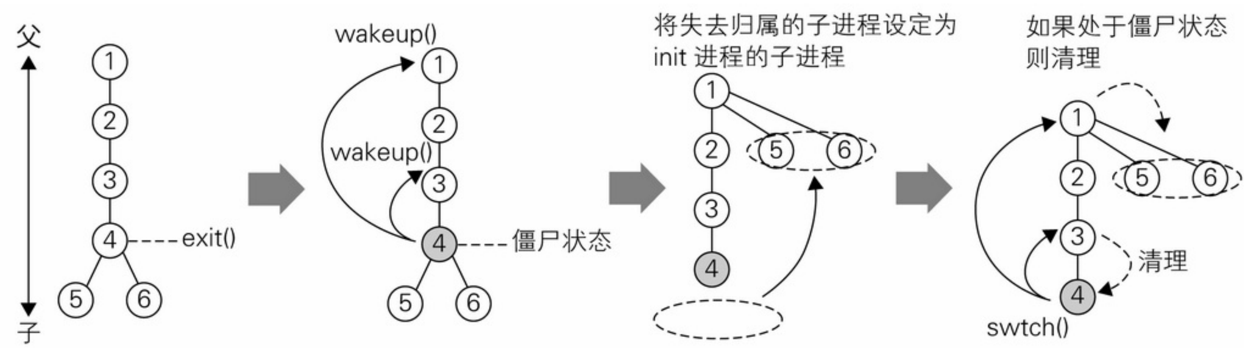


图 **3-19**　系统调用 **exit**

在处理的最后阶段执行 `swtch()` 切换执行进程。期待父进程对当前进程，`init` 进程对原本属于自己的子进程进行清理。因为已进入僵尸状态，执行系统调用 `exit` 的进程不会再次成为执行进程。

如果用户程序执行了系统调用 `exit`，内核的 `rexit()` 将被执行（表 3-27，代码清单 3-39）。`rexit()` 会调用 `exit()` 终止进程运行（代码清单 3-40）。系统调用 `exit` 的参数为进程的完结状态。完结状态保存在 `user.u_arg[0]` 中。当父进程进行清理处理时，`user.u_arg[0]` 以高位 8 比特为完结状态、低位 8 比特为错误信息的格式传递给父进程。

表 **3-27**　系统调用 **exit** 的参数

参数	含义
r0	进程的完结状态

代码清单 **3-39**　**rexit()**（**ken/sys1.c)**

```
rexit()
{

    u.u_arg[0] = u.u_ar0[R0] << 8;
    exit();
}
```

代码清单 **3-40**　**exit()**（**ken/sys1.c)**

```
exit()
{
    register int *q, a;
    register struct proc *p;

    u.u_procp->p_flag =& ~STRC;
    for(q = &u.u_signal[0]; q < &u.u_signal[NSIG];)
        *q++ = 1;
    for(q = &u.u_ofile[0]; q < &u.u_ofile[NOFILE]; q++)
        if(a = *q) {
            *q = NULL;
            closef(a);
        }
    iput(u.u_cdir);
    xfree();
    a = malloc(swapmap, 1);
    if(a == NULL)
        panic("out of swap");
    p = getblk(swapdev, a);
    bcopy(&u, p->b_addr, 256);
    bwrite(p);
    q = u.u_procp;
    mfree(coremap, q->p_size, q->p_addr);
    q->p_addr = a;
    q->p_stat = SZOMB;

loop:
    for(p = &proc[0]; p < &proc[NPROC]; p++)
    if(q->p_ppid == p->p_pid) {
        wakeup(&proc[1]);
        wakeup(p);
        for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(q->p_pid == p->p_ppid) {
            p->p_ppid = 1;
            if (p->p_stat == SSTOP)
                setrun(p);
        }
        swtch();
    }
    q->p_ppid = 1;
    goto loop;
}
```

6 如果当前处于跟踪处理中，则将跟踪标志位设置为无效。

7~8 为了忽略所有信号，将所有 `u.u_signal[n]` 设置为 1。

9~13 关闭所有由当前进程打开的文件。

14 递减当前目录的参照计数器。

15 释放代码段。

16~18 分配交换空间。

19~21 执行 `getblk()`，取得交换磁盘（用作交换空间的块设备）的块设备缓冲区。然后 执行 `bcopy()` 将包含 `user` 结构体在内的位于数据段头部的 512 字节复制到上述缓冲区，再 执行 `bwrite()` 将缓冲区内的数据写入交换空间。

22~25 释放内存中的数据段。然后把 `proc.p_addr` 设置为交换磁盘中的块编号，再将 `proc.p_stat` 设置为 `SZOMB` 。

28~39 唤醒父进程和 `init` 进程。如果存在尚未清理的子进程，则将其父进程设置为 `init` 进程。另外，如果该子进程处于 `SSTOP` 状态，则解除该状态并将其设置为可执行状态。最后执行 `swtch()` 切换执行进程。因为 `proc.p_stat` 为 `SZOMB` ，所以当前进程不会再次执行。

40~41 如果由于某种原因当前进程中才不存在父进程，则将其父进程设置为 `init` 进程，并返回第 28 行再次执行。

系统调用 **wait**

如果用户程序执行了系统调用 `wait` ，当前进程的运行将被中断，控制权会交给其他进程。`wait` 等待子进程运行结束，如果不存在子进程则不做任何处理。

系统调用 `wait` 主要有下述 3 个作用。

- 通过 `sleep()` 中断自身的运行
- 清理处于僵尸状态的子进程
- 寻找处于 `SSTOP` 状态（参见第 6 章）的子进程，并解除该状态

`wait()` 是系统调用 `wait` 的处理函数（代码清单 3-41 ）。

代码清单 **3-41　wait()**（**ken/sys1.c)**

```
wait()
{
    register f, *bp;
    register struct proc *p;

    f = 0;
loop:
    for(p = &proc[0]; p < &proc[NPROC]; p++)
    if(p->p_ppid == u.u_procp->p_pid) {
        f++;
        if(p->p_stat == SZOMB) {
            u.u_ar0[R0] = p->p_pid;
            bp = bread(swapdev, f=p->p_addr);
            mfree(swapmap, 1, f);
            p->p_stat = NULL;
            p->p_pid = 0;
            p->p_ppid = 0;
```

```
18            p->p_sig = 0;
19            p->p_ttyp = 0;
20            p->p_flag = 0;
21            p = bp->b_addr;
22            u.u_cstime[0] =+ p->u_cstime[0];
23            dpadd(u.u_cstime, p->u_cstime[1]);
24            dpadd(u.u_cstime, p->u_stime);
25            u.u_cutime[0] =+ p->u_cutime[0];
26            dpadd(u.u_cutime, p->u_cutime[1]);
27            dpadd(u.u_cutime, p->u_utime);
28            u.u_ar0[R1] = p->u_arg[0];
29            brelse(bp);
30            return;
31        }
32        if(p->p_stat == SSTOP) {
33            if((p->p_flag&SWTED) == 0) {
34                p->p_flag =| SWTED;
35                u.u_ar0[R0] = p->p_pid;
36                u.u_ar0[R1] = (p->p_sig<<8) | 0177;
37                return;
38            }
39            p->p_flag =& ~(STRC|SWTED);
40            setrun(p);
41        }
42    }
43    if(f) {
44        sleep(u.u_procp, PWAIT);
45        goto loop;
46    }
47    u.u_error = ECHILD;
48 }
```

6　将表示子进程数量的 `f` 清 0。

8~9　遍历 `proc[]` 寻找子进程。

10　递增子进程的数量。

11~31　如果子进程处于僵尸状态则进行下列处理。

- 将用户进程的 `r0` 设置为子进程的 `proc.p_pid` ，并将其作为系统调用 `wait` 的返回值。用户程序可通过该值确认执行完毕的子进程。
- 从换出至交换空间的 `user` 结构体中获得子进程占用 CPU 的时间。
- 清除 `proc` 结构体的各个成员。
- 将用户进程的 `r1` 设置为子进程的 `user.u_arg[0]` 。用户程序通过该值可了解子进程运行结束时的状态。

最后执行 `return` 。依次清理处于僵尸状态的子进程。如果需要清理多个子进程，需要父进程再次执行系统调用 `wait` 。

32~41　与跟踪相关的内容会在第 6 章中介绍。

43~46　如果发现既不处于僵尸状态，也不处于 `SSTOP` 状态的子进程，则进入睡眠状态，等待该子进程处理完毕。被唤醒后返回 `loop` 再次进行检查。

47　如果没有发现子进程则认为出错。

3.6 数据区域的扩展

系统调用 break

系统调用 break 用来扩展或缩小数据段中数据区域的长度。供用户程序调用的 C 语言库函数 malloc() 等使用 break 实现对堆区域的扩展等操作。位于虚拟地址空间的数据区域后部的地址被称为 break 。系统调用 break 的用途可以看做是调整 break 的位置（图 3-20 ）。

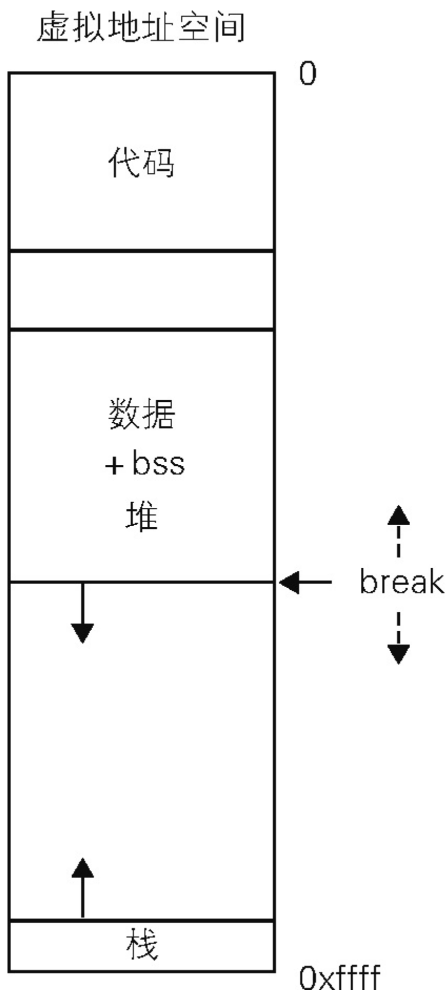


图 **3-20**　系统调用 **break**

数据区域的扩展和缩小通过 expand() 进行。伴随数据区域的扩展和缩小，栈区域的位置也会发生变化。

sbreak() 是系统调用 break 的处理函数（表 3-28，代码清单 3-42 ）。

表 **3-28**　系统调用 **sbreak** 的参数

参数	含义
u.u_arg[0]	新的 break 的虚拟地址（字节单位）

代码清单 **3-42　sbreak()**（**ken/sys1.c)**

```
1 sbreak()
2 {
3     register a, n, d;
4     int i;
5
6     n = (((u.u_arg[0]+63)>>6) & 01777);
7     if(!u.u_sep)
8         n =- nseg(u.u_tsize) * 128;
9     if(n < 0)
10        n = 0;
11    d = n - u.u_dsize;
12    n =+ USIZE+u.u_ssize;
13    if(estabur(u.u_tsize, u.u_dsize+d, u.u_ssize, u.u_sep))
14        return;
15    u.u_dsize =+ d;
16    if(d > 0)
17        goto bigger;
18    a = u.u_procp->p_addr + n - u.u_ssize;
19    i = n;
20    n = u.u_ssize;
21    while(n--) {
22        copyseg(a-d, a);
23        a++;
24    }
25    expand(i);
26    return;
27
28 bigger:
29    expand(n);
30    a = u.u_procp->p_addr + n;
31    n = u.u_ssize;
32    while(n--) {
33        a--;
34        copyseg(a-d, a);
35    }
36    while(d--)
37        clearseg(--a);
38 }
```

6~12 将通过参数获得的以字节为单位的地址变成以 64 字节为单位的形式，然后减去按页（以 8KB 为单位）划分的代码段的长度，得到新的数据区域的长度。d 用于保存当前数据区域和新的数据区域的长度差。n 用于保存新的数据段全体的长度。

13~14　更新用户 APR，以更新用户空间。

15　更新通过 user 结构体管理的数据区域长度。

16~17　如果是扩展数据区域，则跳转到 bigger 。

18~26　如果是缩小数据区域，则将栈区域向物理地址的低位地址方向移动，并执行 expand() 删除剩余的部分。

29~37　如果是扩展数据区域，则将栈区域向物理地址的高位地址方向移动，然后执行 expand() 扩展数据段。

3.7　管理内存和交换空间

进程通过虚拟地址访问分配给自身的物理内存。内核需要为各进程分配物理地址，因此必须对物理内存中已被使用和尚未使用的区域进行管理。出于同样的理由，内核对交换空间也需要进行管理。

本节对内核物理内存和交换空间中未使用区域的管理方法进行说明。

map 结构体

内核利用 map 结构体（代码清单 3-43，表 3-29 ）对物理内存和交换空间的未使用区域进行管理。map 结构体的成员包含未使用区域的地址和长度。

代码清单 **3-43　map**（**ken/malloc.c)**

```
struct map
{
  char *m_size;
  char *m_addr;
};
```

表 **3-29　map** 结构体

成员	含义
*m_size	未使用区域的长度
*m_addr	未使用区域的地址

coremap[] 用来管理物理内存，swapmap[] 用来管理交换空间（代码清单 3-44，表 3-30 ）。数组元素的排列顺序与地址顺序相同（图 **3-21** ）。对物理内存和交换空间的未使用区域采用同样的算法进行管理。

代码清单 **3-44　coremap** 和 **swapmap**（**system.h)**

```
int    coremap[CMAPSIZ];
int    swapmap[SMAPSIZ];
```

请注意，coremap[] 和 swapmap[] 分别以 64 字节和 512 字节为单位管理未使用的区域。

表 **3-30　coremap[]** 和 **swapmap[]**

对象	管理用区域	管理单位	备注
物理内存	coremap	64 字节（APR 的最小管理单位）	对物理地址进行管理
交换空间	swapmap	512 字节（块单位）	地址与交换磁盘的块编号相对应

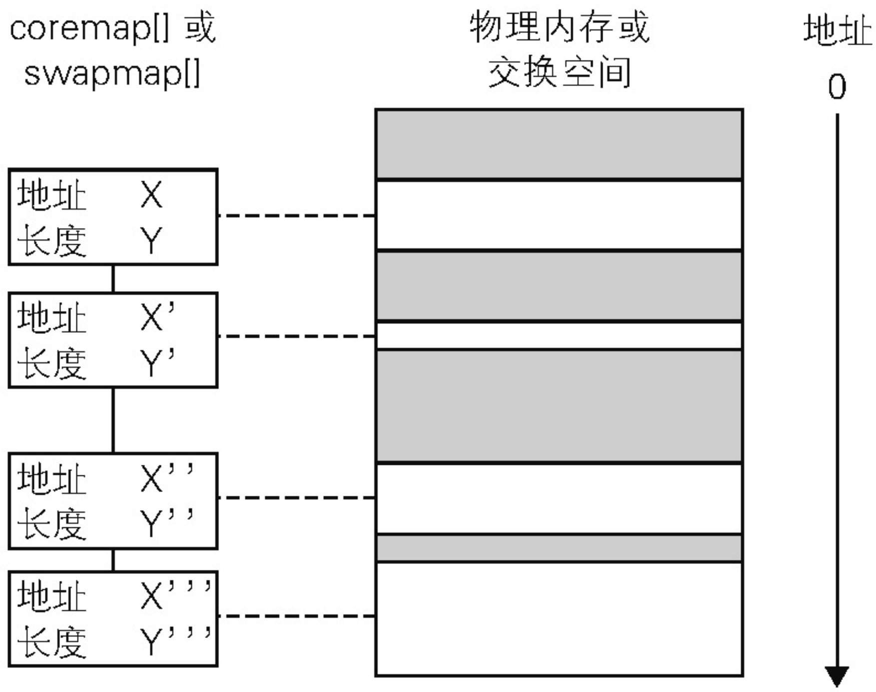


图 **3-21** 管理未使用区域

`coremap[]` 和 **`swapmap[]`** 于系统启动时在 **`main()`** 中被初始化 （详见第 14 章）。

获取未使用区域

在获取物理内存和交换空间的未使用区域时采用了 First Fit 算法，即从起始位置开始遍历数组，寻找满足长度要求并排在最前面的元素（图 3-22 ）。

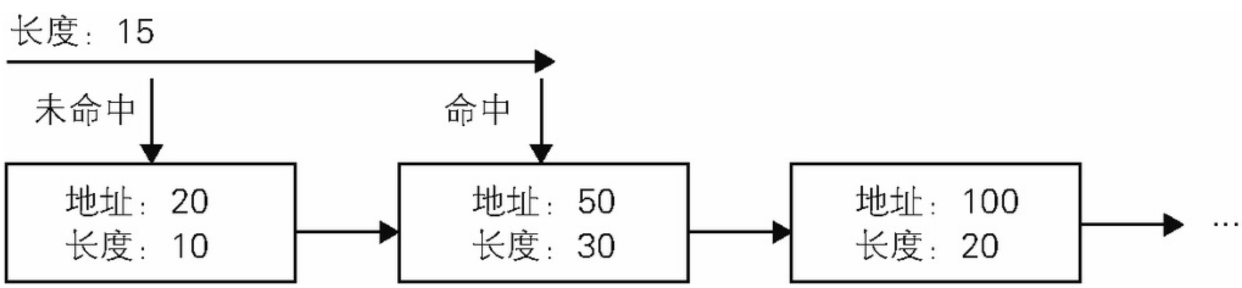


图 **3-22** **First Fit**

以找到的未使用区域的起始位置为起点，分配所需长度的区域。然后增加该数组元素的地址值，并递减其长度值，使增加和减少的长度等于新分配区域的长度（图 3-23 ）。

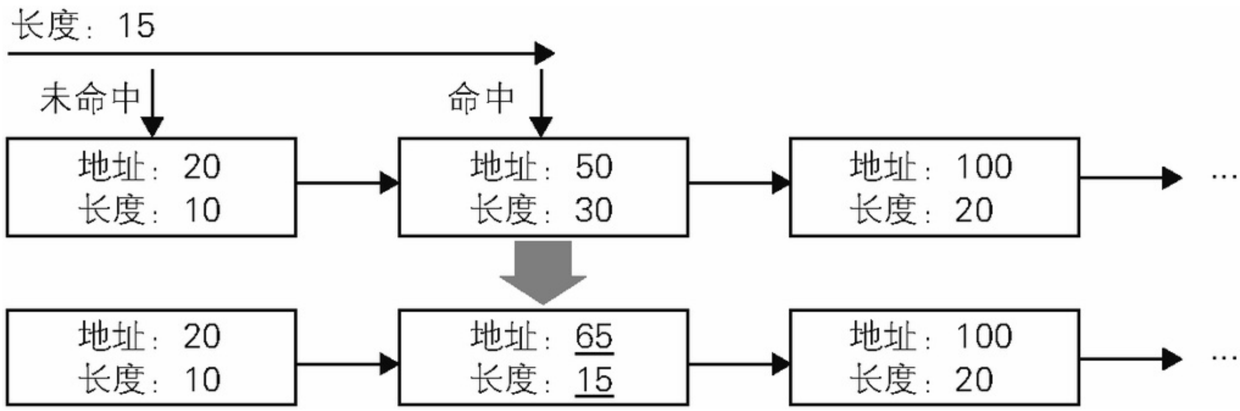


图 **3-23**　命中

如果元素的长度和所需长度相同则删除该元素，并将所有余下的元素向前移动 1 个位置（图 3-24 ）。

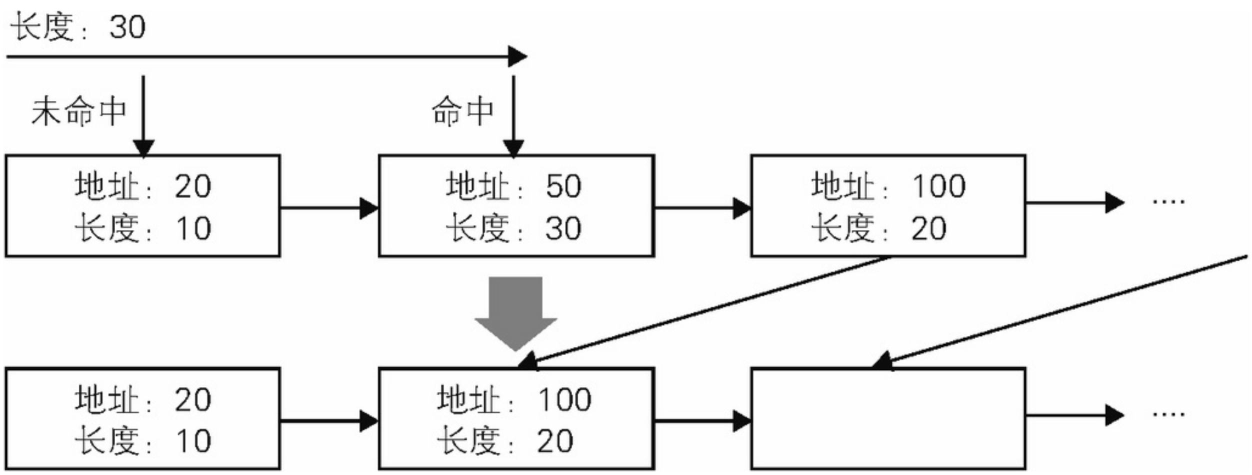


图 **3-24**　向前方移动

在数组有效元素的尾部，有一个长度为 0 的元素。当通过循环依次处理数组时，如果遇见该元素，循环将终止。这个长度为 0 的元素被称为“哨兵”或“看守”（图 3-25 ）。

图 **3-25**　哨兵

`malloc()` 是取得物理内存和交换空间的函数（表 3-31，代码清单 3-45 ）。该函数返回所取得区域的地址。取得失败时返回 0。

表 **3-31**　**malloc()** 的参数

参数	含义
`mp`	`coremap[]` 或 `swapmap[]`

size	获取长度

代码清单 **3-45 malloc()**（**ken/malloc.c)**

```
malloc(mp, size)
struct map *mp;
{
    register int a;
    register struct map *bp;

    for (bp = mp; bp->m_size; bp++) {
        if (bp->m_size >= size) {
            a = bp->m_addr;
            bp->m_addr =+ size;
            if ((bp->m_size =- size) == 0)
                do {
                    bp++;
                    (bp-1)->m_addr = bp->m_addr;
                } while ((bp-1)->m_size = bp->m_size);
            return(a);
        }
    }
    return(0);
}
```

释放区域

mfree() 是用来释放物理内存和交换空间的函数（表 3-32，代码清单 3-46 ）。从 coremap[] 、swapmap[] 的起始位置开始遍历数组，直到到达指定地址的要素，然后将 其返还至数组中（图 3-26 ）。如果释放对象区域的地址与插入位置前后区域的地址相连，则合并相关区域（图 3-27 ）。

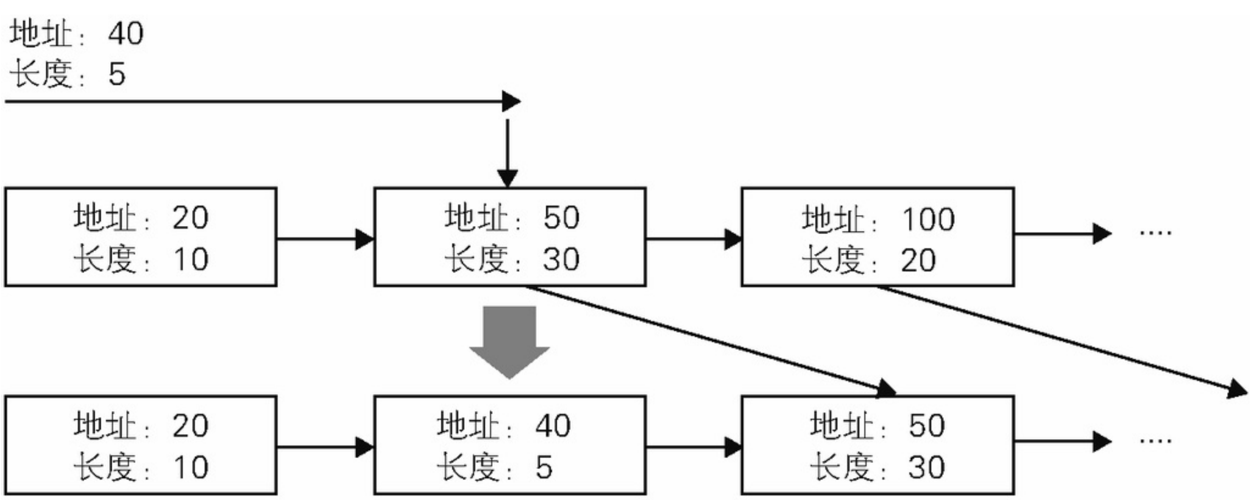


图 **3-26** 释放

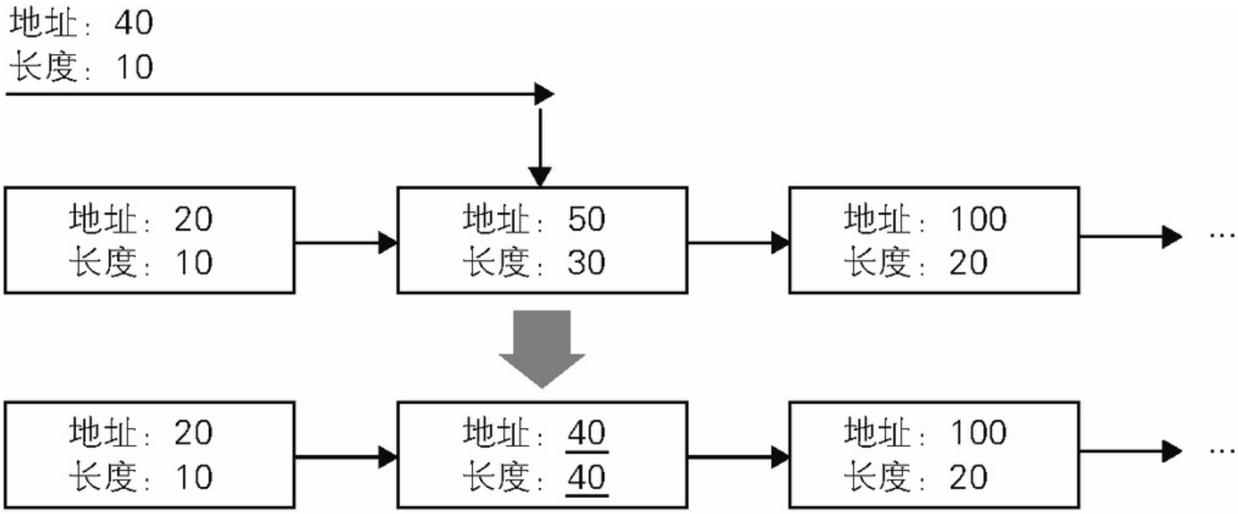


图 **3-27**　区域的合并

表 **3-32**　**mfree()** 的参数

参数	含义
mp	coremap[] 或 swapmap[]
size	释放区域的长度
aa	释放区域的地址

代码清单 **3-46**　**mfree()**（**ken/malloc.c)**

```
mfree(mp, size, aa)
struct map *mp;
{
    register struct map *bp;
    register int t;
    register int a;

    a = aa;
    for (bp = mp; bp->m_addr<=a && bp->m_size!=0; bp++);
    if (bp>mp && (bp-1)->m_addr+(bp-1)->m_size == a) {
        (bp-1)->m_size =+ size;
        if (a+size == bp->m_addr) {
            (bp-1)->m_size =+ bp->m_size;
            while (bp->m_size) {
                bp++;
                (bp-1)->m_addr = bp->m_addr;
                (bp-1)->m_size = bp->m_size;
            }
        }
    } else {
        if (a+size == bp->m_addr && bp->m_size) {
            bp->m_addr =- size;
            bp->m_size =+ size;
        } else if (size) do {
            t = bp->m_addr;
            bp->m_addr = a;
            a = t;
```

```
            t = bp->m_size;
            bp->m_size = size;
            bp++;
        } while (size = t);
    }
}
```

3.8 小结

- 系统调用 fork 用来创建进程。子程序可被视作父进程的拷贝
- 系统调用 wait 使父进程在等待子进程执行结束时进入睡眠状态，并在子进程执行结束后对其进行清理
- 系统调用 exec 将程序执行文件读取到内存，并构筑程序的执行环境
- 系统调用 exit 使进程进入僵尸状态。进入该状态的进程由其父进程清理
- init 进程负责对失去父进程、处于无所属状态的进程进行清理
- 执行进程中断后，控制权将切换到具有更高执行优先级，并处于可执行状态的进程
- 通过保存和恢复 r5、r6、用户 APR、内核 PAR6 的值实现上下文切换
- 利用 sleep() 和 wakeup() 实现对资源的等待处理
- 利用 map 结构体，并采用相同的算法管理物理内存和交换空间的未使用区域
- 采用 First Fit 算法获取未使用区域的处理
- 在启动时对 coremap[] 和 swapmap[] 进行初始化

第 **4** 章　交换处理

4.1　什么是交换处理

执行程序时必须将代码和数据读入内存。内存的处理速度很快，但容量受限。随着进程数量的增多，内存将无法容纳所有程序的代码和数据。

因此，内核通过定期执行交换处理，将处于休眠状态或执行优先级较低的进程从内存移至处理速度较慢但容量较大的磁盘等交换空间（换出，swap out），当这些进程成为可执行状态时，再将其移回内存（换入，swap in）。通过交换处理,只有马上需要执行的进程才会存在于内存中,因此可以更有效地利用有限的内存资源 。同时也可以避免内存容量的限制,以并列方式执行更多的进程 （图 4-1 ）。

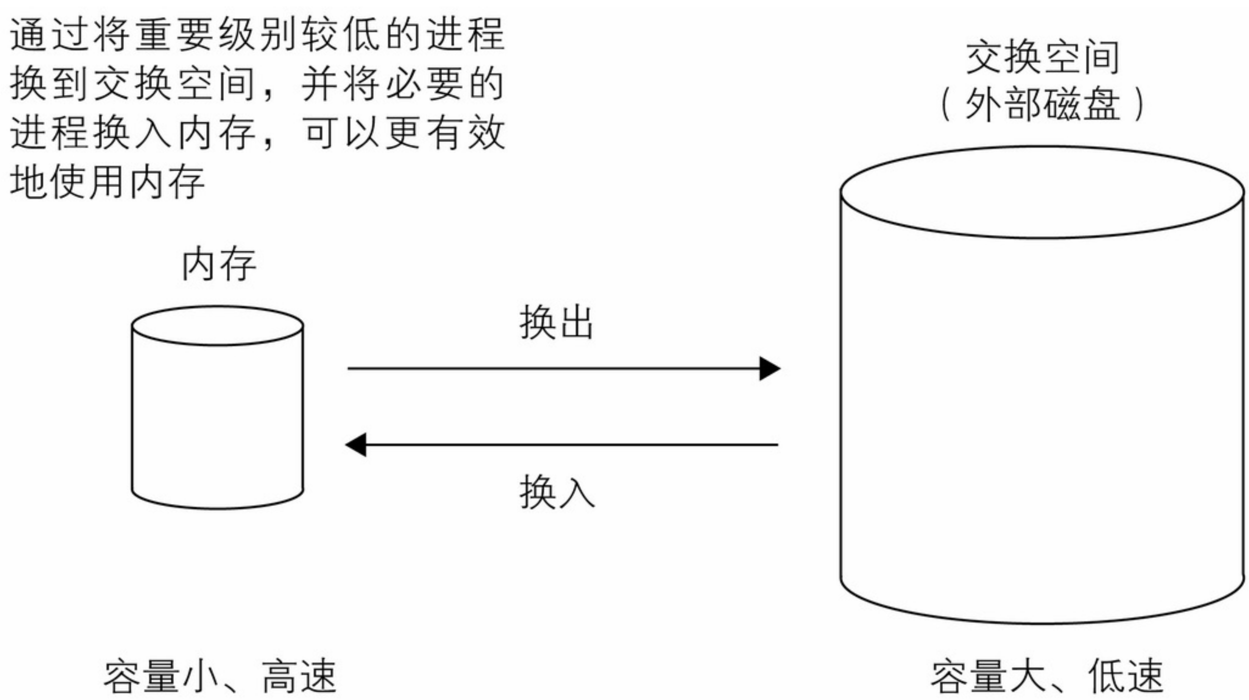


图 **4-1**　交换处理

代码段和数据段

对进程进行换入处理时，当代码段读入到内存后，位于交换空间内的代码段仍将保留。当进程执行结束或被换出内存时，如果代码段不再被内存中的任何进程所参照，则会从内存中被释放。

与此相反，对数据段进行换入、换出处理时,作为处理对象的数据将从原有设备上被释放 （图 4-2 ）。

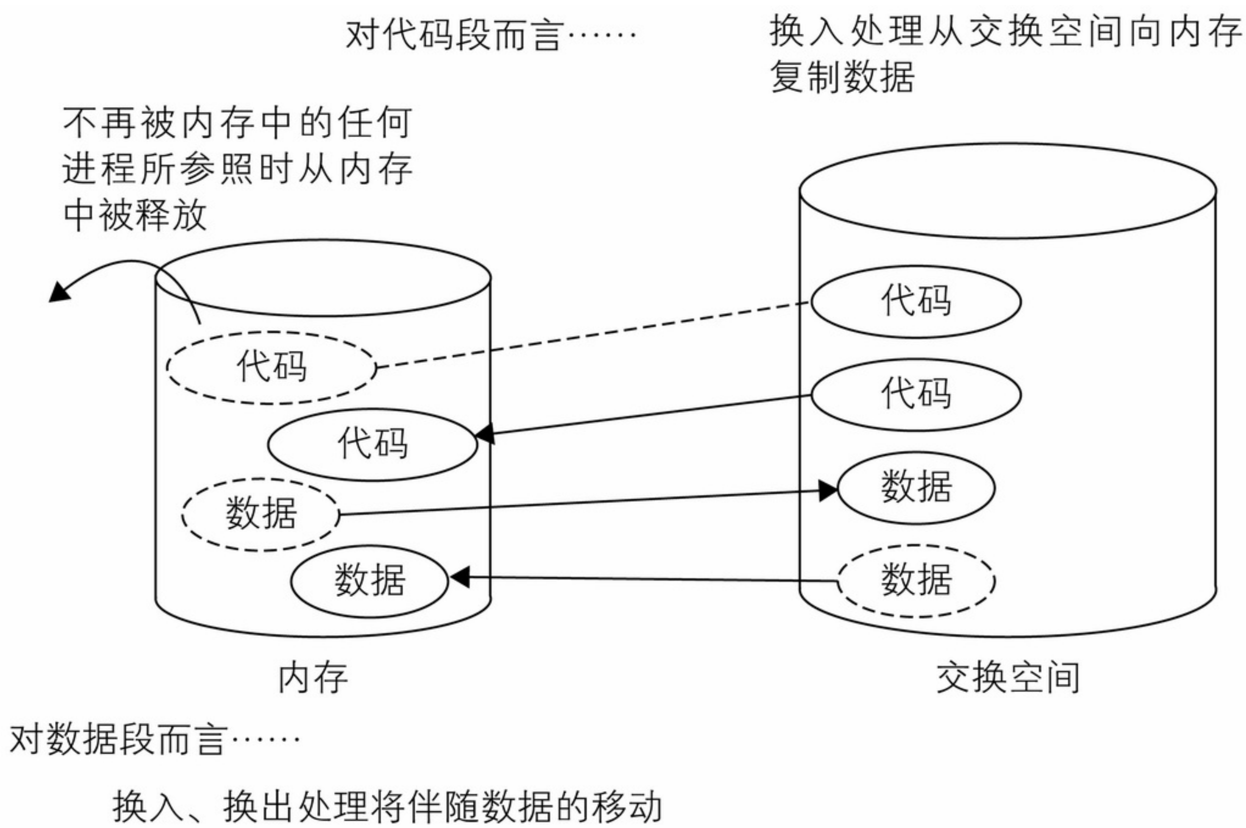


图 **4-2**　代码段和数据段

代码段与数据段的不同之处在于，代码段数据是只读的，这使交换空间和内存中的数据总保持一致，因此系统对其进行了上述优化。

sched()

`sched()` 也被称作 swapper 函数，用来寻找作为交换处理对象的进程（代码清单 4-2 ），由系统启动时生成的进程 **`proc[0]`** 定期调用 。`swtch()` 也由 `proc[0]` 调用，因此 **`proc[0]`** 被称为调度进程或调度器 。

`sched()` 从 `proc[]` 中寻找满足下述条件的进程作为换入处理的对象。

- 位于交换空间的时间最长
- 处于可执行状态

如果在执行换入处理时内存容量不足，对满足下述条件的进程执行换出处理。

- 位于内存中
- 处于 `SWAIT` 或 `SSTOP` 状态

如果不存在满足条件的进程，则放宽条件再次寻找作为换出处理对象的进程。

- 滞留内存的时间最长
- 处于 `SRUN` 或 `SSLEEP` 状态

但是在这种情况下，作为换入对象的进程距上次被换出的时间必须大于或等于 3 秒，而作为换出对象的进程距上次被换入的时间必须大于或等于 2 秒。这是为了防止由于过度的交换处理，导致无法继续进行原本的处理的情况。

上述处理将反复执行，直到不再存在作为换入或换出对象的进程（图 4-3 ）。如果处理结束时不再存在可作为换入对象的进程，`runout` 标志变量会被设置为大于 0 的值。如果不再存在可作为换出对象的进程，`runin` 标志变量会被设置为大于 0 的值（代码清单 4-1，表 4-1 ）。

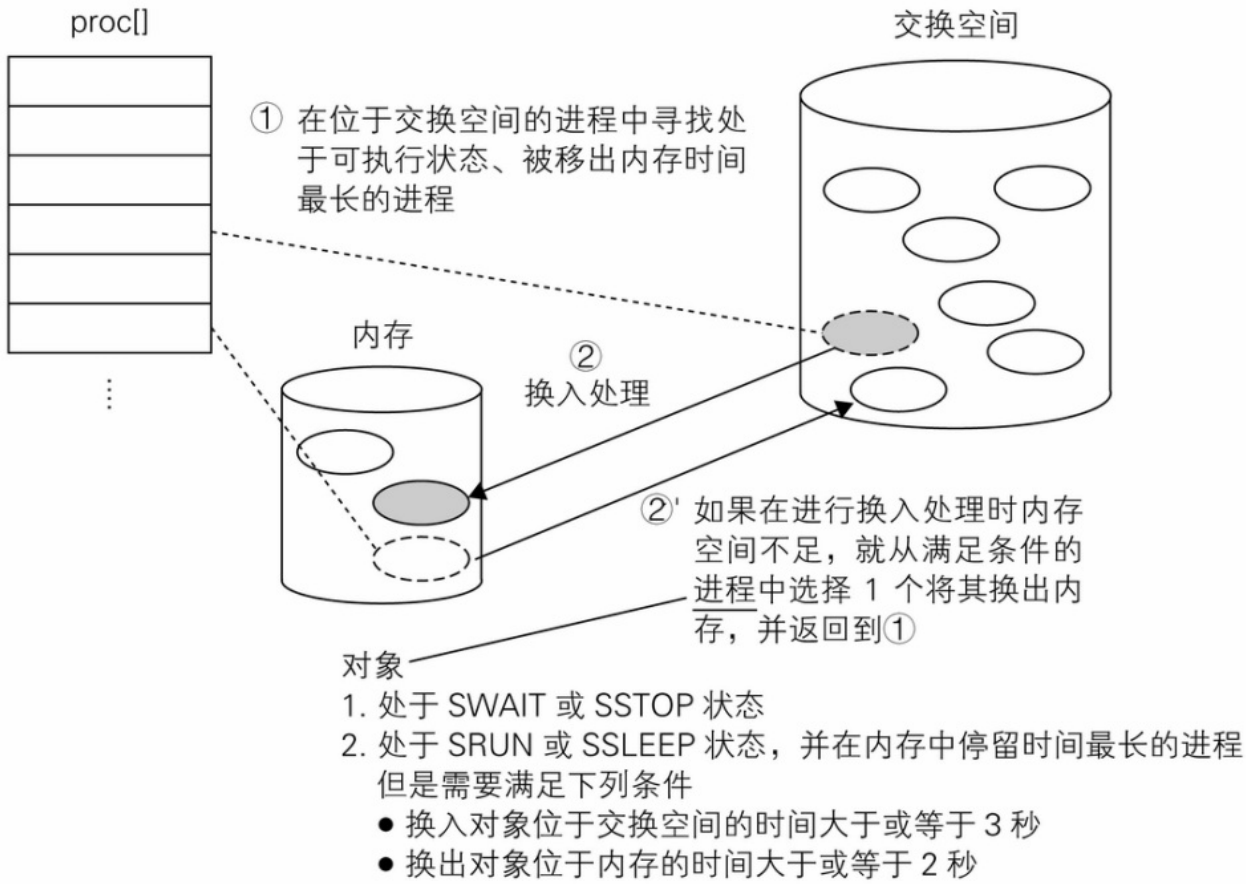


图 **4-3　sched()**

代码清单 **4-1　runin** 和 **runout**（**system.h)**

```
1 char runin;
2 char runout;
```

表 **4-1　sched()** 的结束标志

标志变量	含义
`runout`	不存在可作为换入对象的进程
`runin`	不存在可作为换出对象的进程

代码清单 **4-2　sched()**（**sys/slp.c)**

```
sched()
{
    struct proc *p1;
    register struct proc *rp;
    register a, n;

    goto loop;

sloop:
    runin++;
    sleep(&runin, PSWP);

loop:
    /* 寻找作为换入对象的进程 */
    spl6();
    n = -1;
    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
    if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)==0 &&
        rp->p_time > n) {
        p1 = rp;
        n = rp->p_time;
    }
    if(n == -1) {
        runout++;
        sleep(&runout, PSWP);
        goto loop;
    }

    spl0();
    rp = p1;
    a = rp->p_size;
    if((rp=rp->p_textp) != NULL)
        if(rp->x_ccount == 0)
            a =+ rp->x_size;
    if((a=malloc(coremap, a)) != NULL)
        goto found2;

    /* 寻找处于SWAIT或SSTOP状态的进程作为换出对象 */
    spl6();
    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
    if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
        (rp->p_stat == SWAIT || rp->p_stat==SSTOP))
        goto found1;

    /* 寻找在内存中停留时间最长的进程作为换出对象 */
    if(n < 3)
        goto sloop;
    n = -1;
    for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
    if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
       (rp->p_stat==SRUN || rp->p_stat==SSLEEP) &&
        rp->p_time > n) {
        p1 = rp;
        n = rp->p_time;
    }
    if(n < 2)
        goto sloop;
    rp = p1;

found1:
    /* 换出处理 */
    spl0();
    rp->p_flag =& ~SLOAD;
    xswap(rp, 1, 0);
    goto loop;

found2:
```

```
68     /* 换入处理 */
69         if((rp=p1->p_textp) != NULL) {
70         if(rp->x_ccount == 0) {
71             if(swap(rp->x_daddr, a, rp->x_size, B_READ))
72                 goto swaper;
73             rp->x_caddr = a;
74             a =+ rp->x_size;
75         }
76         rp->x_ccount++;
77     }
78     rp = p1;
79     if(swap(rp->p_addr, a, rp->p_size, B_READ))
80         goto swaper;
81     mfree(swapmap, (rp->p_size+7)/8, rp->p_addr);
82     rp->p_addr = a;
83     rp->p_flag =| SLOAD;
84     rp->p_time = 0;
85     goto loop;
86
87 swaper:
88     panic("swap error");
89 }
```

9~11 不存在换出对象时的处理。设置 runin 标志变量后进入睡眠状态。被唤醒后，从寻找换入对象进程处继续进行交换处理。

15 将处理器优先级设置为 6，防止发生中断。这是为了避免代表在内存或交换空间内生存时间的 proc.p_time 因为时钟中断而发生改变。关于中断和处理器优先级，请参照第 5 章的内容。

16~27 在处于可执行状态（SRUN ）并且处于被换出状态（!SLOAD ）的进程中，寻找在交换空间中停留时间最长（proc.p_time 具有最大值）的进程。如果不存在满足条件的进程，则在设置 runout 标志变量后进入睡眠状态。

32~34 如果换入进程使用的代码段在内存中不存在，说明其代码段也需要被换入内存，因此需要增加分配给进程的内存，增加的长度为代码段的长度。

35~36 执行 malloc() 分配换入进程使用的内存。如果分配成功，则跳转至 found2 。

39~43 如果内存空间不足，需要将相对不重要的进程换出至交换空间。首先寻找存在于内存之中（SLOAD ）、既不处于交换处理（!SLOCK ）也不是系统进程（!SSYS ）的进程。该进 程还必须满足下述条件。优先级大于等于 0，处于睡眠状态（SWAIT ）或者因跟踪处理处于停止状态（SSTOP ）。如果找到满足上述条件的进程，则跳转至 found1 。

46~58 如果没有找到满足条件的进程，则放宽进行换出处理的条件。但是，如果作为换入对象的进程距上次换出的时间小于 3 秒，设置 runin 标志变量后会进入睡眠状态。

寻找处于睡眠状态（SSLEEP ，此时优先级为负值）或可执行状态（SRUN ），并且滞留内存时间最长（proc.p_time 具有最大值）的进程。但是，如果该进程距上次换入的时间小于 2 秒，设置 runin 标志变量后会进入睡眠状态。

62~65 执行换出处理。在清除换出对象进程的 SLOAD 标志位之后，调用 xswap() 进行换出处理。该函数的第 2 个参数设定为 1，表示将从内存中释放对象进程。换出处理结束后，返回 loop 并再次选择作为换入对象的进程。

67~85 执行进程的换入处理。首先换入代码段，如果进程所需的代码段在内存中不存在，则调用 swap() 将其换入内存。关于 swap() 的详细介绍请见第 7 章。换入处理结束后，用代码段的物理地址设置 text[] 中代表该代码段的元素，同时递增参照计数器以反映内存中的进程对代码段的参照次数。然后换入数据段。执行 swap() 进行换入处理。释放位于交换空间的数据段，更新 proc[] 元素后返回

loop，再次寻找是否还存在其他的换入对象。

xswap()

xswap() 是用来对进程的数据段 进行换出处理的函数（表 4-2，代码清单 4-3 ）。通过参数可以指定被换出的数据段是否需要从内存中释放。如果需要释放，则将被换出的数据段从内存移动至交换空间。如果不需要释放，则将其从内存复制到交换空间。newproc() 使用 xswap() 的从内存复制到交换空间的功能（图 4-4 ）。

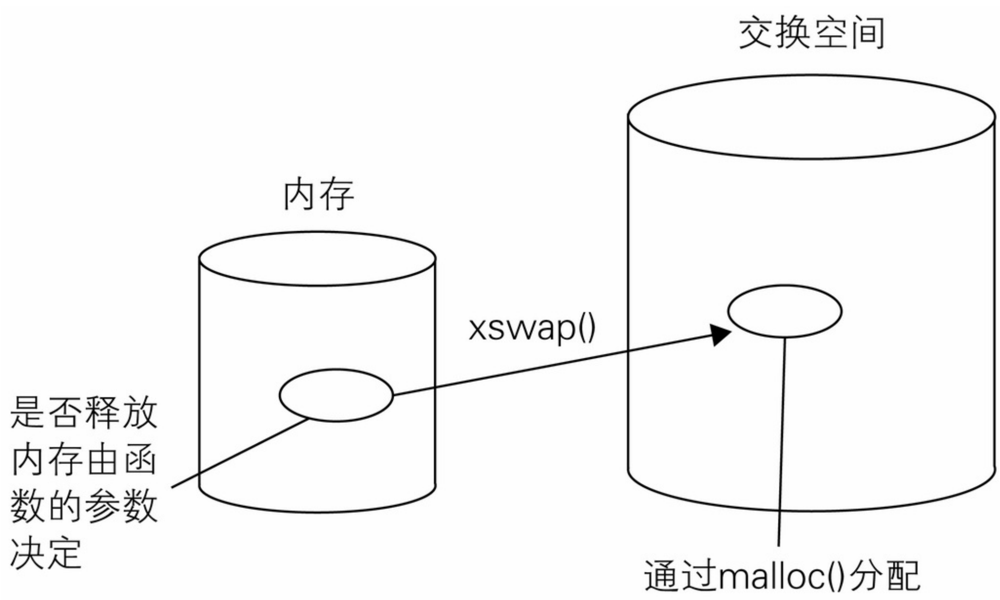


图 **4-4　xswap()**

表 **4-2　xswap()** 的参数

参数	含义
p	proc[] 元素，代表被换出的进程
ff	是否释放内存的标志
os	被换出数据的长度（以 64 字节为单位），如果为 0 则使用 proc.p_size

代码清单 **4-3　xswap()**（**ken/text.c)**

```
xswap(p, ff, os)
int *p;
{
    register *rp, a;

    rp = p;
    if(os == 0)
        os = rp->p_size;
    a = malloc(swapmap, (rp->p_size+7)/8);
    if(a == NULL)
        panic("out of swap space");
    xccdec(rp->p_textp);
```

```
13     rp->p_flag =| SLOCK;
14     if(swap(a, rp->p_addr, os, 0))
15         panic("swap error");
16     if(ff)
17         mfree(coremap, os, rp->p_addr);
18     rp->p_addr = a;
19     rp->p_flag =& ~(SLOAD|SLOCK);
20     rp->p_time = 0;
21     if(runout) {
22         runout = 0;
23         wakeup(&runout);
24     }
25 }
```

7~8　如果第 3 个参数 os 的值为 0，将其设置为换出对象进程的数据段的长度。

9~11　分配交换空间。malloc() 返回交换磁盘的块编号。

12　执行 xccdec()，递减换出对象进程所参照的代码段的参照计数器，该计数器反映内存中的进程对此代码段的参照次数。

13　设置换出对象进程的 SLOCK 标志位（表示处于交换处理中）。

14~15　执行 swap() 进行换出处理。

16~17　如果第 2 个参数 ff 不为 0，则从内存中释放换出对象进程的数据段。

18~20　将换出的对象进程的数据段的地址换成为其分配的交换空间的地址（块编号）。清除 SLOAD 和 SLOCK 标志位，并将在交换空间内表示滞留时间的 proc.p_time 清 0。

21~24　如果设置了 runout 标志变量（表示不存在可换入的进程），则会启动调度器。

4.2　共享代码段的处理

代码段是用来容纳程序指令的只读 区域。因为指令不会发生变化，所以如果某个程序同时存在多个运行中的进程，这些进程将共享同一个代码段 。通过这种方式可以节约内存的使用量。

代码段通过 text 结构体的数组 text[] 进行管理（代码清单 4-4，代码清单 4-5，表 4-3 ）。text[] 的每个元素分别对应一个代码段。各个进程所使用的 text[] 元素由 proc.p_textp 设定。在多个进程共享代码段时，这些进程将指向 text[] 的同一个元素。代码段的长度由 user.u_tsize 表示（图 4-5 ）。

text[] 元素持有指向 inode[] 中对应程序执行文件的元素指针。内核通过此处的设定来判断是否存在多个进程试图运行同一个程序。如果程序执行文件作为代码段使用，那么将设置 inode[] 中对应该文件元素的 ITEXT 标志位。关于 inode[] 将在第 9 章进行详细介绍。

代码清单 **4-4　text[]**（**text.h)**

```
struct text
{
    int    x_daddr;
    int    x_caddr;
    int    x_size;
    int    *x_iptr;
    char   x_count;
    char   x_ccount;
} text[NTEXT];
```

代码清单 **4-5　NTEXT（param.h)**

```
#define    NTEXT    40
```

表 **4-3　text** 结构体

成员	含义
x_daddr	交换磁盘中的地址（1 个块=512字节）
x_caddr	读入内存时的物理内存地址（以64字节为单位）
x_size	代码段的长度（以64字节为单位）
*x_iptr	指向inode[] 中对应程序执行文件的元素
x_count	以所有进程为对象的参照计数器
x_ccount	以内存中的进程为对象的参照计数器

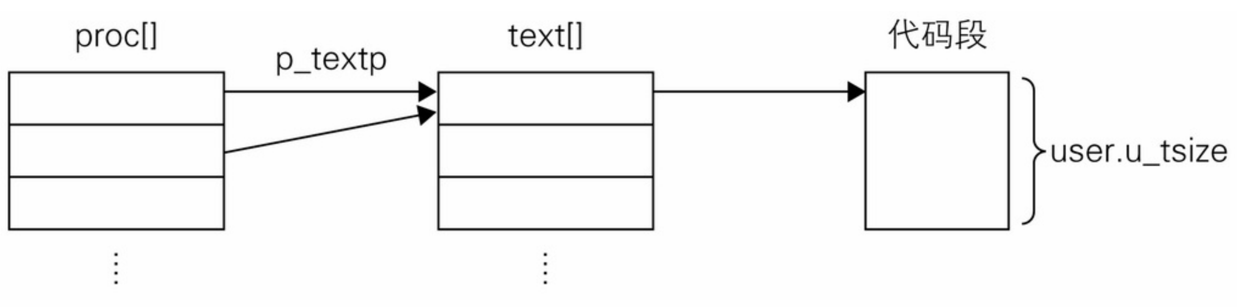


图 **4-5**　进程间共享代码段

当内存中的所有进程 不再参照某个代码段时，该代码段将从内存中被释放。当所有进程 不再参照 text[] 的某个元素时，此元素对应的代码段将从交换空间中被释放，同时该元素也将被释放。为了区分上述两种情况，text 结构体拥有两种参照计数器，分别是以所有进程为对象的参照计数器 x_count ，以及以内存中的进程为对象的参照计数器 x_ccount 。

创建新的代码段时，需要按照下述步骤进行操作（图 4-6 ）。

1. 取得 text[] 元素。

2. 读取程序的指令并暂时保存到进程的数据区域。

3. 对数据段进行换出处理，并释放内存中的数据段。

4. 调度器（proc[0] ）将该进程换入内存，并将数据段中的指令作为代码段配置到进程的虚拟地址空间

中。

如果所需的代码段已经位于内存之中，则省略上述处理。

如果所需的代码段位于交换空间中，则省略上述 1~3 的处理。如果 **inode[]** 中设置了对应程序执行文件元素的 **Sticky Bit**（即设置了 **inode.i_mode** 的 **ISVTX** 标志位）**,**即使代码段没有被任何进程参照也不会从交换空间释放 。Shell 等使用频率较高的程序通常会设置 Sticky Bit，这样可以通过省略上述 1~3 的处理改善程序的启动速度。

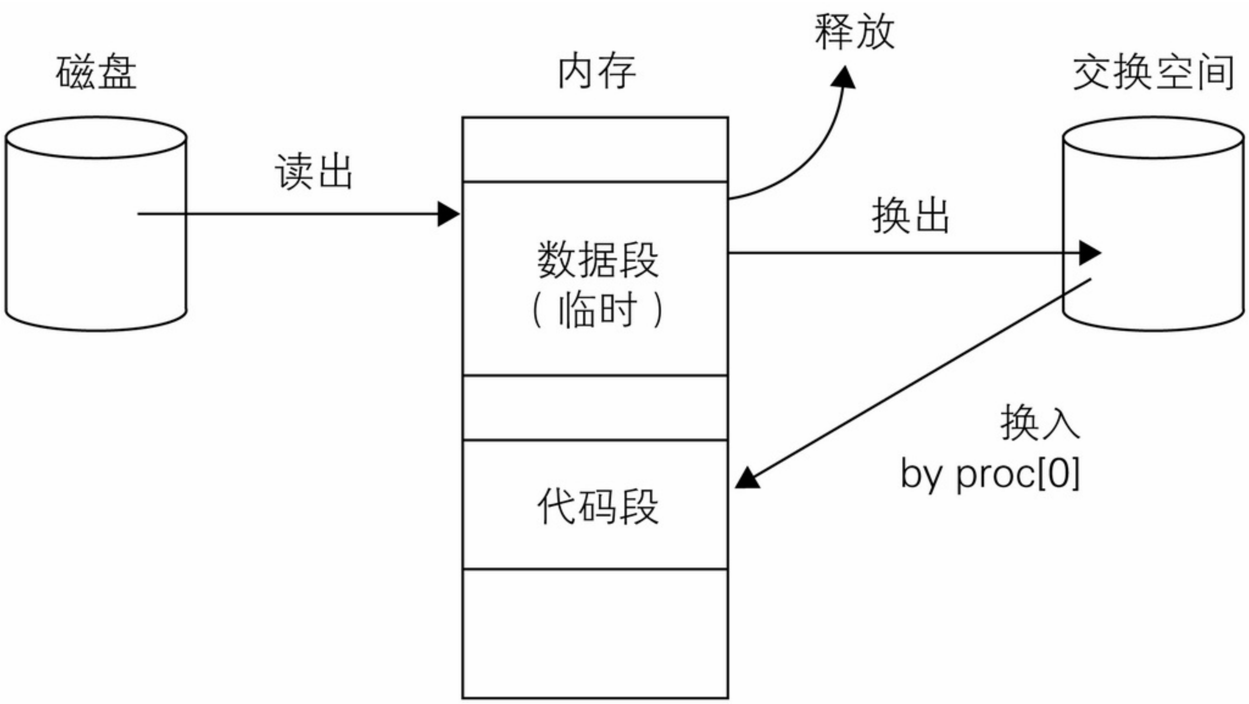


图 **4-6**　代码段的创建

xalloc()

xalloc() 将代码段分配给执行进程（表 4-4，代码清单 4-6 ）。该函数只被 exec() 调用。

xalloc() 从 text[] 中寻找未使用的元素，同时检查所需代码段是否已经存在于 text[] 中。如果已经存在**,**则将该元素分配给进程 。

如果所需的代码段在 text[] 中不存在，且 text[] 中存在未使用的元素时，将该元素分配给进程。在将程序的指令读取到数据段后，将数据段换出至交换空间。对 inode[] 中对应程序执行文件的元素设置 ITEXT 标志位，将其标明为代码段，然后递增该 inode[] 元素的参照计数器。代码段的换入处理由 **sched[]** 进行。

表 **4-4**　**xalloc()** 的参数

参数	含义
ip	inode[] 中对应程序执行文件的元素

代码清单 **4-6**　**xalloc()**（**ken/text.c)**

```
 1 xalloc(ip)
 2 int *ip;
 3 {
 4     register struct text *xp;
 5     register *rp, ts;
 6 
 7     if(u.u_arg[1] == 0)
 8         return;
 9     rp = NULL;
10     for(xp = &text[0]; xp < &text[NTEXT]; xp++)
11         if(xp->x_iptr == NULL) {
12             if(rp == NULL)
13                 rp = xp;
14         } else
15             if(xp->x_iptr == ip) {
16                 xp->x_count++;
17                 u.u_procp->p_textp = xp;
18                 goto out;
19         }
20     if((xp=rp) == NULL)
21         panic("out of text");
22     xp->x_count = 1;
23     xp->x_ccount = 0;
24     xp->x_iptr = ip;
25     ts = ((u.u_arg[1]+63)>>6) & 01777;
26     xp->x_size = ts;
27     if((xp->x_daddr = malloc(swapmap, (ts+7)/8)) == NULL)
28         panic("out of swap space");
29     expand(USIZE+ts);
30     estabur(0, ts, 0, 0);
31     u.u_count = u.u_arg[1];
32     u.u_offset[1] = 020;
33     u.u_base = 0;
34     readi(ip);
35     rp = u.u_procp;
36     rp->p_flag =| SLOCK;
37     swap(xp->x_daddr, rp->p_addr+USIZE, ts, 0);
38     rp->p_flag =& ~SLOCK;
39     rp->p_textp = xp;
40     rp = ip;
41     rp->i_flag =| ITEXT;
42     rp->i_count++;
43     expand(USIZE);
44 
45 out:
46     if(xp->x_ccount == 0) {
47         savu(u.u_rsav);
48         savu(u.u_ssav);
49         xswap(u.u_procp, 1, 0);
50         u.u_procp->p_flag =| SSWAP;
51         swtch();
52     }
53     xp->x_ccount++;
54 }
```

7~8 代码段的长度为 0 时不做任何处理。u.u_arg[1] 由 exec() 设置为程序执行文件的代码长度（以字节为单位）。

9~21 在 text[] 中寻找未使用元素。如果所需的代码段已存在于 text[] 中，将该代码段的参照计数器加 1，并将执行进程指向该 text[] 元素。

22~26 对取得的 text[] 元素进行初始化。

27~28 分配交换空间。malloc() 返回交换磁盘的块编号。

29~30　为了将程序的指令读取至数据区域，执行 expand()、estabur() 更新用户空间。以用户空间的地址 0 为起点，分配与程序文件代码数据相同长度的内存作为数据区域。

31~34　将代码段配置于虚拟地址 0 的位置。u.u_offset[1] = 020 表示跳过程序执行文件的文件头。

36~38　执行 swap()，将（容纳着代码数据的）数据段换出内存。在进行交换处理的过程中保持已设置 SLOCK 标志位的状态。

39~42　将 text[] 元素分配给进程，并设置与代码段相对应的 inode[] 元素的参数。

43　执行 expand()，将数据段压缩至最小长度。

45~52　如果没有任何内存中的进程参照程序文件的代码数据，则将进程（数据段）暂时换出内存。此时，数据段中只包含最低限度的数据（PPDA）。然后执行 swtch() 切换执行进程。当进程再次执行时，从退出 xalloc() 的位置开始继续处理。

53　将代码段参照计数器加 1，该计数器表示内存中的进程对代码段的参照数量。

xfree()

xfree() 用来递减执行进程使用的代码段的参照计数器（代码清单 4-7），包括以内存中的进程为对象的计数器，和以所有进程为对象的计数器。当后者的值为 **0** 时,执行 **mfree()** 释放位于交换空间的代码段, 并释放 **text[]** 中相应的元素。但是，当与代码段相对应的 inode[] 元素的 ISVTX 标志位（Sticky Bit）被设置时，不进行上述的释放处理。

代码清单 **4-7**　**xfree()**（**ken/text.c)**

```
 1 xfree()
 2 {
 3     register *xp, *ip;
 4
 5     if((xp=u.u_procp->p_textp) != NULL) {
 6         u.u_procp->p_textp = NULL;
 7         xccdec(xp);
 8         if(--xp->x_count == 0) {
 9             ip = xp->x_iptr;
10             if((ip->i_mode&ISVTX) == 0) {
11                 xp->x_iptr = NULL;
12                 mfree(swapmap, (xp->x_size+7)/8, xp->x_daddr);
13                 ip->i_flag =& ~ITEXT;
14                 iput(ip);
15             }
16         }
17     }
18 }
```

5　如果执行进程没有使用代码段，则不做任何处理。

7　执行 xccdec()，递减以内存中的进程为对象的的代码段参照计数器。

xccdec()

xccdec() 用来递减以内存中的进程为对象的代码段参照计数器（表 4-5，代码清单 4-8）。当参照计数器为 0 时，执行 mfree() 以释放内存中的代码段。

表 **4-5**　**xccdec()** 的参数

参数	含义
xp	text[] 元素

代码清单 **4-8　xccdec()**（**sys/text.c)**

```
xccdec(xp)
int *xp;
{
    register *rp;

    if((rp=xp)!=NULL && rp->x_ccount!=0)
        if(--rp->x_ccount == 0)
            mfree(coremap, rp->x_size, rp->x_caddr);
}
```

4.3　小结

- 通过交换处理和共享代码段可以有效利用有限的内存。
- proc[0] 定期进行交换处理。proc[0] 被称为调度进程或调度器。
- 每一次交换处理都会持续至不再存在可交换对象为止。
- 代码段在被读取至内存时，在交换空间中也会保留一份相同的数据。反之，数据段在被换入内存时，交换空间中的数据将被释放。
- 代码段拥有两个参照计数器。
- 如果代码段不再被内存中的任何进程参照，将从内存中被释放。
- 如果代码段不再被任何进程参照，将从交换空间中被释放。
- 如果设定了 Sticky Bit，则不会从交换空间中释放代码段。这样可以加快启动速度。
- 代码段首先在交换空间中生成，随后被读取至内存。

第 **III** 部分　中断

周边设备发出的请求以及 CPU 内部特定的事件,以中断或陷入的形式加以处理。执行中的进程将暂停运行,转而处理发生的中断或陷入。待处理完成后,被暂停的进程再次恢复运行。此外,系统调用也是通过陷入的机制加以实现的。第 III 部分主要对以下内容进行介绍。

- 与中断相对应的处理是如何被调用的
- 执行中的进程是如何暂停运行,又是如何被恢复的
- 系统调用是如何实现的

通过阅读本部分的内容,对实现高效处理 CPU 内外发生事件的方法应该会有比较清晰的理解。

第 5 章　中断与陷入

5.1　什么是中断与陷入

什么是中断

中断是指用来实现下述处理的机制：当周边设备发出请求时，暂停 执行中的进程，转而执行与请求相对应的处理。待处理完成后再恢复 被暂停的进程。对中断请求进行处理的函数被称为中断处理函数 。

中断请求包括下述几种类型。

- 块设备处理完成通知
- 终端输入
- 时钟中断

通过中断能够实现如下功能。比如，当某进程在向块设备提出处理请求后，在等待设备处理完成之前,系统可以先处理其他进程 （图 5-1 ）。如果没有中断的机制，提出请求的进程必须定期检查设备是否发出了处理结束的通知。这种方式被称为轮询。与处理器相比，周边设备的处理速度比较慢，因此，轮训所花费的时间也会比较长（图 5-2 ）。

由此可知，中断是一种对进程运行和异步事件进行高效处理的机制 。

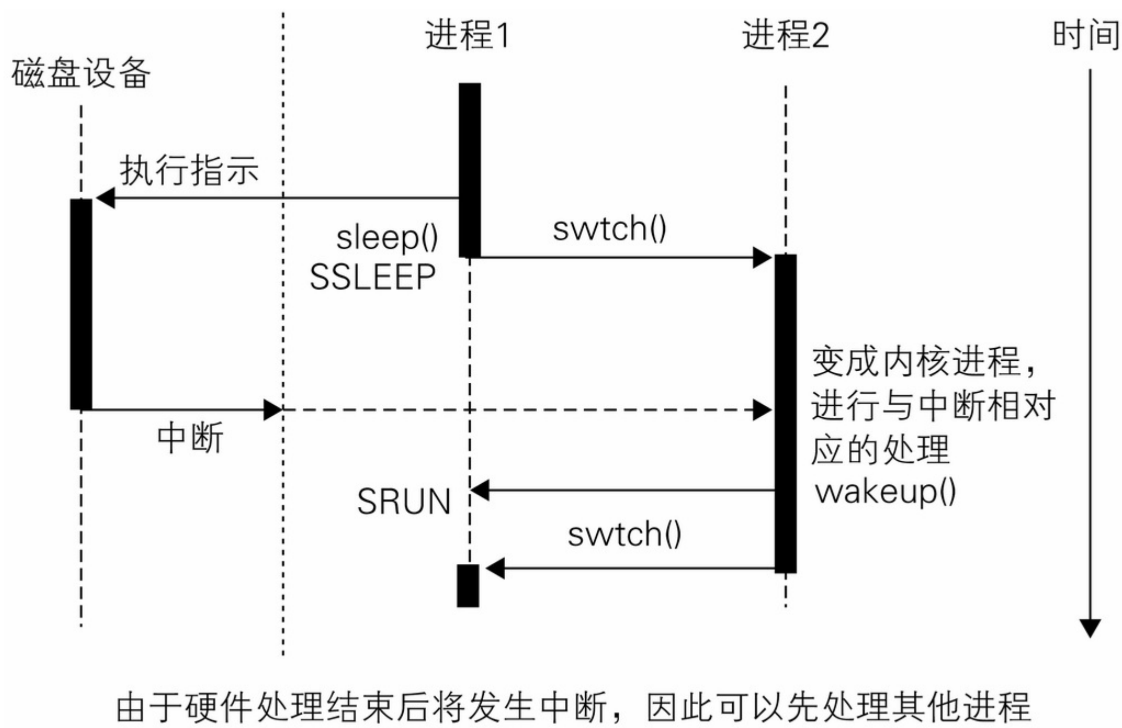


图 **5-1**　中断

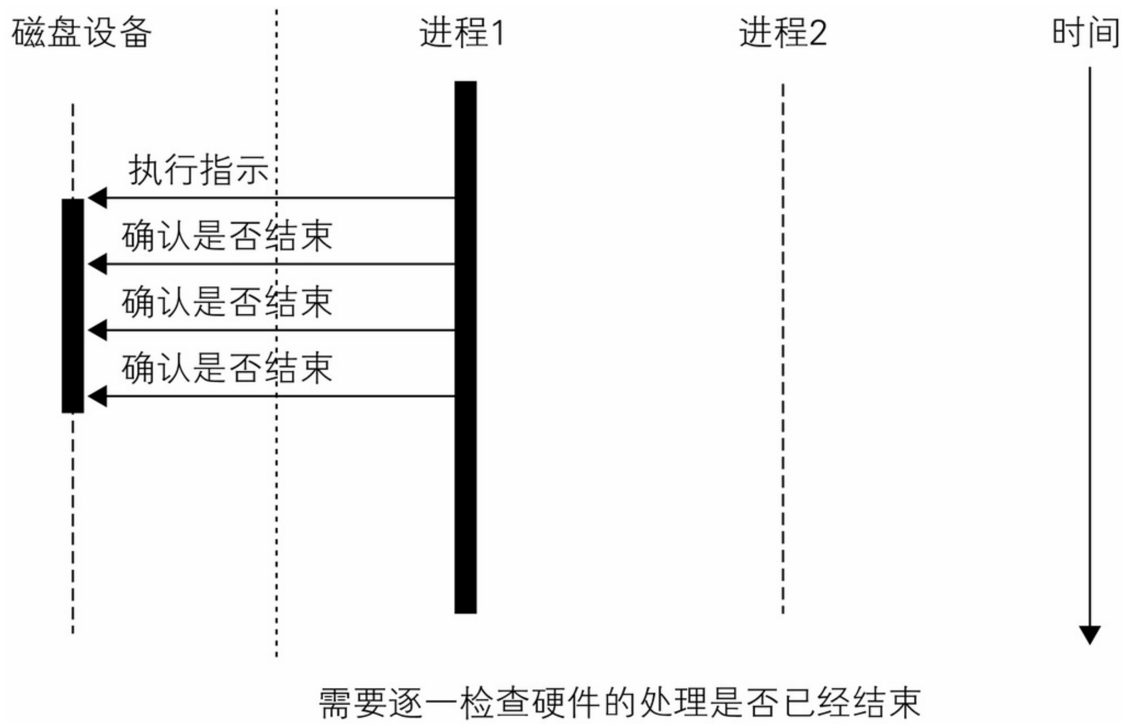


图 **5-2**　轮询

中断处理函数由内核进程负责执行 。在运行用户进程时发生中断的话，则通过硬件切换至内核进程。

被中断的进程的数据暂存于内核栈之中 。当中断处理函数结束后，将恢复内核栈中的数据，并继续处理被中断的进程。

什么是陷入

陷入与前述的中断一样，会引起执行进程的暂停和恢复处理。与中断的不同之处在于，陷入是由 **CPU** 内部的事件引起的 。

当程序执行中发生异常时，会设置 PSW 的陷入位（PSW[4]），表示陷入被触发。异常包括以下情况。

- 被 0 除
- 访问了未被分配的区域
- 总线超时

因为存在陷入机制**,**所以用户程序不必每次都确认异常情况**,**或逐一处理 。当异常出现时会触发陷入，并自动执行共用的处理。当某种陷入发生时，如果希望在对其进行适当处理后继续原来的操作，可以定义独自的陷入处理函数来实现。

由内核进程触发的陷入，通常应由独自定义的陷入处理函数处理。内核程序在执行可能会触发陷入的指令前，将变量 `nofault` 设定为与此陷入相对应的处理函数的地址。当陷入发生时，`nofault` 指向的处理函数将自动执行。

用户程序触发的陷入最终作为信号被处理 （参见第 6 章）。通过对信号设定处理函数，在异常发生时可以进行独自处理。

什么是系统调用

前面已经讲过，用户程序通过系统调用的机制，访问内核提供的各种功能。

由于用户程序不能直接操作内核的功能，因此具有下述优点。

- 可以防止用户程序因无意或有意执行了可能会对系统造成危害的指令
- 用户程序无须了解内核内部进行的复杂处理

当用户程序执行系统调用后，内核进程会进行相应的处理。从用户进程切换到内核进程并进行相应处理的过程,都是利用陷入实现的 。

用户程序执行汇编器的 **sys** 指令 后会触发表示系统调用的陷入。系统调用存在多种类型，通过 `sys` 指令的第 1 个参数指定。有的系统调用还带有自己的参数。参数的指定方法因系统调用种类的不同而不同,具体请参考 **UPM**（**2**）的说明 。

`sys` 指令为汇编器的模拟指令，PDP-11/40 中实际触发陷入的指令为 `trap` 。`trap` 指令的低位比特含有表示陷入种类的值。汇编器将`sys` 指令和第1个参数，编码为`trap` 指令。

5.2　优先级与向量（Vector）

中断优先级和处理器优先级

中断中含有从**0**到**7**的中断优先级。当 **PSW** 中的处理器优先级（**PSW[7-5]**）大于或等于中断优先级时,该中断不会被处理 。周边设备将持续发出中断请求，直到处理器优先级下降，该中断得到处理为止（图 5-3 ）。

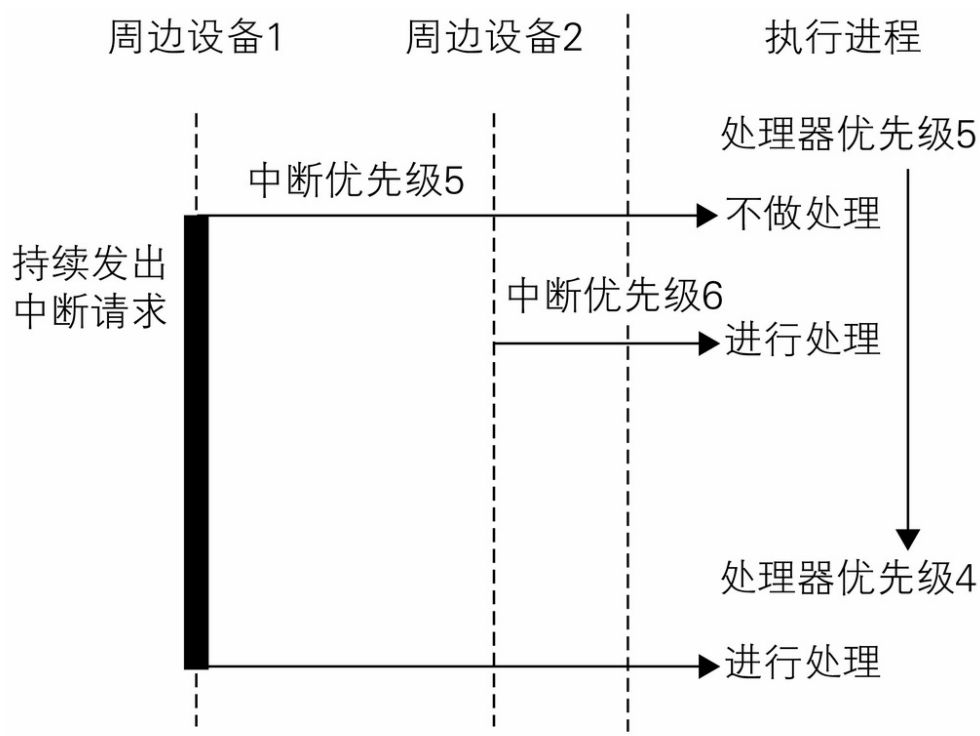


图 **5-3**　中断的掩码

处理器优先级可以通过 `spln()` 函数（n 为整数）进行变更（代码清单 5-1 ）。

代码清单 **5-1**　**spln()**（**conf/m40.s)**

```
.globl     _spl0, _spl1, _spl4, _spl5, _spl6, _spl7
_spl0:
    bic    $340,PS
    rts    pc

_spl1:
    bis    $40,PS
    bic    $300,PS
    rts    pc

_spl4:
_spl5:
    bis    $340,PS
    bic    $100,PS
    rts    pc

_spl6:
    bis    $340,PS
    bic    $40,PS
    rts    pc

_spl7:
    bis    $340,PS
    rts    pc
```

执行中断处理函数时处理器优先级被设置为中断优先级的值，在处理过程中也可以提高处理器优先级。但是用小于中断优先级的处理器优先级执行中断处理函数会导致一些问题 。在运行中断处理函数时，如果再次收到相同类型的中断请求，中断与由中断处理引起的副作用在发生的顺序上会出现逆转。比如说，在终端里输入文字列后，文字列会按照与输入时不同的顺序显示（图 5-4 ）。

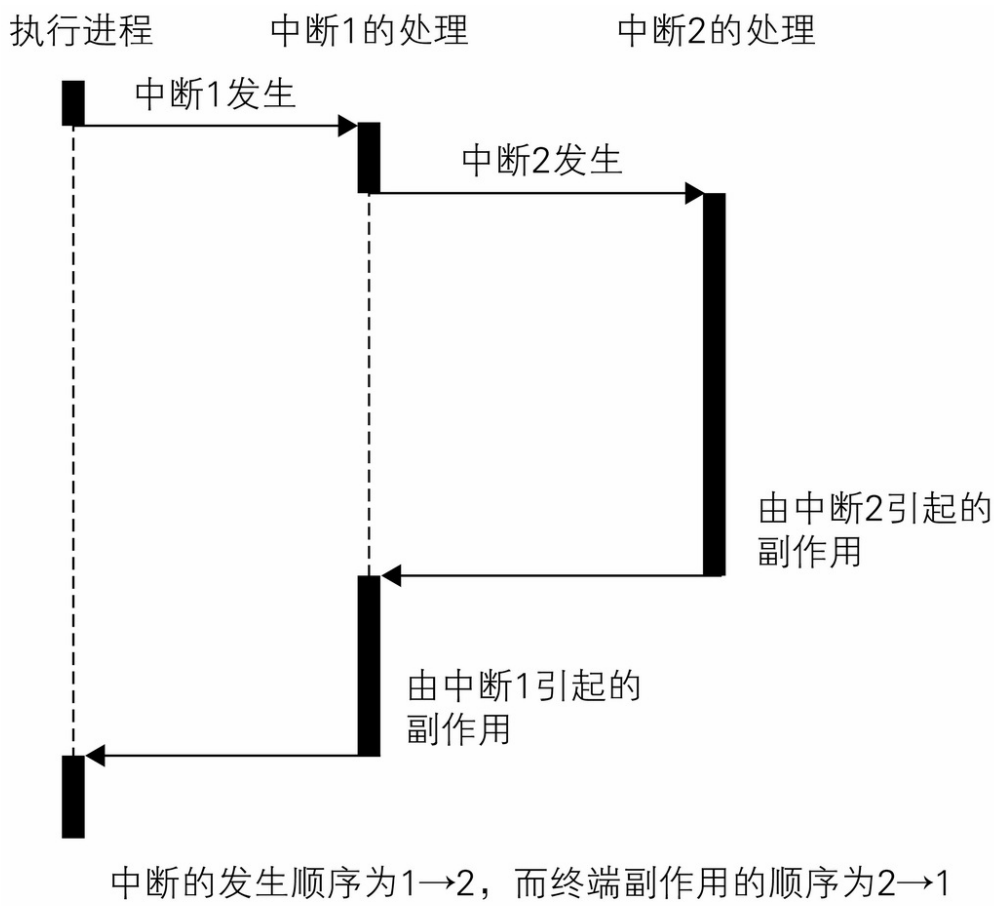


图 **5-4**　中断处理的顺序发生逆转

陷入的优先级相当于中断优先级的 **8**。无论当前处理器优先级的值是多少**,**当陷入发生时都会马上得到处理。

中断和陷入向量

中断和陷入拥有相应的中断和陷入向量。中断和陷入向量是一些预设的**PSW**值和**pc**值的存放地址**,**内核进程在开始处理中断和陷入时会使用这些预设值来设置**PSW**和**pc**寄存器 。

具有代表性的中断和陷入向量如表 5-1、表 5-2 所示。表 5-1 同时显示了中断优先级。

表 **5-1**　中断向量

设备	中断优先级	向量
TTY终端（输入）	4	060
TTY终端（输出）	4	064
电源频率时钟	6	0100
可编程时钟	6	0104

行打印机LP-11	4	0200
块设备RK-11	5	0220

表 **5-2**　陷入向量

种类	向量
总线超时	004
指令错误	010
断点	014
iot（输入输出陷入）	020
电源异常	024
模拟陷入指令	030
sys(trap)指令	034
浮动小数点错误	0244
段错误	0250

5.3　中断和陷入的处理流程

中断和陷入由发生到处理结束基本遵循相同的处理流程（图 5-5 ）。

1. 发生中断或陷入

2. 将当前的 PSW与 pc 保存于内核栈

3. 从向量指定的地址读取 PSW 和 pc

4. 执行`call` 或`trap`

5. 执行中断处理函数或陷入处理函数

6. 从内核栈中恢复PSW 和 pc

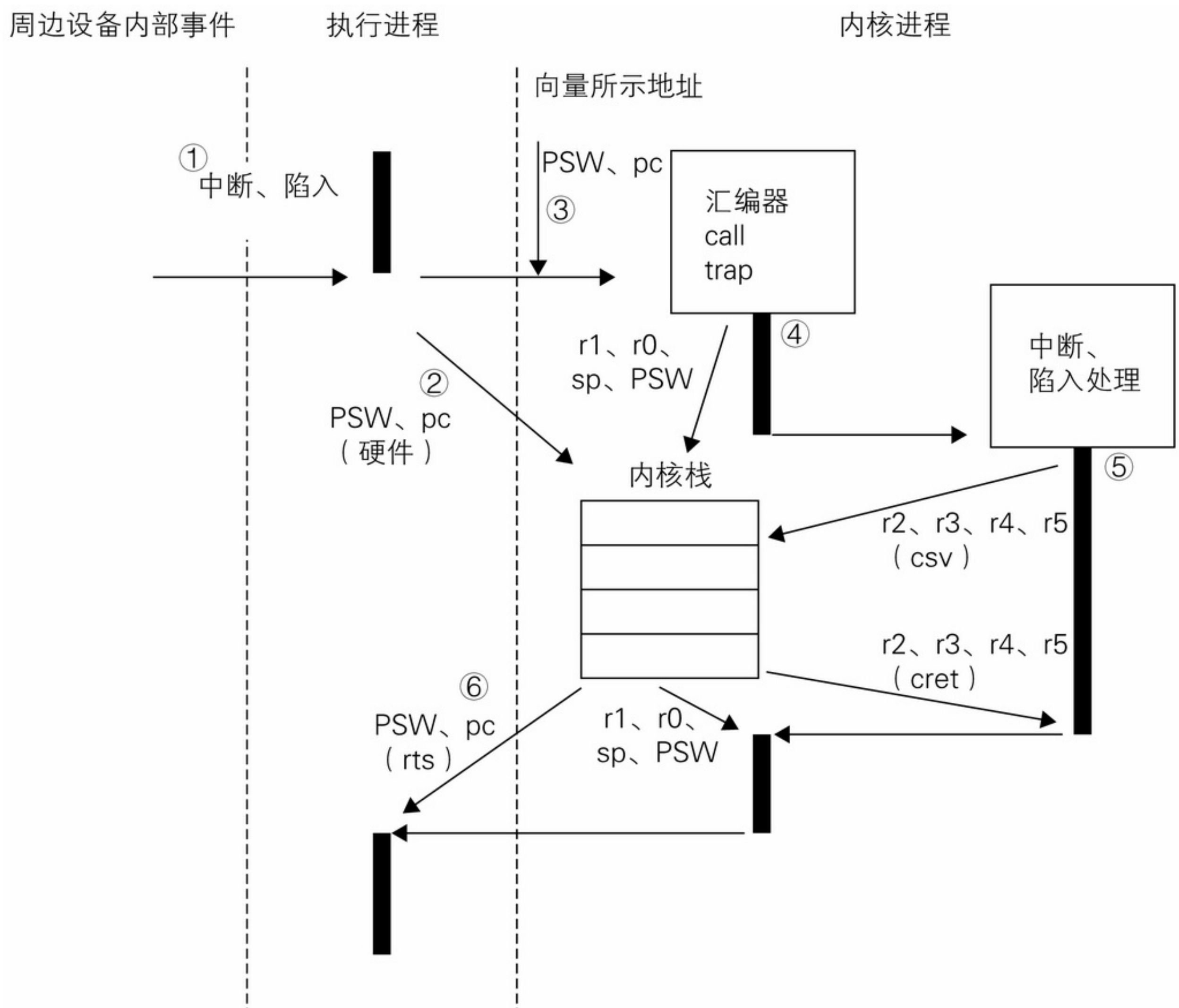


图 **5-5**　中断和陷入

发生中断或陷入

当中断或陷入发生并通知系统后**,**执行进程的 **PSW** 和 **pc** 将被压入内核栈， 然后从与中断或陷入通知一起送来的向量指向的地址中读取并设置 PSW 和 pc。此处理通过硬件完成。

更新 PSW 后，执行进程变为内核进程。将 pc 设置为与此中断或陷入的种类相对应的地址，这使得系统开始执行中断或陷入处理。

接下来看一下位于内核程序低位地址的代码，此处容纳与向量相对应的数据（代码清单 5-2 ）。

代码清单 **5-2**　内核低位地址（**unix/low.s)**

```
/ low core

br4 = 200
br5 = 240
br6 = 300
br7 = 340

. = 0^.
```

```
                br      1f
                4

/ trap vectors
                trap; br7+0.    / bus error
                trap; br7+1.    / illegal instruction
                trap; br7+2.    / bpt-trace trap
                trap; br7+3.    / iot trap
                trap; br7+4.    / power fail
                trap; br7+5.    / emulator trap
                trap; br7+6.    / system entry

. = 40^.
.globl  start, dump
1:      jmp     start
                jmp     dump

. = 60^.
                klin; br4
                klou; br4

. = 100^.
                kwlp; br6
                kwlp; br6

. = 114^.
                trap; br7+7.    / 11/70 parity

. = 214^.
                tcio; br6

. = 224^.
                tmio; br5

. = 240^.
                trap; br7+7.    / programmed interrupt
                trap; br7+8.    / floating point
                trap; br7+9.    / segmentation violation

. = 254^.
                hpio; br5

//////////////////////////////////////////////////////
/                interface code to C
//////////////////////////////////////////////////////

.globl  call, trap

.globl  _klrint
klin:   jsr    r0,call; _klrint
.globl  _klxint
klou:   jsr    r0,call; _klxint

.globl  _clock
kwlp:   jsr    r0,call; _clock

.globl  _tcintr
tcio:   jsr    r0,call; _tcintr

.globl  _tmintr
tmio:   jsr    r0,call; _tmintr

.globl  _hpintr
hpio:   jsr    r0,call; _hpintr
```

发生中段时的处理

假设当前发生了中断向量为 0100 的时钟中断，下面以此为前提进行说明。

第 30 行的 `. = 100^.` 表示当程序被读入内存时，此处的地址为 0100。第 31 行的两个字 `kwlp` 和 `br6` 分别保存在 pc 和 PSW 中。因为 `br6` 的值为 0300，所以处理器优先级 PSW[7-5] 的值为 6。由于将 pc 设定为 `kwlp`，因此中断处理从 `kwlp` 标签的位置开始执行。

`kwlp` 标签所在的第 63 行的 `jsr` 指令首先被执行。`jsr r0, call; _clock` 将 `r0` 的值压入栈，并将 pc 设定为 `call` 标签指向的地址。然后将 r0 设定为 `_clock` （`clock()` 函数）的地址。此处 `r0` 指向的函数即为中断种类对应的中断处理函数（表 5-3，表 5-4 ）。

表 **5-3**　内核栈的状态

sp	值
->	旧的 r0
	被中断进程的 pc
	被中断进程的 PSW

表 **5-4**　寄存器的状态

寄存器	值
r0	_clock（中断处理函数 clock() 的地址）

发生陷入时的处理

下面考虑一下由 `sys` 指令触发的陷入的情况。`sys` 指令的陷入向量为 034。第 19 行的 `trap`； `br7+6` 分别保存在 pc 和 PSW 中。由于 `br7` 的 值 为 0340，因此处理器优先级 PSW[7-5] 的值为 7。另外，**PSW** 低位的 **3** 比特记录了陷入的种类，此处的值为 6。

中断和陷入处理跳转至由 pc 指向的 `call` 标签或 `trap` 标签处继续运行。

执行 **call** 和 **trap**

用汇编语言编写的 `call` 和 `trap` 分别调用相应的中断处理函数或陷入处理函数（代码清单 5-3）。发生中断时调用保存在 `r0` 中的中断处理函数。发生陷入时，如果 `nofault` 被设定了陷入处理函数，则调用该处理函数，如果未设定，则调用 `trap()` 函数。

代码清单 **5-3**　**call(),trap()**（**conf/m40.s)**

```
.globl     trap, call
.globl     _trap
trap:
    mov    PS,-4(sp)
    tst    nofault
    bne    1f
    mov    SSR0,ssr
    mov    SSR2,ssr+4
    mov    $1,SSR0
    jsr    r0,call1; _trap
    / no return
```

```
12 1:
13     mov    $1,SSR0
14     mov    nofault,(sp)
15     rtt
16
17 .globl     _runrun, _swtch
18 call1:
19     tst    -(sp)
20     bic    $340,PS
21     br     1f
22
23 call:
24     mov    PS,-(sp)
25 1:
26     mov    r1,-(sp)
27     mfpi   sp
28     mov    4(sp),-(sp)
29     bic    $!37,(sp)
30     bit    $30000,PS
31     beq    1f
32     jsr    pc,*(r0)+
33 2:
34     bis    $340,PS
35     tstb   _runrun
36     beq    2f
37     bic    $340,PS
38     jsr    pc,_swtch
39     br     2b
40 2:
41     tst    (sp)+
42     mtpi   sp
43     br     2f
44 1:
45     bis    $30000,PS
46     jsr    pc,*(r0)+
47     cmp    (sp)+,(sp)+
48 2:
49     mov    (sp)+,r1
50     tst    (sp)+
51     mov    (sp)+,r0
52     rtt
```

中断处理的流程

首先来看一下中断处理的流程。处理从 `call` 标签处开始执行。假设此时发生了时钟中断，那么中断处理函数 `clock()` 的地址就会保存在 r0 中。

24~26　将 PSW 与 pc 压入栈。

27　将被中断的进程的 `sp` 压入栈。`mfpi` 指令用来将 PSW 表示的前一模式的虚拟地址空间的值压入当前模式的地址空间的栈顶部。`sp` 和其他的通用寄存器的不同之处在于存在两套，分别供用户模式和内核模式使用。内核进程在访问用户进程的 `sp` 时需要使用特殊的指令。

28　将在第 24 行压入栈的 PSW 再次压入栈（表 5-5）。

29　将位于栈顶部的 PSW 的除低位 5 比特之外的部分清 0。

30~31　检查被中断的进程的模式，如果为内核模式则跳转到第 44 行。

32　如果被中断的进程为用户进程，则使用 `jsr` 指令跳转到 `r0` 指向的地址。然后将当前 pc 保存于栈中，作为从中断处理函数返回时的地址（表 5-6）。

表 **5-5**　栈的状态

sp	值
->	PSW
	被中断的进程的 sp
	旧的 r1
	PSW
	旧的 r0
	被中断的进程的 pc
	被中断的进程的 PSW

表 **5-6** 栈的状态

sp	值
->	pc（从中断处理函数返回时的地址）
	经过掩码的 PSW
	被中断的进程的 sp
	旧的 r1
	PSW
	旧的 r0
	被中断的进程的 pc
	被中断的进程的 PSW

34 从中断处理函数返回后进行的处理。将处理器优先级设置为 7 防止发生中断。

35~36 如果未设置标志变量 `runrun`[1]，则跳转到第 40 行。

37~38 如果设置了标志变量 `runrun`，则将处理器优先级重置为 0。由于与执行进程（被中断的进程）相比存在执行优先级更高的进程，因此执行 `swtch()` 切换执行进程。

39 当此进程再次执行时，返回第 33 行的位置再次检查标志变量 `runrun`。

41 如果未设置标志变量 `runrun`，被中断的进程将再次获得控制权。保存于栈中的经过掩码的 PSW

会被忽略。

42 恢复保存于栈中的被中断的进程的 sp。`mtpi` 指令用来复制前一处理器模式（由 PSW 表示）虚拟地址空间中栈顶部的值。

43 跳转到第 48 行。

49~51 恢复保存在栈内的 `r1` 和 `r0`，忽略 PSW（表 5-7）。

52 执行 `rtt` 指令，恢复保存在栈中的被中断的进程的 PSW 和 pc，并继续被中断的进程的处理。`rtt` 指令用来将栈顶部两个字长的数据设置到 PSW 和 pc 中。

45~46 被中断的进程为内核进程时，将前一模式设置为用户模式，然后跳转到由 `r0` 指向的中断处理函数的位置。

47 从中断处理函数返回后，忽略栈顶部的经过掩码的 PSW 以及被中断的进程的 sp。之后的处理与用户进程被中断时的处理相同（表 5-8）。

[1] `runrun` 是全局进程调度标志。操作系统会在某些情况（比如在发现有优先级更高的进程可以上台时）下置 `runrun`，要求在合适的时机（比如此处）进行进程调度。——审校者注

表 **5-7** 栈的状态

sp	值
	pc（从中断处理函数返回时的地址）
	经过掩码的 PSW
	被中断的进程的 sp
	旧的 r1
	PSW
	旧的 r0
->	被中断的进程的 pc
	被中断的进程的 PSW

表 **5-8** 栈的状态

sp	值
	pc（从中断处理函数返回时的地址）
	经过掩码的 PSW

	被中断的进程的 sp
->	旧的 r1
	PSW
	旧的 r0
	被中断的进程的 pc
	被中断的进程的 PSW

陷入处理的流程

接下来看一下陷入处理。处理从 `trap` 标签的位置开始执行。

4　将 PSW 复制到距栈顶部两个字长的位置。因为 PSW 包含着与陷入种类相关的信息，因此需要马上保存（表 5-9）。

5~6　检查 `nofault`，如果已设置了 `nofault`，则跳转到第 12 行。

7~10　如果未设置 `nofault`，则保存 SR0 和 SR2 的当前状态，初始化 SR0 之后使用 `jsr` 指令跳转到 `call1` 的位置。`r0` 被压入栈，它保存着 `trap()` 的地址（表 5-10）。

表 **5-9**　栈的状态

sp	值
	包含陷入种类信息的 PSW
->	被中断的进程的 pc
	被中断的进程的 PSW

表 **5-10**　栈的状态

sp	值
	包含陷入种类信息的 PSW
->	r0
	被中断的进程的 pc
	被中断的进程的 PSW

19~20　将 `sp` 向上移动一个位置，并将处理器优先级设置为 `0` 。

21　跳转至第 25 行。从此处开始的处理与中断处理时相同。因为 r0 保存着 `trap()` 的地址，所以可以通过第 32 行或第 46 行的 `jsr` 指令跳转到 `trap()` 的位置。

13~14　如果已设置 `nofault` ，则初始化 SR0，然后将保存在栈顶部被中断的进程的 pc 值用 `nofault` 的值替换（表 5-11）。

15　使用 `rtt` 指令恢复栈顶部的 PSW 和 pc。将 pc 设置为 `nofault` 的值，控制权转交给由 `nofault` 指定的陷入处理函数。陷入处理函数通过 `rts` 指令终止运行。栈顶部保存着从触发陷入的函数返回时的地址，因此将返回至该地址（表 5-12）。

表 **5-11**　栈的状态

sp	值
	包含陷入种类信息的 PSW
->	nofault
	被中断的进程的 PSW

表 **5-12**　栈的状态

sp	值
	包含陷入种类信息的PSW
	nofault
	被中断的进程的PSW
->	pc（从陷入发生位置返回时的地址）

5.4　时钟中断处理函数

本节以时钟中断处理函数为例介绍中断处理函数。此外，块设备的中断处理函数的介绍请见第 8 章，字符设备的中断处理函数请参考第 12 章，终端的中断处理函数也会在第 13 章中进行介绍。

时钟设备的规格

时钟设备是用来定期向系统发出中断请求的设备 。系统通过时钟中断对时间等进行管理。时钟中断的间隔被称为 **tick** 。

UNIX V6 可以使用两种时钟设备。两种设备中的某一种必须处于可用状态，两者都可用的时候 KW11-L 优先。

- 通过电源频率生成时钟的 KW11-L
- 可编程时钟 KW11-P

KW11-L 通过电源频率生成时钟。时钟频率依赖电源频率，为 60Hz 或 50Hz（表 5-13）。

表 **5-13　KW11-L** 的寄存器

比特位	含义
7	此位由时钟设备置 1 后将引发中断，通过程序可以将其清 0
6	中断有效位。置 1 时中断有效，清 0 时不引发中断

KW11-P 为可编程的时钟设备。时钟发生的间隔可通过程序设定。当内部计数器发生向上或向下溢出时引发中断（表 5-14 ）。

表 **5-14　KW11-P** 的寄存器

比特位	含义
15	错误位
7	计数器向上或向下溢出时置1，并引发中断
6	中断有效位。置 1 时中断有效，清 0 时不引发中断
5	维护时使用。用来停止计数器
4	1：递增计数器，0：递减计数器
3	1：重复时钟中断，0：只引发1 次时钟中断
2~1	时钟中断间隔。00：100KHz，01：10KHz，10：电源频率，11：由外部设备输入引发
0	启动计数器

系统管理者通过 `param.h` 中的 `HZ` 来设定时钟频率，并根据需要重新构筑系统内核（代码清单 5-4 ）。

代码清单 **5-4　HZ（param.h)**

```
#define HZ 60
```

以下内容以时钟频率等于 60Hz 为前提进行说明。

时钟中断处理函数的内容

时钟中断是定期发生的，时钟中断处理函数在中断发生时会进行下述处理，同时以 1 秒和 4 秒的周期进行相应的处理（图 5-6 ）。

- 时钟中断发生时的处理
    - 再次设定时钟设备的寄存器
    - 在指定时刻执行事先登录的函数
    - 递增 CPU 时间
- 以 1 秒为周期进行的处理
    - 时间处理
    - 唤起通过系统调用 sleep 的进程
    - 对进程的执行优先级进行再计算
    - 尝试对进程进行再调度
    - 对信号进行处理
- 以 4 秒为周期进行的处理
    - lightning bolt

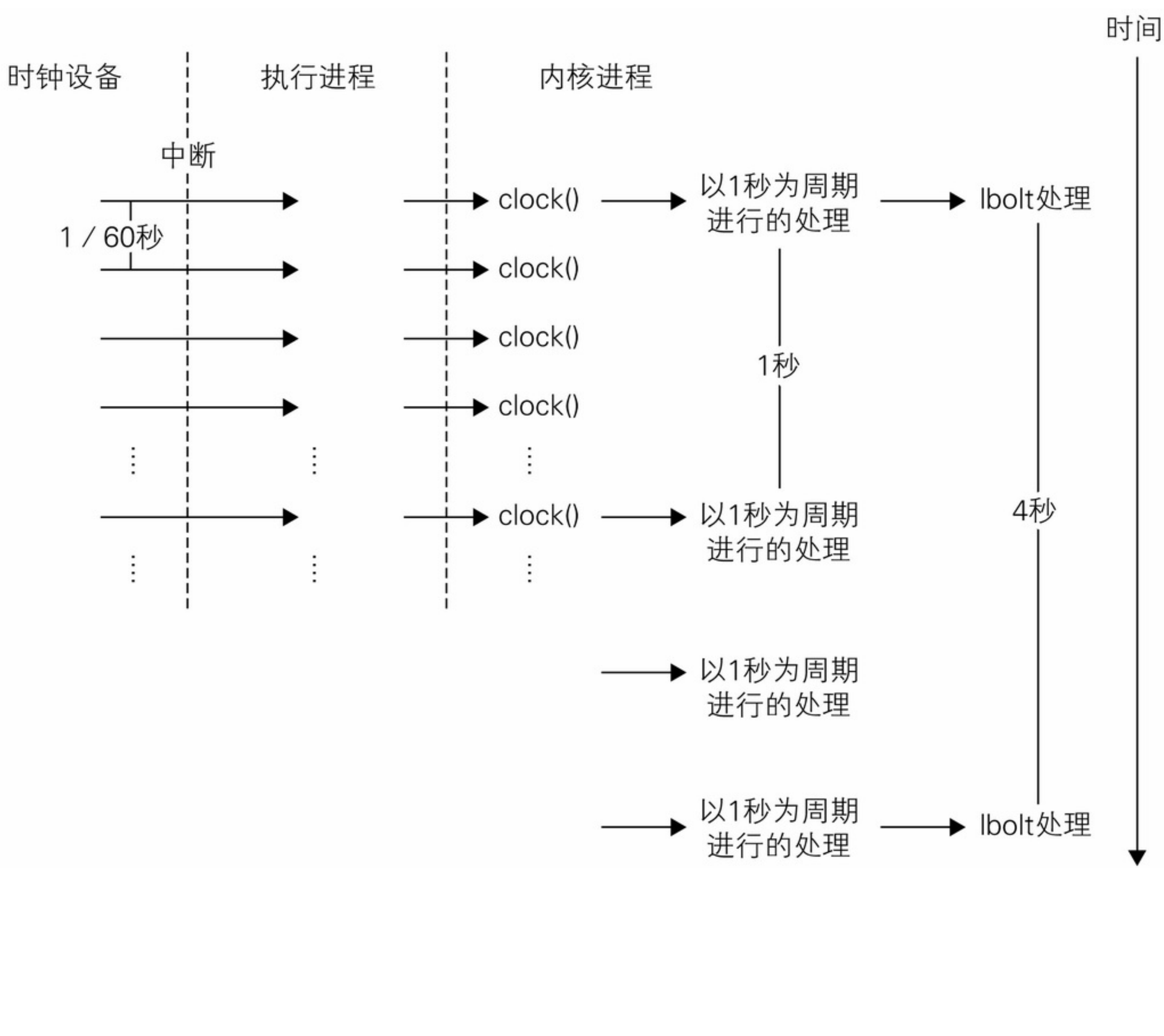


图 **5-6**　时钟中断

时钟设备寄存器的再设定

清除时钟设备寄存器中的中断发生位，使时钟中断可以再次被触发。

定时执行

内核通过 `timeout()` 函数可以指定某个函数在某个时钟 tick 后执行。时钟中断处理函数在上述指定的时刻执行该函数。内核利用 `callo` 结构体的数组 `callout[]` 管理需要定期执行的函数（代码清单 5-5，代码清单 5-6，表 5-15 ）。

代码清单 **5-5**　**callo**（**system.h)**

```
struct    callo
{
    int    c_time;
    int    c_arg;
    int    (*c_func)();
} callout[NCALL];
```

代码清单 **5-6**　**NCALL**（**param.h)**

```
#define    NCALL    20
```

表 **5-15**　**callo** 结构体

成员	含义
`c_time`	函数执行时间（tick）。距前一元素的相对时间
`c_arg`	函数的参数
`(*c_func)()`	指向定时执行的函数的指针

`callo` 结构体表示，在指定时间（`c_time` ）后执行指定函数 `c_func` ，并将 `c_arg` 作为参数传给该函数。`callout[]` 的元素按照执行顺序排列，执行过的元素（`callo` 结构体）将从 `callout[]` 中移除。`callo` 结构体的 `c_time` 不是以绝对时间的形式，而是以距前一个 `callo` 结构体的相对时间的形式设定的（图 5-7 ）。

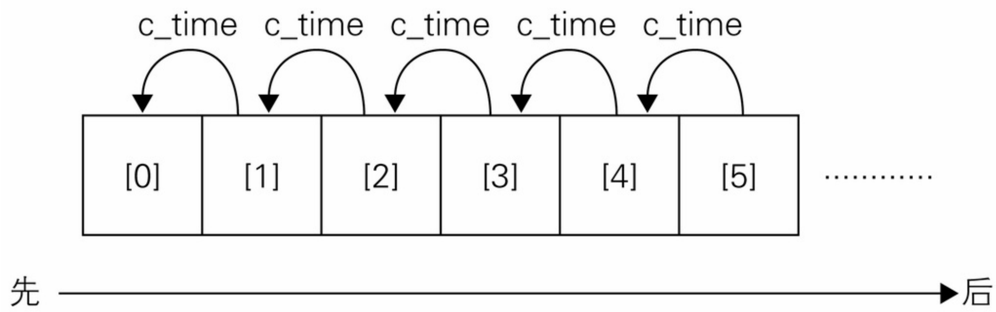


图 **5-7　callout[]**

`timeout()` 函数用来指定在某一时间执行某个函数。它向 `callout[]` 追加元素并使其按照执行顺序排列（表 5-16，代码清单 5-7 ）。

表 **5-16　timeout()** 的参数

参数	含义
`fun`	执行对象的函数
`arg`	函数的参数
`tim`	函数执行时间（tick）

代码清单 **5-7　timeout（ken/clock.c)**

```
timeout(fun, arg, tim)
{
    register struct callo *p1, *p2;
    register t;
    int s;

    t = tim;
    s = PS->integ;
    p1 = &callout[0];
    spl7();
    while(p1->c_func != 0 && p1->c_time<= t) {
        t =- p1->c_time;
        p1++;
    }
    p1->c_time =- t;
    p2 = p1;
    while(p2->c_func != 0)
        p2++;
    while(p2 >= p1) {
        (p2+1)->c_time = p2->c_time;
        (p2+1)->c_func = p2->c_func;
        (p2+1)->c_arg = p2->c_arg;
        p2--;
    }
    p1->c_time = t;
    p1->c_func = fun;
    p1->c_arg = arg;
    PS->integ = s;
}
```

10　将处理器优先级提升至 7 防止发生时钟中断。因为时钟中断处理函数的内部也会操作 `callout[]` ，为了避免与 `timeout` 下面的处理发生冲突，进行了上述处理。

11~14　从 `callout[]` 的起始位置遍历数组，跳过那些执行时间早于参数指定时间的元素，同时用参数指定的时间减去 `callo.c_time` ，得到距前一元素的相对时间。

15　调整排在追加元素之后的元素的相对时间。

16~18　将指针移动至最终有效元素之后的位置。

19~24　将排在追加元素之后的元素向后依次移动一个位置。

25~27　追加一个元素。

28　恢复处理器优先级。

系统调用 **sleep**

系统调用`sleep` 使执行进程进入睡眠状态直至指定时间。到达指定时间后通过时钟中断处理函数唤醒该进程。`sslep()` 为系统调用 `sleep` 的处理函数（表 5-17，代码清单 5-8）。

表 **5-17**　系统调用 **sleep** 的参数

参数	含义
r0	睡眠时间（秒）

代码清单 **5-8**　**sslep()**（**ken/sys2.c)**

```
sslep()
{
    char *d[2];

    spl7();
    d[0] = time[0];
    d[1] = time[1];
    dpadd(d, u.u_ar0[R0]);

    while(dpcmp(d[0], d[1], time[0], time[1]) > 0) {
        if(dpcmp(tout[0], tout[1], time[0], time[1]) <= 0 ||
           dpcmp(tout[0], tout[1], d[0], d[1]) > 0) {
            tout[0] = d[0];
            tout[1] = d[1];
        }
        sleep(tout, PSLEP);
    }
    spl0();
}
```

时间处理

时钟中断处理函数以 1 秒的周期递增表示时间的变量 `time` 。

递增 **CPU** 时间

CPU 时间表示某个进程占用 CPU 的时间。执行进程的 `proc.p_cpu` 在发生时钟中断时递增。用户模式和内核模式下占用 CPU 的时间是分别进行管理的，发生时钟中断时，如果为用户模式则递增 `user.u_utime`，如果为内核模式则递增 `user.u_stime`。

再计算执行优先级

以 1 秒为周期递增全部进程的 `proc.p_time`。`proc.p_time` 表示该进程在内存或交换空间停留的时间，对计算执行优先级有很大的影响。随后执行 `setpri()` 重新计算进程的执行优先级。

尝试对进程进行再调度

如果内存或交换空间中刚刚移入的进程导致交换处理无法继续进行，则唤醒调度器尝试对进程进行再调度。

信号处理

如果执行进程为用户进程，并且已接收到信号，则对信号进行处理。

lightning bolt

时钟中断处理函数以 4 秒为周期，将 `proc.p_wchan` 设置为 lbolt 变量（代码清单 5-9 ）的地址，并唤醒入睡的进程。此处理被称为 lightning bolt。

代码清单 **5-9　lbolt**（**system.h)**

```
int    lbolt;
```

对于不存在使其再次运行的事件，并且待机时间亦不明确的内核进程而言，可以利用 lightning bolt 进行短时间的睡眠。

clock()

`clock()` 为时钟中断处理函数（代码清单 5-10 ）。执行权转交给 `clock()` 时，栈的状态如表 5-18 所示。

表 **5-18**　执行权转交给 **clock()** 时栈的状态

sp	值
->	pc（从中断处理函数返回时的地址）
	经过掩码的 PSW
	被中断的进程的 sp
	旧的 r1
	PSW
	旧的 r0

	被中断的进程的 pc
	被中断的进程的 PSW

在函数的起始位置首先执行 `csv` 。r5、r4、r3、r2 被压入栈（表 5-19 ）。

表 **5-19**　在 **clock()** 内执行 **csv** 后栈的状态以及**fl clock()** 的参数的对应关系

sp	值	参数
->	pc	
	旧的r2	
	旧的r3	
	旧的r4	
r5	旧的r5	
	pc（从中断处理函数返回时的地址）	
	经过掩码的 PSW	dev
	被中断的进程的 sp	sp
	旧的 r1	r1
	PSW	nps
	旧的 r0	r0
	被中断的进程的 pc	pc
	被中断的进程的 PSW	ps

传递给中断处理函数的参数为通过 `call` 、`trap` 处理压入栈的值，距离栈中 `r5` 所在位置两个字长（`dev` 至 `ps` ）。

代码清单 **5-10**　**clock()**（**ken/clock.c)**

```
clock(dev, sp, r1, nps, r0, pc, ps)
{
    register struct callo *p1, *p2;
    register struct proc *pp;

    /* 对时钟设备的寄存器进行再设定 */
    *lks = 0115;
```

```

    display();

    /* callout处理 */
    if(callout[0].c_func == 0)
        goto out;
    p2 = &callout[0];
    while(p2->c_time<=0 && p2->c_func!=0)
        p2++;
    p2->c_time--;

    if((ps&0340) != 0)
        goto out;

    spl5();
    if(callout[0].c_time <= 0) {
        p1 = &callout[0];
        while(p1->c_func != 0 && p1->c_time <= 0) {
            (*p1->c_func)(p1->c_arg);
            p1++;
        }
        p2 = &callout[0];
        while(p2->c_func = p1->c_func) {
            p2->c_time = p1->c_time;
            p2->c_arg = p1->c_arg;
            p1++;
            p2++;
        }
    }

out:
    /* 递增CPU时间 */
    if((ps&UMODE) == UMODE) {
        u.u_utime++;
        if(u.u_prof[3])
            incupc(pc, u.u_prof);
    } else
        u.u_stime++;
    pp = u.u_procp;
    if(++pp->p_cpu == 0)
        pp->p_cpu--;

    /* 以1秒为周期的处理 */
    if(++lbolt >= HZ) {
        if((ps&0340) != 0)
            return;
        lbolt =- HZ;
        /* 递增时间 */
        if(++time[1] == 0)
            ++time[0];
        spl1();
        /* 唤醒通过系统调用sleep进入睡眠状态的进程 */
        if(time[1]==tout[1] && time[0]==tout[0])
            wakeup(tout);
        /* lightning bolt */
        if((time[1]&03) == 0) {
            runrun++;
            wakeup(&lbolt);
        }
        /* 递增proc.p_time，并对执行优先级进行再计算 */
        for(pp = &proc[0]; pp < &proc[NPROC]; pp++)
        if (pp->p_stat) {
            if(pp->p_time != 127)
                pp->p_time++;
            if((pp->p_cpu & 0377) > SCHMAG)
                pp->p_cpu =- SCHMAG; else
                pp->p_cpu = 0;
            if(pp->p_pri > PUSER)
                setpri(pp);
        }
        /* 尝试对进程进行再次调度 */
        if(runin!=0) {
```

```
80            runin = 0;
81            wakeup(&runin);
82        }
83        /* 信号处理 */
84        if((ps&UMODE) == UMODE) {
85            u.u_ar0 = &r0;
86            if(issig())
87                psig();
88            setpri(u.u_procp);
89        }
90    }
91 }
```

7　这种写法可以正确设置 KW11-L 和 KW11-P 两种时钟设备的寄存器。

9　display() 在 PDP-11/40 的环境下不做任何处理。

12~13　如果 callout[] 中不存在需要执行的元素，则跳转到 out 。

14~17　对 callout[] 中未到执行时间的元素，递减其中首个元素 callo.c_time 。因为 callo.c_time 为距前一个元素的相对时间，所以如果改变了第一个元素的值，对排在后面的所有元素都会产生影响。

19~20　被中断的进程的处理器优先级如果不为 0，就不处理 callout[] ，跳转到 out 。[2]

23~36　如果 callout[] 中存在 callo.c_time<=0 的元素，那么就执行该元素的 c_func(c_arg) ，并将 callo.c_time>0 的元素向前方移动。

40　UMODE 用来检查 PSW 是否被设置了用户模式（代码清单 5-11）。[3]

[2] 此处的意图是若处理器优先级大于 0，意味着有更紧急的事物处理，则延缓调用 callout[] 中计划的任务。——审校者注

[3] 此处检查 UMODE 的意图是：若先前为用户态，且启用了统计直方图，则要更新统计直方图。但由于本书不涉及统计直方图的内容，请忽略这部分。——审校者注

代码清单 **5-11　UMODE**（**ken/clock.c)**

```
1 #define    UMODE    0170000
```

51　递增 lbolt 。如果已经经过了 1 秒以上的时间，则启动以 1 秒为周期的处理。

52~53　如果被中断的进程的处理器优先级不为 0 则返回。

54　从 lbolt 递减相当于 1 秒的值。

58　将处理器优先级设定为 1。此后的操作需要花费一些时间，因此可以适当降低处理器优先级。

63~66　进行 lightning bolt 处理。以 4 秒为周期唤醒通过 lbolt 进入睡眠状态的进程。设置标志变量 runrun ，表示存在执行优先级较高的进程（即通过 lbolt 进入睡眠状态的进程）。

68~77　对所有存在的进程进行下述处理。递增 proc.p_time ，最大值为 127。调整 proc.p_cpu 使其最大值小于 SCHMAG （代码清单 5-12）。如果执行优先级小于或等于基准值 PUSER （代码清单 5-13），将再次计算执行优先级。

代码清单 **5-12　SCHMAG（ken/clock.c)**

```
1 #define    SCHMAG    10
```

代码清单 **5-13　PUSER（param.h)**

```
1 #define    PUSER    100
```

85　为了让信号处理函数能够访问被中断的进程的寄存器，将 `u.u_ar0` 设定为保存于栈中的 `r0` 的地址。

5.5　陷入处理函数

陷入处理函数根据陷入的种类进行相应的处理。很多情况下通过向自己发送信号，将其后的处理委托给信号处理函数。

trap()

`trap()` 为一般情况下的陷入处理函数（代码清单 5-14 ）。与 `clock()` 相同，将通过 `call` 和 `trap` 保存在栈中的值作为参数（表 5-20 ）。

表 **5-20**　在 **trap()** 内执行 **csv** 后栈的状态，以及**fl trap()** 的参数的对应关系

sp	值	参数
->	pc	
	旧的 r2	
	旧的 r3	
	旧的 r4	
r5	旧的 r5	
	pc（从陷入处理函数返回时的地址）	
	经过掩码的 PSW	dev
	被中断的进程的 sp	sp
	旧的 r1	r1
	PSW	nps

	旧的 r0	r0
	被中断的进程的 pc	pc
	被中断的进程的 PSW	ps

代码清单 **5-14　trap()（ken/trap.c)**

```
trap(dev, sp, r1, nps, r0, pc, ps)
{
    register i, a;
    register struct sysent *callp;

    savfp();
    if ((ps&UMODE) == UMODE)
        dev =| USER;
    u.u_ar0 = &r0;

    switch(dev) {

    default:
        printf("ka6 = %o\n", *ka6);
        printf("aps = %o\n", &ps);
        printf("trap type %o\n", dev);
        panic("trap");

    case 0+USER: /* bus error */
        i = SIGBUS;
        break;

    case 1+USER: /* illegal instruction */
        if(fuiword(pc-2) == SETD && u.u_signal[SIGINS] == 0)
            goto out;
        i = SIGINS;
        break;

    case 2+USER: /* bpt or trace */
        i = SIGTRC;
        break;

    case 3+USER: /* iot */
        i = SIGIOT;
        break;

    case 5+USER: /* emt */
        i = SIGEMT;
        break;

    case 6+USER: /* sys call */
        u.u_error = 0;
        ps =& ~EBIT;
        callp = &sysent[fuiword(pc-2)&077];
        if (callp == sysent) { /* indirect */
            a = fuiword(pc);
            pc =+ 2;
            i = fuword(a);
            if ((i & ~077) != SYS)
                i = 077;    /* illegal */
            callp = &sysent[i&077];
            for(i=0; i<callp->count; i++)
                u.u_arg[i] = fuword(a =+ 2);
        } else {
            for(i=0; i<callp->count; i++) {
                u.u_arg[i] = fuiword(pc);
                pc =+ 2;
            }
        }
```

```
60        u.u_dirp = u.u_arg[0];
61        trap1(callp->call);
62        if(u.u_intflg)
63            u.u_error = EINTR;
64        if(u.u_error < 100) {
65            if(u.u_error) {
66                ps =| EBIT;
67                r0 = u.u_error;
68            }
69            goto out;
70        }
71        i = SIGSYS;
72        break;
73
74    case 8: /* floating exception */
75        psignal(u.u_procp, SIGFPT);
76        return;
77
78    case 8+USER:
79        i = SIGFPT;
80        break;
81
82    case 9+USER: /* segmentation exception */
83        a = sp;
84        if(backup(u.u_ar0) == 0)
85        if(grow(a))
86            goto out;
87        i = SIGSEG;
88        break;
89    }
90
91    psignal(u.u_procp, i);
92
93 out:
94    if(issig())
95        psig();
96    setpri(u.u_procp);
97 }
```

6　savfp() 在 PDP-11/40 的环境下不做任何处理。

7~8　如果触发陷入的进程为用户进程，则设置 dev 的 USER 标志位（= 第 4 比特位，代码清单 5-15）。此后的处理通过此标志位来判断触发陷入的进程为用户进程还是内存进程。

代码清单 **5-15**　**UMODE** 和 **USER**（**ken/trap.c)**

```
#define    UMODE    0170000
#define    USER     020
```

9　将 u.u_ar0 设定为保存于栈中的 r0 的地址。通过 u.u_ar0[Rn] 可以访问触发陷入的进程的 r0~r7，以及 PSW。请注意，保存于栈中的值最终将会恢复至用户进程。Rn 在 reg.h 中被定义（代码清单 5-16）。

代码清单 **5-16**　**Rn**（**reg.h)**

```
#define    R0     (0)
#define    R1     (-2)
#define    R2     (-9)
#define    R3     (-8)
#define    R4     (-7)
#define    R5     (-6)
```

```
7 #define    R6    (-3)
8 #define    R7    (1)
9 #define    RPS   (2)
```

可以看出 Rn 的值与保存于栈中的 `r0` 的相对位置保持一致（表 5-21 ）。

表 **5-21**　在 **trap()** 内执行 **csv** 后栈的状态

sp	值	与 **r0** 的相对距离
->	pc	-10
	旧的 r2	-9
	旧的 r3	-8
	旧的 r4	-7
r5	旧的 r5	-6
	pc（从陷入处理函数返回时的地址）	-5
	经过掩码的 PSW	-4
	被中断的进程的 sp	-3
	旧的 r1	-2
	PSW	-1
	旧的 r0	0
	被中断的进程的 pc	+1
	被中断的进程的 PSW	+2

11　`dev` 的值为陷入种类。根据 `dev` 进行相应的处理。

13~17　在内核进程内发生陷入时的处理。`trap()` 基本没有考虑过要如何处理这种情况，通常是将 `nofault` 指向陷入处理函数，由该函数进行后续处理。因此，此处在输出内核 PAR6、被陷入中断的进程的 PSW 和 `dev` （陷入种类）之后，调用 `panic()` 结束处理。

19~21　`case` 语句中的 `n+USER` 表示用户模式的陷入。`USER` 标志位在第 8 行被设定。如果是总线错误，则将 `i` 设置为信号种类并退出 `switch` 语句。

23~27　发生错误指令时的处理。与总线错误相同，将 `i` 设置为信号种类并退出 `switch` 文。如果引发

错误指令的指令（即 fuiword(pc-2)。pc 指向触发陷入的指令的下一个指令）为 SETD，但同时却并没有设定错误指令的信号（u.u_signal[SIGINS]）时，则忽略此陷入。SETD 为 C 编译器插入到所有程序起始位置的指令（代码清单 5-17）

代码清单 **5-17　SETD**（**ken/trap.c**）

```
1 #define    SETD    0170011
```

29~39　发生错误指令、断点 / 跟踪、iot、emt 时，执行与发生总线错误时相同的处理。

41~72　发生系统调用时的处理，下文会对此做详细说明。

74~76　内核进程触发浮动小数点异常时的处理。发送信号后随即返回。内核进程本身不进行浮动小数点运算。在内核进程中发生了浮动小数点异常，则表示由用户程序执行的浮动小数点运算在执行系统调用，并切换到内核进程后触发了陷入。因为 PDP-11/40 似乎无法在 正确位置触发浮动小数点运算的异常，所以会导致这种现象的发生（图 5-8）。

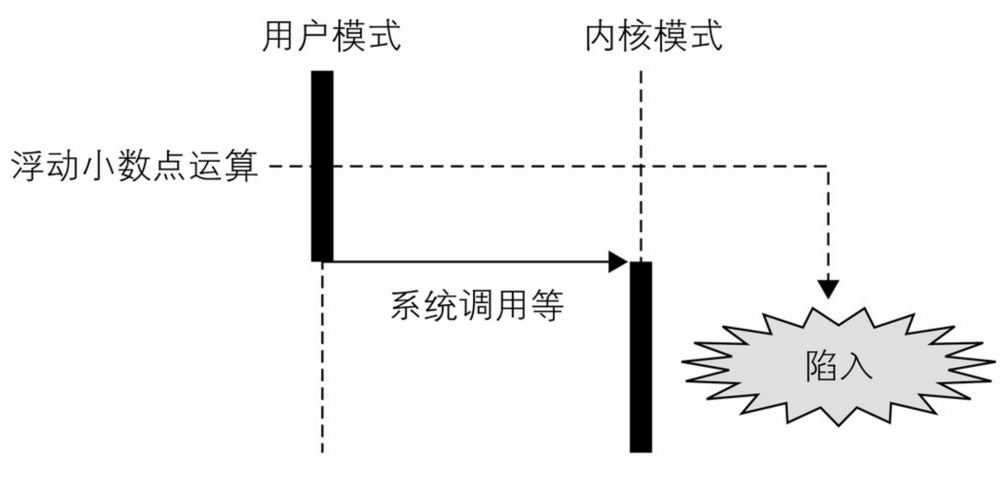


图 **5-8**　内核进程中发生浮动小数点的异常

78~80　用户进程触发浮动小数点异常时的处理。与总线错误相同，将 i 设置为陷入种类并退出 switch 语句。

82~88　用户进程触发段异常时的处理。由于异常的原因有可能是栈区域溢出，因此首先执行 backup() 恢复触发陷入前的状态，再执行 grow() 尝试扩展栈区域。正是此处的处理实现了栈区域的自动扩展 。

91　退出 switch 语句后的处理。执行 psignal() 向执行进程发送信号。

94~95　如果收到信号，则进行信号处理。此前通过 psignal() 发送的信号在此处立即得到处理。

96　执行 setpri()，再次计算进程的执行优先级。

grow()

grow() 用来扩展用户进程栈区域的长度（表 5-22，代码清单 5-18 ）。参数为位于用户空间中的新的栈区域的上限地址。将此地址再加上 20×64 字节即是新的栈区域的长度。如果此地址已在当前栈区域的范围之内，则不做任何处理（图 5-9 ）。

图 **5-9　grow()**

表 **5-22　grow()** 的参数

参数	含义
sp	新的栈区域的地址

代码清单 **5-18　grow()**（**ken/sig.c)**

```
grow(sp)
char *sp;
{
    register a, si, i;

    if(sp >= -u.u_ssize*64)
        return(0);
    si = ldiv(-sp, 64) - u.u_ssize + SINCR;
```

```
9      if(si <= 0)
10         return(0);
11     if(estabur(u.u_tsize, u.u_dsize, u.u_ssize+si, u.u_sep))
12         return(0);
13     expand(u.u_procp->p_size+si);
14     a = u.u_procp->p_addr + u.u_procp->p_size;
15     for(i=u.u_ssize; i; i--) {
16         a--;
17         copyseg(a-si, a);
18     }
19     for(i=si; i; i--)
20         clearseg(--a);
21     u.u_ssize =+ si;
22     return(1);
23 }
```

6~7　如果栈区域已经足够大，则不做任何处理并返回。请注意，由于栈区域的地址从 0xffff 开始，因此为负值。

8~10　计算栈需要扩展的长度，并检查是否为正值，如果不是则返回，且不做扩展处理。SINCR 被定义为 20（代码清单 5-19）。

代码清单 **5-19**　**SINCR**（**param.h)**

```
#define    SINCR    20
```

11~12　执行 estabur() 更新用户 APR 。

13　执行 expand() 扩展数据段。

14~20　移动栈区域，移动的距离为对数据段进行扩展的长度。

21~22　更新栈区域的长度，返回 1 表示扩展成功。

5.6　系统调用的处理流程

传递参数的方法

如前所述，向系统调用传递参数的方法，因系统调用的种类而异。基本上是通过 r0 ，或者是通过在 trap 指令（sys 指令及其参数）之后指定的地址向系统调用传递参数（代码清单 5-20 ）。在系统调用处理函数内部，对前者通过 u.u_ar0[] 、对后者通过 u.u_arg[] 访问。此外也存在两种方式并用的情况。

代码清单 **5-20**　系统调用执行的示例

```
mov $1, r0
sys sys_number1    / 通过r0传递参数
```

```
sys sys_number2    / 在sys指令的后面指定参数
arg
```

系统调用也支持间接执行的方式。编号为 0 的系统调用用来间接执行其他系统调用。在 sys 指令及其参数（=0 ）后面指定数据区域的地址，该地址指向实际执行系统调用的指令（代码清单 5-21 ）。

代码清单 **5-21**　系统调用间接执行的示例

```
.text
    sys 0; sys_call
.data
sys_call:
    sys sys_number; sys_arg
```

sysent 结构体

sysent 结构体保存与系统调用处理函数相关的数据（代码清单 5-22，表 5-23 ）。具体来说，就是保存着参数的数量，和系统调用处理函数的地址。

代码清单 **5-22**　**sysent**（**ken/trap.c)**

```
struct sysent    {
    int count;
    int (*call)();
} sysent[64];
```

表 **5-23**　**sysent** 结构体

成员	含义
count	参数的数量
(*call)()	系统调用处理函数的地址

内核提供的系统调用，和与之对应的系统调用处理函数，都是由 sysent[] 进行管理（代码清单 5-23 ）。sys 指令的参数对应 sysent[] 的下标。

代码清单 **5-23**　**sysent[]**（**ken/sysent.c)**

```
int    sysent[]
{
    0, &nullsys,        /*  0 = indir */
    0, &rexit,          /*  1 = exit */
    0, &fork,           /*  2 = fork */
    2, &read,           /*  3 = read */
    2, &write,          /*  4 = write */
    2, &open,           /*  5 = open */
    0, &close,          /*  6 = close */
    0, &wait,           /*  7 = wait */
    2, &creat,          /*  8 = creat */
    2, &link,           /*  9 = link */
    1, &unlink,         /* 10 = unlink */
    2, &exec,           /* 11 = exec */
```

```
    1, &chdir,          /* 12 = chdir */
    0, &gtime,          /* 13 = time */
    3, &mknod,          /* 14 = mknod */
    2, &chmod,          /* 15 = chmod */
    2, &chown,          /* 16 = chown */
    1, &sbreak,         /* 17 = break */
    2, &stat,           /* 18 = stat */
    2, &seek,           /* 19 = seek */
    0, &getpid,         /* 20 = getpid */
    3, &smount,         /* 21 = mount */
    1, &sumount,        /* 22 = umount */
    0, &setuid,         /* 23 = setuid */
    0, &getuid,         /* 24 = getuid */
    0, &stime,          /* 25 = stime */
    3, &ptrace,         /* 26 = ptrace */
    0, &nosys,          /* 27 = x */
    1, &fstat,          /* 28 = fstat */
    0, &nosys,          /* 29 = x */
    1, &nullsys,        /* 30 = smdate; inoperative */
    1, &stty,           /* 31 = stty */
    1, &gtty,           /* 32 = gtty */
    0, &nosys,          /* 33 = x */
    0, &nice,           /* 34 = nice */
    0, &sslep,          /* 35 = sleep */
    0, &sync,           /* 36 = sync */
    1, &kill,           /* 37 = kill */
    0, &getswit,        /* 38 = switch */
    0, &nosys,          /* 39 = x */
    0, &nosys,          /* 40 = x */
    0, &dup,            /* 41 = dup */
    0, &pipe,           /* 42 = pipe */
    1, &times,          /* 43 = times */
    4, &profil,         /* 44 = prof */
    0, &nosys,          /* 45 = tiu */
    0, &setgid,         /* 46 = setgid */
    0, &getgid,         /* 47 = getgid */
    2, &ssig,           /* 48 = sig */
    0, &nosys,          /* 49 = x */
    0, &nosys,          /* 50 = x */
    0, &nosys,          /* 51 = x */
    0, &nosys,          /* 52 = x */
    0, &nosys,          /* 53 = x */
    0, &nosys,          /* 54 = x */
    0, &nosys,          /* 55 = x */
    0, &nosys,          /* 56 = x */
    0, &nosys,          /* 57 = x */
    0, &nosys,          /* 58 = x */
    0, &nosys,          /* 59 = x */
    0, &nosys,          /* 60 = x */
    0, &nosys,          /* 61 = x */
    0, &nosys,          /* 62 = x */
    0, &nosys           /* 63 = x */
};
```

nullsys() 是不做任何处理的函数，被赋予用于间接执行的 sysent[0]，实际上不会被任何人调用（代码清单 5-24 ）。执行 nosys() 后将触发错误，会被赋予还未使用的 sysent[] 的元素（代码清单 5-25 ）。

代码清单 **5-24　nullsys()**（**trap.c)**

```
nullsys()
{
}
```

代码清单 **5-25　nosys()**（**trap.c)**

```
nosys()
{
    u.u_error = 100;
}
```

trap()

用户程序执行系统调用后，在 `trap()` 中进行的处理如下所示（代码清单 5-26 ）。

代码清单 **5-26　trap()** 中的相应处理（**trap.c)**

```
41     case 6+USER: /* sys call */
42         u.u_error = 0;
43         ps =& ~EBIT;
44         callp = &sysent[fuiword(pc-2)&077];
45         if (callp == sysent) { /* indirect */
46             a = fuiword(pc);
47             pc =+ 2;
48             i = fuword(a);
49             if ((i & ~077) != SYS)
50                 i = 077;    /* illegal */
51             callp = &sysent[i&077];
52             for(i=0; i<callp->count; i++)
53                 u.u_arg[i] = fuword(a =+ 2);
54         } else {
55             for(i=0; i<callp->count; i++) {
56                 u.u_arg[i] = fuiword(pc);
57                 pc =+ 2;
58             }
59         }
60         u.u_dirp = u.u_arg[0];
61         trap1(callp->call);
62         if(u.u_intflg)
63             u.u_error = EINTR;
64         if(u.u_error < 100) {
65             if(u.u_error) {
66                 ps =| EBIT;
67                 r0 = u.u_error;
68             }
69             goto out;
70         }
71         i = SIGSYS;
72         break;
```

41　发生系统调用时的处理。

42　重置 `u.u_error` 。

43　重置触发陷入的进程的 PSW[0]。`EBIT` 的值为 1（代码清单 5-27）。

代码清单 **5-27　EBIT**（**ken/trap.c)**

```
#define    EBIT    1
```

44　fuiword(pc-2) 表示触发陷入的指令（=trap 指令）。trap 指令低位 6 比特的数字代表系统调用的种类。

45　间接系统调用时的处理。

46~50　紧跟着 trap 指令的下一个地址中存放的值也是一个地址，如果该地址指向的数据不是 trap 指令，则将 i 设定为 077（nosys ）。SYS 表示 trap 指令（代码清单 5-28）。

代码清单 **5-28**　**SYS**（**ken/trap.c)**

```
#define    SYS    0104400
```

51　将 callp 设置为实际执行的 sysent[] 的元素。

52~53　将 u.u_arg[i] 赋予位于 trap 指令后方传递给系统调用的参数。

54　直接系统调用时的处理。

55~58　将 u.u_arg[] 赋予位于 trap 指令后方传递给系统调用的参数。

60　将 u.u_dirp 赋予位于 trap 指令后方传递给系统调用的参数的起始位置的值。u.u_dirp 用于在系统调用处理函数内部取得从用户程序传递过来的文件路径名。

61~63　trap1() 用来执行系统调用处理函数（表 5-24，代码清单 5-29）。会在第 6 章中介绍。

表 **5-24**　**trap1()** 的参数

参数	含义
*f	系统调用处理函数的地址

代码清单 **5-29**　**trap1()**（**ken/trap.c)**

```
trap1(f)
int (*f)();
{

    u.u_intflg = 1;
    savu(u.u_qsav);
    (*f)();
    u.u_intflg = 0;
}
```

64　u.u_error 不为 EFAULT 时的处理。未发生错误时这里的 if 语句的值为真。如果 u.u_error 为 EFAULT ，与发生总线错误时相同，将 i 设置为陷入种类并退出 switch 语句。

65~68　如果 u.u_error 被赋予了错误代码，则将触发陷入的进程的 PSW[0] 设为 1，并将 r0 设定为 u.u_error 的值。执行系统调用的程序可以通过 PSW[0] 判断系统调用中是否发生了错误。

69　跳转至 out ，对信号进行处理并再次计算进程优先级，然后结束系统调用。

5.7 小结

- 由于某种原因引发中断和陷入后，暂停当前处理，由内核进程执行相应的处理函数后再恢复被暂停的处理。
- 中断是对周边设备提出的异步中断请求进行有效处理的机制。
- 陷入是对 CPU 内部发生的异常进行有效处理的机制。
- 系统调用是用户程序执行内核功能的手段，利用陷入实现。
- 系统调用由 `sysent[]` 管理。通过 `sysent[0]` 可实现间接执行。

第 6 章　信号

6.1　什么是信号

信号是一种实现进程间通信的机制。收到信号的进程将暂停当前执行的内容，并根据信号种类做相应的处理。根据信号暂停当前处理，并在需要时恢复的特点，可视为中断的一种。

信号可实现以下功能。

- 剥夺用户进程的控制权
- 终止用户进程的运行
- 跟踪用户进程的处理（trace）

当进程成为执行进程后，会检查是否收到信号，如果收到了信号则调用信号处理函数（图 6-1 ）。进程也可以选择忽略信号，或执行独自定义的信号处理函数。

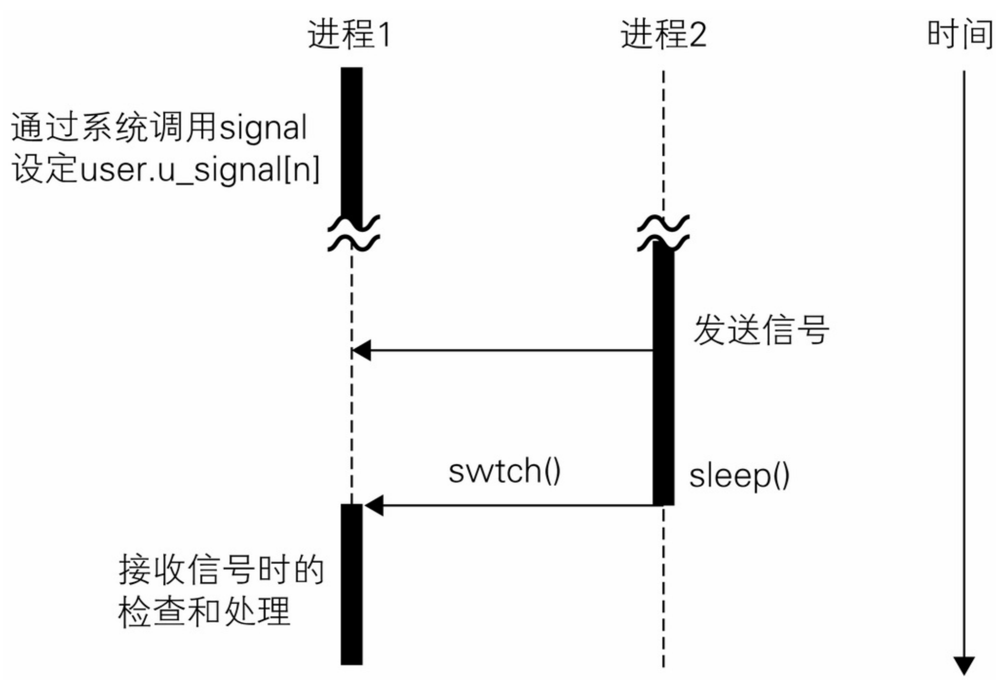


图 **6-1**　信号处理流程

信号的发送方法

用户进程执行系统调用 `kill` 后，即可向其他进程发送信号。此外，也可以通过终端发送信号。

对于收到信号的进程，`proc.p_sig` 会被设为代表信号种类的数值。如果处理信号前又接收到了新的信号，旧的信号将被覆盖 。但是，用于强制结束的 `SIGKIL` 却不会被覆盖。

最近的操作系统一般将信号作为比特向量处理，因此即使收到新的信号也不会覆盖旧的信号。

确认接收信号

内核进程会在下述处理中确认是否收到了信号。如果收到，则执行信号处理函数。

- `proc.p_pri` 大于等于 0，并准备进入休眠状态时
- 时钟中断处理函数以每秒 1 次的频率确认
- 陷入处理函数

执行进程之外的进程无法确认是否收到了信号 ，也就是说，即使向某个进程发送了信号，也不能保证对方会立即进行处理。只有当该进程变为执行进程后才会处理信号。这一点与周边设备引发的中断相比有很大不同。

信号的种类

信号具有多个种类。虽然现在在其他操作系统中可以设定多达 20 种的信号，但是 UNIX V6 只定义了 13 种（代码清单 6-1，表 6-1 ）。

代码清单 **6-1**　信号（**param.h**）

```
#define        NSIG      20
#define        SIGHUP    1
#define        SIGINT    2
#define        SIGQIT    3
#define        SIGINS    4
#define        SIGTRC    5
#define        SIGIOT    6
#define        SIGEMT    7
#define        SIGFPT    8
#define        SIGKIL    9
#define        SIGBUS    10
#define        SIGSEG    11
#define        SIGSYS    12
#define        SIGPIPE   13
```

表 **6-1**　信号的种类

信号	含义
SIGHUP(1)	终止（Hangup）
SIGINT(2)	中断
SIGQIT(3)	终止（Quit）
SIGINS(4)	错误指令
SIGTRC(5)	跟踪、断点
SIGIOT(6)	执行输入输出陷入指令
SIGEMT(7)	执行模拟指令

SIGFPT(8)	浮动小数点计算的异常
SIGKIL(9)	强制结束
SIGBUS(10)	总线错误
SIGSEG(11)	段异常
SIGSYS(12)	对系统调用指定了错误参数
SIGPIPE(13)	终止管道

通过赋值 `user.u_signal[n]`（`n` 代表信号的种类），可选择忽略该信号，或执行独自定义的信号处理函数。`user.u_signal[n]` 的值与收到信号后进行的处理之间的关系如表 6-2 所示。

表 **6-2**　设定 **user.u_signal[]**

user.u_signal[]	处理
0	使进程自我终止
奇数	使进程忽略信号
偶数（信号处理函数的地址[1]）	使进程执行指定的处理函数

[1] 由于 PDP-11/40 中的指令以字为单位（偶数字节单位）对齐，因此函数的地址一定为偶数。

但是，n=9 的 `SIGKIL` 比较特殊。进程无法忽略 **`SIGKIL`** 或执行独自定义的处理函数 。收到 `SIGKIL` 的进程必须结束自身的处理，因此，`user.u_signal[9]` 的值始终为 `0`。

用户进程通过执行系统调用 `signal` 可以设定 `user.u_signal[]` 的值。

ssig()

`ssig()` 为系统调用 `signal` 的处理函数（表 6-3，代码清单 6-2 ）。执行此函数将改变 `u.u_signal[]` 的值。

表 **6-3**　系统调用 **signal** 的参数

参数	含义
u.u_arg[0]	信号的种类
u.u_arg[1]	设定值

代码清单 **6-2**　**ssig()**（**ken/sys4.c)**

```
1 ssig()
2 {
3     register a;
4 
5     a = u.u_arg[0];
6     if(a<=0 || a>=NSIG || a ==SIGKIL) {
7         u.u_error = EINVAL;
8         return;
9     }
10    u.u_ar0[R0] = u.u_signal[a];
11    u.u_signal[a] = u.u_arg[1];
12    if(u.u_procp->p_sig == a)
13        u.u_procp->p_sig = 0;
14 }
```

5~9　信号的种类小于等于 0，或大于等于 `NSIG` (20)，或等于 `SIGKIL` (9) 时判断为出错。

10　将当前 `u.u_signal[a]` 的值保存于用户进程的 `r0`，此数值将成为系统调用 `signal` 的返回值。

11　将 `u.u_signal[a]` 设置为由参数指定的值。

12~13　如果执行中的进程在此之前已经收到了与刚刚设定的 `u.u_signal[a]` 相对应的信号，则将该信号清除。此处大概是开发者认为接收信号时的处理已经发生了变化，需要重置。

kill()

`kill()` 为系统调用 `kill` 的处理函数（表 6-4，代码清单 6-3 ）。通过 `psignal()` 向用户程序指定的进程 ID 相对应的进程发送信号，但是无法对自己发送信号。此外，超级用户以外的用户，如果发送与接收的进程不具备相同的实效 UID（参见第 9 章），也是无法发送信号的。

当指定的进程 ID 为 0 时，向与执行进程位于同一终端、除 `proc[0]` 和 `proc[1]` 之外的所有进程发送信号。如果对象进程不存在，则会引发错误。虽然名为 `kill`，但不只是发送 `SIGKIL` 那么简单。

表 **6-4**　系统调用 **kill** 的参数

参数	含义
`r0`	信号针对的进程ID。如果 r0 为 0，则对象包括除 ID 为 0、1 的进程之外的所有进程
`u.u_arg[0]`	信号的种类

代码清单 **6-3**　**kill()**（**ken/sys4.c)**

```
kill()
{
    register struct proc *p, *q;
    register a;
    int f;

    f = 0;
    a = u.u_ar0[R0];
    q = u.u_procp;
    for(p = &proc[0]; p < &proc[NPROC]; p++) {
        if(p == q)
            continue;
        if(a != 0 && p->p_pid != a)
```

```
            continue;
        if(a == 0 && (p->p_ttyp != q->p_ttyp || p <= &proc[1]))
            continue;
        if(u.u_uid != 0 && u.u_uid != p->p_uid)
            continue;
        f++;
        psignal(p, u.u_arg[0]);
    }
    if(f == 0)
        u.u_error = ESRCH;
}
```

signal()

`signal()` 通过执行 `psignal()`，向与执行进程位于同一终端的所有进程发送信号（表 6-5，代码清单 6-4 ）。`signal()` 由终端的中断处理函数运行。

表 **6-5　signal()** 的参数

参数	含义
tp	tty 结构体（参见第 13 章）
sig	信号的种类

代码清单 **6-4　signal()（ken/sig.c)**

```
signal(tp, sig)
{
    register struct proc *p;

    for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(p->p_ttyp == tp)
            psignal(p, sig);
}
```

psignal()

`psignal()` 向指定的进程发送信号（表 6-6，代码清单 6-5 ）。但是，`SIGKIL` 信号不会被覆盖。

表 **6-6　psignal()** 的参数

参数	含义
p	proc[] 中代表对象进程的元素
sig	信号的种类

代码清单 **6-5　psignal()（ken/sig.c)**

```
 1 psignal(p, sig)
 2 int *p;
 3 {
 4     register *rp;
 5
 6     if(sig >= NSIG)
 7         return;
 8     rp = p;
 9     if(rp->p_sig != SIGKIL)
10         rp->p_sig = sig;
11     if(rp->p_stat > PUSER)
12         rp->p_stat = PUSER;
13     if(rp->p_stat == SWAIT)
14         setrun(rp);
15 }
```

11~12　此处疑为 Bug，处理对象的变量不应为 `proc.p_stat`，而应是 `proc.p_pri`。开发者的意图应该是使执行优先级保持一个定值，从而使信号更容易处理。

13~14　如果对象进程的状态为 `SWAIT` 并处于睡眠之中，将其唤醒并促使其进行信号处理。`SWAIT` 在进程调用 `sleep()` 时进入睡眠状态，且 `proc.p_pri` 大于等于 0 时被设定。进程被唤醒并通过 `swtch()` 成为执行进程后，在 `sleep()` 内将对是否收到信号进行确认。

issig()

`issig()` 用来确认执行进程是否收到了信号（代码清单6-6）。当`user.u_signal[n]`（`n` 表示信号的种类）被设为奇数时，忽略该信号。

代码清单 **6-6　issig()**（**ken/sig.c**）

```
 1 issig()
 2 {
 3     register n;
 4     register struct proc *p;
 5
 6     p = u.u_procp;
 7     if(n = p->p_sig) {
 8         if (p->p_flag&STRC) {
 9             stop();
10             if ((n = p->p_sig) == 0)
11                 return(0);
12         }
13         if((u.u_signal[n]&1) == 0)
14             return(n);
15     }
16     return(0);
17 }
```

7　收到信号时的处理。

8~12　与跟踪功能相关的处理。在后文中将对其进行说明。

13~14　如果 `u.u_signal[n]`（n=proc.p_sig）的值为偶数，则返回 n。

16　如果没有收到信号，或是需要忽略信号时返回 0。

psig()

`psig()` 进行与执行进程的 `proc.p_sig` 相对应的信号处理（代码清单 6-7 ）。当内核进程通过 `issig()` 进行的检查结果为真时被调用。

如果 `user.u_signal[n]` 的值为偶数，执行由该值指向的信号处理函数。首先将用户进程的 PSW、pc 的当前值压入用户进程的栈的顶部，并将栈指针指向存放 pc 的位置（表 6-7 ）。然后清除用户进程 PSW 的陷入位（trap bit），并将 pc 设定为信号处理函数的地址。信号处理函数在控制权返回用户进程后被执行。信号处理函数由 **`rtt`** 或 **`rti`** 指令终止 。`rtt` 和 `rti` 指令从栈顶部恢复 pc 和 PSW 的值，随后被暂停的进程将继续原有的处理。

表 **6-7**　用户栈的状态

sp	值
->	被中断的进程的 pc
	被中断的进程的 PSW
	被中断的进程的栈的顶部

如果 `user.u_signal[n]` 的值为 0，则终止进程。根据信号的种类可以输出调试用的 core 文件。最后将用户进程的 `r0` 的值、信号的种类、是否生成了 core 文件等信息作为结束状态通知父进程。

代码清单 **6-7**　**psig()**（**ken/sig.c)**

```
psig()
{
    register n, p;
    register *rp;

    rp = u.u_procp;
    n = rp->p_sig;
    rp->p_sig = 0;
    if((p=u.u_signal[n]) != 0) {
        u.u_error = 0;
        if(n != SIGINS && n != SIGTRC)
            u.u_signal[n] = 0;
        n = u.u_ar0[R6] - 4;
        grow(n);
        suword(n+2, u.u_ar0[RPS]);
        suword(n, u.u_ar0[R7]);
        u.u_ar0[R6] = n;
        u.u_ar0[RPS] =& ~TBIT;
        u.u_ar0[R7] = p;
        return;
    }
    switch(n) {

    case SIGQIT:
    case SIGINS:
    case SIGTRC:
    case SIGIOT:
    case SIGEMT:
    case SIGFPT:
    case SIGBUS:
    case SIGSEG:
    case SIGSYS:
        u.u_arg[0] = n;
        if(core())
            n =+ 0200;
    }
    u.u_arg[0] = (u.u_ar0[R0]<<8) | n;
```

```
38     exit();
39 }
```

9　`user.u_signal[n]` 被设定为独自定义的信号处理函数地址时的处理。

14　为了安全起见，执行 `grow()` 扩展用户进程的栈区域。

22　`u.u_signal[n]` 被设定为 0 时的处理。

35　如果生成了 core 文件，则将 `n` 的第 7 比特位设置为 1。

37　将 `u.u_arg[0]` 的高位 8 比特设定为用户进程的 `r0` 的值，低位 8 比特设定为表示信号种类以及是否生成了 core 文件的信息。`u.u_arg[0]` 作为结束状态通知父进程。

38　执行 `exit()`，使进程结束自身的处理。

core()

`core()` 在当前目录下生成名为 core 的文件（代码清单 6-8 ）。文件包括数据段的全部内容（PPDA、数据区域、栈区域）。用户可以通过检查该文件调试程序。

在第 1 章中介绍了内存也可以被称为 Core。此处的 `core()` 进行的处理通常被称作内核转储（Core Dump）。

代码清单 **6-8**　**core()**（**ken/sig.c)**

```
core()
{
    register s, *ip;
    extern schar;

    u.u_error = 0;
    u.u_dirp = "core";
    ip = namei(&schar, 1);
    if(ip == NULL) {
        if(u.u_error)
            return(0);
        ip = maknode(0666);
        if(ip == NULL)
            return(0);
    }
    if(!access(ip, IWRITE) &&
        (ip->i_mode&IFMT) == 0 &&
        u.u_uid == u.u_ruid) {
        itrunc(ip);
        u.u_offset[0] = 0;
        u.u_offset[1] = 0;
        u.u_base = &u;
        u.u_count = USIZE*64;
        u.u_segflg = 1;
        writei(ip);
        s = u.u_procp->p_size - USIZE;
        estabur(0, s, 0, 0);
        u.u_base = 0;
        u.u_count = s*64;
        u.u_segflg = 0;
        writei(ip);
    }
    iput(ip);
    return(u.u_error==0);
}
```

在系统调用处理中处理信号

UNIX V6 在系统调用处理中如果处理了信号，将引发 EINTR 错误。此时将恢复接收信号时的进程状态，用户根据需要可以再次执行系统调用。

实际执行系统调用处理函数的 trap1() 的代码如下所示（代码清单 6-9 ）。

代码清单 **6-9**　**trap1()** 中的相关代码（**ken/trap.c)**

```
trap1(f)
int (*f)();
{

    u.u_intflg = 1;
    savu(u.u_qsav);
    (*f)();
    u.u_intflg = 0;
}
```

trap1() 首先设定 u.u_intflg ，在 u.u_qsav 中保存 r5、r6 的当前值，然后执行通过参数取得的系统调用处理函数。执行完毕后重置 u.u_intflg 并返回。

因为 u.u_intflg 被重置，所以在 trap() 中不会发生 EINTR 错误（代码清单 6-10）。

代码清单 **6-10**　**trap()** 中的相关代码（**ken/trap.c)**

```
        trap1(callp->call);
        if(u.u_intflg)
            u.u_error = EINTR;
```

因为系统调用处理中已经适度提高了处理器优先级，所以对终端的输入不会引发中断处理。在系统调用处理中能够引发信号处理的只有这种情况，即系统调用处理函数执行 sleep() 中断自身运行后，执行进程切换至其他进程，而后者向前者发送了信号。

通常，在读取文件等情况下会将执行优先级设为负值并进入睡眠状态，此时信号将被忽略。而例如终端处理等对处理速度较慢的设备执行系统调用 read 、write 或 wait 时，有可能在系统调用处理函数中将执行优先级设为大于或等于 0 的值并进入睡眠状态。只有在这种情况下有可能发生对信号的处理。

收到信号时，会在 sleep() 中将其检出（代码清单 6-11）。sleep() 执行 aretu(u.u_qsav) 后，将从 trap1() 返回处继续进行处理。此时因为 u.u_intflg 未被重置，所以在 trap() 中将发生 EINTR 错误。

代码清单 **6-11**　**sleep()** 中的相关代码（**ken/slp.c)**

```
sleep(chan, pri)
{

  （中略）

    if(pri >= 0) {
        if(issig())
            goto psig;
```

```
                  （中略）

        swtch();
        if(issig())
            goto psig;
    } else {

    （中略）

psig:
    aretu(u.u_qsav);
```

6.2　跟踪功能

什么是跟踪

当子进程处理（由断点陷入触发的）信号时，跟踪功能提供了一种使父进程能够介入子进程处理的机制。通过系统调用 `ptrace`，可以操作子进程的数据。跟踪功能通常供调试器等使用。

ipc 结构体

`ipc`（inter process communication）结构体是供跟踪功能使用的全局结构体（代码清单 6-12，表 6-8）。父进程通过执行系统调用 `ptrace`，可以将对子进程的请求保存在 `ipc` 结构体中。

代码清单 **6-12**　**ipc** 结构体（**ken/sig.c)**

```
struct
{
    int    ip_lock;
    int    ip_req;
    int    ip_addr;
    int    ip_data;
} ipc;
```

表 **6-8**　**ipc** 结构体

成员	含义
`ip_lock`	用于排他处理
`ip_req`	父进程请求的种类
`ip_addr`	处理对象的地址
`ip_data`	传递的数据

跟踪的处理流程

子进程执行系统调用 `ptrace` 设定 `STRC` 标志位。`STRC` 标志位表明当前进程为跟踪对象进程。

如果子进程在设置了 STRC 标志位的情况下进行信号处理，则会进入 SSTOP 状态，控制权会转移至父进程。SSTOP 状态表明当前进程正在等待父进程进行介入处理。

在系统调用 wait 的处理函数中，如果父进程发现了处于 SSTOP 状态的子进程，会将子进程的 SWTED 标志位置 1。

此时控制权会暂时交还给用户进程。父进程如果执行系统调用 ptrace ，将清除 SWTED 标志，并可以把对子进程的请求赋予 ipc 结构体。如果父进程没有执行 ptrace ，而是再次执行系统调用 wait 并试图将控制权交给子进程，SWTED 标志将保持为 1 的状态，系统以此判断父进程无意介入子进程处理，因此将结束跟踪处理。

控制权转交给子进程后，子进程根据 ipc 结构体的值进行相应处理。子进程或是等待父进程的再次介入，或是返回一般处理。

跟踪处理的流程如图 6-2 所示。

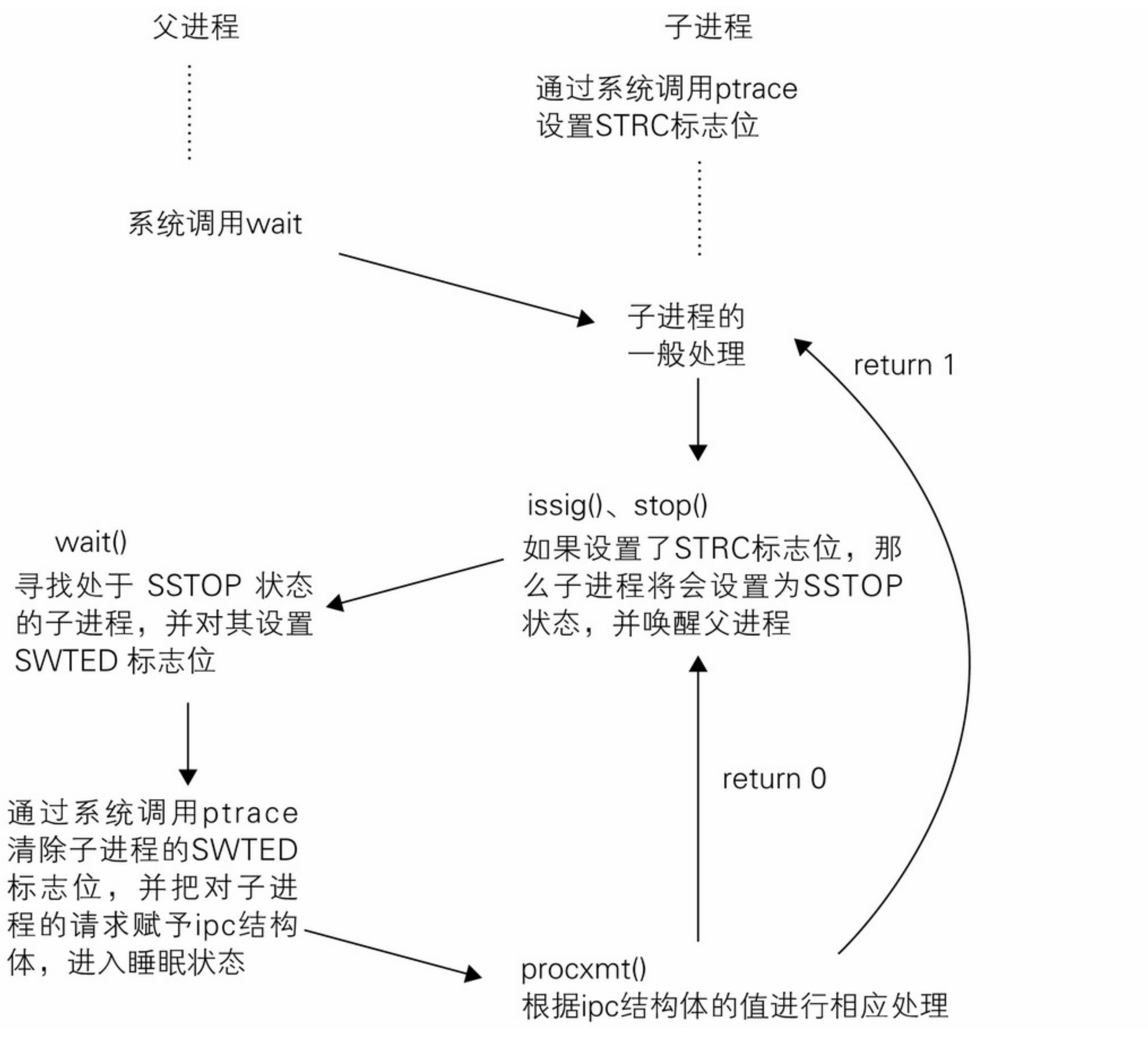


图 **6-2**　跟踪处理流程图

stop()

stop() 将进程设置为 SSTOP 状态（代码清单 6-14 ）。设置后，进程在收到信号，并开始对其进行相应处理时将通知父进程，父进程也可以对子进程发送指令。

当进程的跟踪标志位（STRC ）为 1 时，stop() 由 issig() 调用（代码清单 6-13 ）。issig() 在执行 stop() 后，会检查是否设置了 proc.p_sig 的值，如果未设置则返回 0。这应该是考虑到 stop() 处理中信号有可能被清除而采取的措施。

代码清单 **6-13**　**issig()** 中的相关代码（**ken/sig.c)**

```
    if (p->p_flag&STRC) {
        stop();
        if ((n = p->p_sig) == 0)
            return(0);
```

代码清单 **6-14**　**stop()** 中的相关代码（**ken/sig.c)**

```
 1 stop()
 2 {
 3     register struct proc *pp, *cp;
 4
 5 loop:
 6     cp = u.u_procp;
 7     if(cp->p_ppid != 1)
 8     for (pp = &proc[0]; pp < &proc[NPROC]; pp++)
 9         if (pp->p_pid == cp->p_ppid) {
10             wakeup(pp);
11             cp->p_stat = SSTOP;
12             swtch();
13             if ((cp->p_flag&STRC)==0 || procxmt())
14                 return;
15             goto loop;
16         }
17     exit();
18 }
19
```

7~11　如果父进程不是 init 进程，唤醒父进程并将执行进程设置为 SSTOP 状态。

12　执行 swtch() 切换执行进程并期待能够切换至父进程。当前进程若希望再次成为执行进程，必须由父进程将其设定为 SRUN 状态。

13~15　当前进程再次成为执行进程时的处理。如果跟踪处理已结束，或是 procxmt() 的处理结果为真时，执行 return 并返回一般处理。否则将继续循环，以促使父进程再次介入子进程的处理。

17　如果没有发现父进程，或父进程为 init 进程时则认为发生异常，调用 exit() 终止进程。

ptrace()

ptrace() 是系统调用 ptrace 的处理函数（表 6-9，代码清单 6-15 ），向作为跟踪对象的子进程发出指令进行处理。父进程和子进程通过 ipc 结构体通信。

子进程也可以利用系统调用 ptrace 将自身的 STRC 标志位设置为 1。

表 **6-9**　**ptrace()** 的参数

参数	含义
`r0`	设定 ipc.ip_data 的值
`u.u_arg[0]`	跟踪对象进程的 ID
`u.u_arg[1]`	设定 ipc.ip_addr 的值
`u.u_arg[2]`	指令的种类。0 与其他数字的含义不同，表示子进程正被父进程跟踪

代码清单 **6-15　ptrace()**（**ken/sig.c)**

```
 1 ptrace()
 2 {
 3     register struct proc *p;
 4 
 5     if (u.u_arg[2] <= 0) {
 6         u.u_procp->p_flag =| STRC;
 7         return;
 8     }
 9     for (p=proc; p < &proc[NPROC]; p++)
10         if (p->p_stat==SSTOP
11          && p->p_pid==u.u_arg[0]
12          && p->p_ppid==u.u_procp->p_pid)
13             goto found;
14     u.u_error = ESRCH;
15     return;
16 
17     found:
18     while (ipc.ip_lock)
19         sleep(&ipc, IPCPRI);
20     ipc.ip_lock = p->p_pid;
21     ipc.ip_data = u.u_ar0[R0];
22     ipc.ip_addr = u.u_arg[1] & ~01;
23     ipc.ip_req = u.u_arg[2];
24     p->p_flag =& ~SWTED;
25     setrun(p);
26     while (ipc.ip_req > 0)
27         sleep(&ipc, IPCPRI);
28     u.u_ar0[R0] = ipc.ip_data;
29     if (ipc.ip_req < 0)
30         u.u_error = EIO;
31     ipc.ip_lock = 0;
32     wakeup(&ipc);
33 }
```

5~8　如果表示跟踪指令种类的参数小于或等于 0，则将执行进程的跟踪标志位 STRC 置 1 并返回。

9~15　寻找满足下述条件的进程，找到后跳转到 `found` 。

- 处于 SSTOP 状态
- `proc.p_pid` 与系统调用 `ptrace` 的参数值相同
- 执行进程的子进程

18~19　找到跟踪对象进程时的处理。首先尝试取得 `ipc` 结构体的锁，如果无法取得则进入睡眠状态。

20~24　如果成功取得锁，设置 `ipc` 结构体的参数，并解除跟踪对象进程的 `SWTED` 标志位。

25　执行 `setrun()`，设定子进程的状态为 `SRUN`。此时子进程的 `SSTOP` 状态被解除，成为可执行状态。

26~27　进入睡眠状态直至 `ipc.ip_req` 小于或等于 0。由子进程执行的 `procxmt()` 会把 `ipc.ip_req` 设为 0。

28　当 `procxmt()` 由子进程执行完毕，且当前进程通过 `swtch()` 被选为执行进程后，将 `r0` 设为 `ipc.ip_data`，并将其作为系统调用 `ptrace` 的返回值。

29~30　`ipc.ip_req` 的值小于 0 时按出错处理。如果 `procxmt()` 在处理中出错，会将 `ipc.ip_req` 设置为小于 0 的值。

31~32　最后解除 `ipc` 结构体的锁，并唤醒正在等待同一把锁的其他进程。

procxmt()

`procxmt()` 由被跟踪的子进程执行（代码清单 6-16）。根据由父进程执行的系统调用 `ptrace` 设定的 `ipc` 结构体进行读写数据等处理（表 6-10）。

表 **6-10　ipc.ip_req**

ipc.ip_req	处理
1	读取位于子进程地址空间的数据
2	读取位于子进程地址空间的数据。只有当代码段和数据段分别由不同的 APR 进行管理时才被使用，因此在 PDP-11/40 的环境下未被使用
3	读取 PPDA 的数据
4	向子进程地址空间写入数据
5	向子进程地址空间写入数据。只有当代码段和数据段分别由不同的 APR 进行管理时才被使用，因此在 PDP-11/40 的环境下未被使用
6	向 PPDA 写入数据
7	发送信号
8	终止子进程

代码清单 **6-16　procxmt()（ken/sig.c）**

```
procxmt()
{
    register int i;
    register int *p;

    if (ipc.ip_lock != u.u_procp->p_pid)
        return(0);
    i = ipc.ip_req;
```

```
 9     ipc.ip_req = 0;
10     wakeup(&ipc);
11 
12     switch (i) {
13 
14     case 1:
15         if (fuibyte(ipc.ip_addr) == -1)
16             goto error;
17         ipc.ip_data = fuiword(ipc.ip_addr);
18         break;
19 
20     case 2:
21         if (fubyte(ipc.ip_addr) == -1)
22             goto error;
23         ipc.ip_data = fuword(ipc.ip_addr);
24         break;
25 
26     case 3:
27         i = ipc.ip_addr;
28         if (i<0 || i >= (USIZE<<6))
29             goto error;
30         ipc.ip_data = u.inta[i>>1];
31         break;
32 
33     case 4:
34         if (suiword(ipc.ip_addr, 0) < 0)
35             goto error;
36         suiword(ipc.ip_addr, ipc.ip_data);
37         break;
38 
39     case 5:
40         if (suword(ipc.ip_addr, 0) < 0)
41             goto error;
42         suword(ipc.ip_addr, ipc.ip_data);
43         break;
44 
45     case 6:
46         p = &u.inta[ipc.ip_addr>>1];
47         if (p >= u.u_fsav && p < &u.
u_f        sav[25])
48             goto ok;
49         for (i=0; i<9; i++)
50             if (p == &u.u_ar0[regloc[i]])
51                 goto ok;
52         goto error;
53     ok:
54         if (p == &u.u_ar0[RPS]) {
55             ipc.ip_data =| 0170000;
56             ipc.ip_data =& ~0340;
57         }
58         *p = ipc.ip_data;
59         break;
60 
61     case 7:
62         u.u_procp->p_sig = ipc.ip_data;
63         return(1);
64 
65     case 8:
66         exit();
67 
68     default:
69     error:
70         ipc.ip_req = -1;
71     }
72     return(0);
73 }
```

6~7 如果锁没有被正确取得则退出处理。此处期待锁由 `ptrace()` 取得。

9~10　重置 ipc.ip_req，并唤醒因执行 ptrace() 并等待 procxmt() 执行结束而处于睡眠状态的进程。

12~72　根据由父进程设定的 ipc.ip_req 进行各种处理。如果 ipc.ip_req 的值为 7，则向子进程发送信号并返回 1，然后继续子进程的一般处理。如果值为 8，则执行 exit() 强制终止进程的运行。如果为其他值则返回 0，再次促使父进程介入子进程的处理。

wait()

wait() 中与跟踪相关的处理如下所示（代码清单 6-17 ）。

代码清单 **6-17**　**wait()** 中的相关代码（**ken/sys1.c)**

```
32        if(p->p_stat == SSTOP) {
33            if((p->p_flag&SWTED) == 0) {
34                p->p_flag =| SWTED;
35                u.u_ar0[R0] = p->p_pid;
36                u.u_ar0[R1] = (p->p_sig<<8) | 0177;
37                return;
38            }
39            p->p_flag =& ~(STRC|SWTED);
40            setrun(p);
41        }
```

33~38　如果未设置 SWTED 标志位，则将其设置为 1。将用户进程的 r0 设置为子进程的 proc.p_pid，将 r1 的高位 8 比特设置为 proc.p_psig，低位 8 比特设置为 0177，随后返回。同时可以期待一下此后的执行进程将执行系统调用 ptrace 。

39~40　如果在 34 行设定的 SWTED 标志位此时仍为 1（即父进程没有通过执行系统调用 ptrace 介入子进程的处理，再次执行了系统调用 wait ），表明父进程无意跟踪子进程的处理，因此将停止跟踪。解除 STRC 和 SWTED 标志位，使子进程进入可执行状态。

6.3　小结

- 信号是一种实现进程间通信的机制。
- 收到信号的进程将暂停一般处理，根据信号的种类进行相应的处理。因此可以认为信号属于中断的一种。
- 与周边设备发生的中断不同，信号完全是由软件实现的。
- 可以忽略信号，也可以执行独自定义的信号处理函数。
- 如果收到多个信号，旧的信号将被覆盖（SIGKIL 除外）。
- 由执行进程检查自己是否收到了信号。因此，向进程发送的信号未必会立即得到处理。
- 跟踪是父进程介入子进程处理的机制，主要供调试器等使用。

第 IV 部分　块 I/O 系统

块设备指的是磁盘等可以处理大量数据的设备。包括程序在内的数据被保存在块设备中,并于执行时读取至内存。第 IV 部分主要介绍以下内容。

- 内核如何提高对块设备的访问性能
- 块设备驱动如何操作块设备

通过阅读本部分的内容,可以对如何访问保存于外部设备中的数据有比较清晰的理解。

第 7 章　块设备子系统

7.1　设备的基础

设备的种类

UNIX V6 使用的周边设备可以分为块设备 和字符设备 两类（表 7-1 ）。

表 **7-1**　块设备和字符设备的比较

	块设备	字符设备
处理单位	块（512字节）	文字（1 字节）
地址	从起始块开始依次分配 0、1、2 等地址	不分配地址，数据用过后即丢弃
访问方式	可以根据需要访问任意块（随机访问）	只能从队列的起始位置开始访问数据（顺序访问）
处理速度	（一般而言）高速	（一般而言）低速
特征	适用于传输大量数据。内核通过使用缓冲区（buffer）可以实现缓存（cache）、异步处理、延迟写入等功能。文件系统也是构筑在块设备之上的	适用于传输少量数据
例如	磁盘设备、磁带设备	行打印机、控制终端

设备驱动

设备驱动是操作设备的程序 。对 1 个设备或相同种类的设备，通常存在与其对应的 1 个设备驱动。设备驱动由设备驱动表管理，需要操作某设备时，设备驱动表中相应的驱动将被调用。管理块设备驱动的设备驱动表为 `bdevsw[]` ，而管理字符设备驱动的设备驱动表为 `cdevsw[]` （图 7-1 ）。

关于块设备驱动和块设备驱动表的详细内容将在第 8 章中说明。

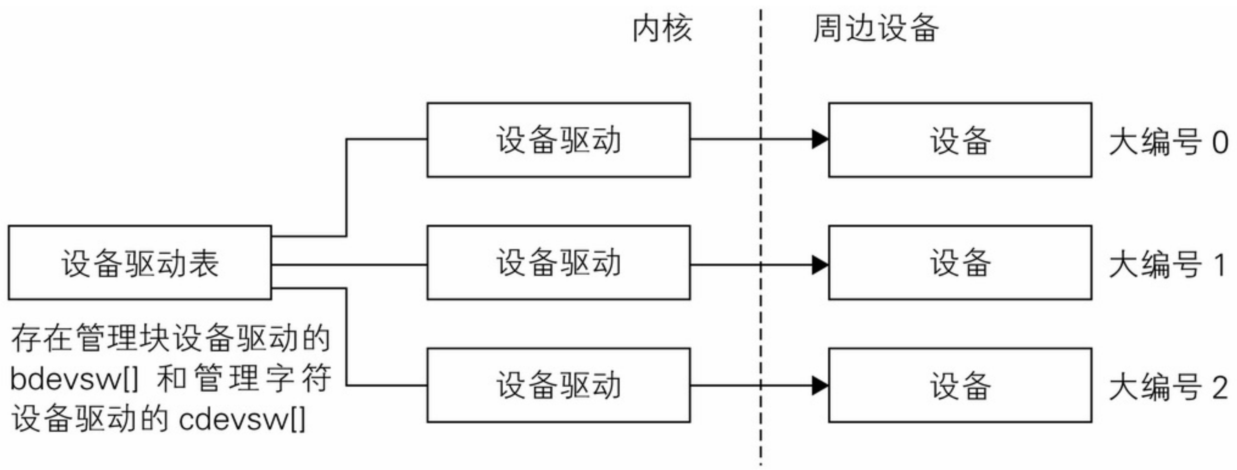


图 **7-1**　设备驱动表

类别和设备编号

设备通过类别（**class**） 和长度为 16 比特的设备编号 管理。类别用来区分块设备和字符设备。设备编号的高位 8 比特为大（**major**）编号 ，低位 8 比特为小（**minor**）编号 。大编号表示设备种类，小编号则分配给各个设备。

类别决定使用的设别驱动表，大编号决定设备驱动表中应使用的设备驱动，小编号的用途依设备而异。

特殊文件

为了使用某个设备，必须生成对应的特殊文件 。系统管理者通过执行用户程序 `/etc/mknod` 生成特殊文件。特殊文件包括设备类别和设备编号等内容。

特殊文件在 `/dev` 下生成，文件名为代表该设备的字符串和小编号的组合。例如，`/dev/rk0` 。

`/etc/mknod` 执行成功后，特殊文件被添加至指定目录，也将自动生成对应的 `inode` （参见第 9 章）。在该 `inode` 内会设置表示设备类别的标志变量，同时也会设置 `inode.i_addr[0]` 为设备编号（图 7-2 ）。

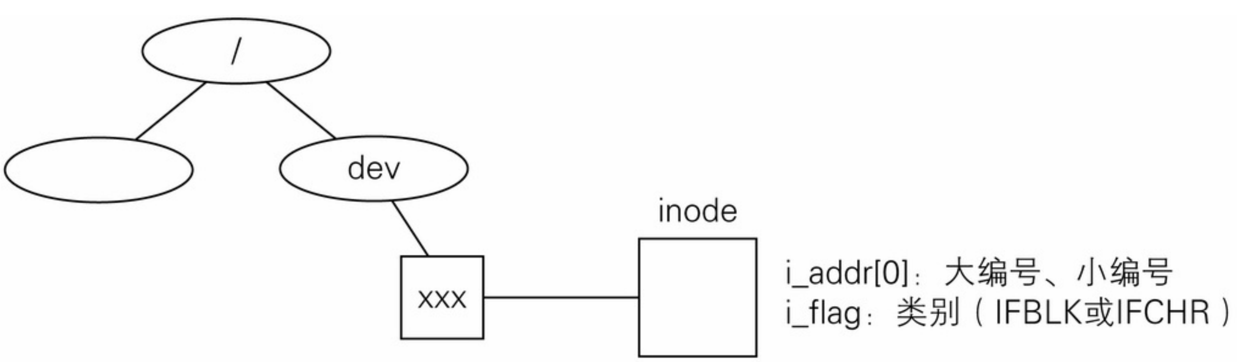


图 **7-2**　特殊文件

如果对此特殊文件执行 `open` 、`read` 、`write` 、`close` 系统调用，则在系统调用的处理函数内部将调用与大编号对应的设备驱动，来操作设备。

从用户的角度来看，操作一般文件与操作设备具有相同的处理界面 ，因此在进行将用户程序的输出对象从文件变为行打印机等操作时，只需切换 `open` 处理的对象即可（代码清单 7-1 ）。这可以视为 UNIX V6 的

特征之一。

代码清单 **7-1**　将输出对象从文件变更为设备

```
fp = open( "/work/file" ) ;
↓
fp = open( "/dev/xxx" ) ;
```

7.2　块设备子系统

访问块设备的处理由块设备子系统统一进行。块设备子系统由若干函数构成。此外，对块设备进行数据读写时使用的缓冲区也由块设备子系统进行管理（图 7-3 ）。

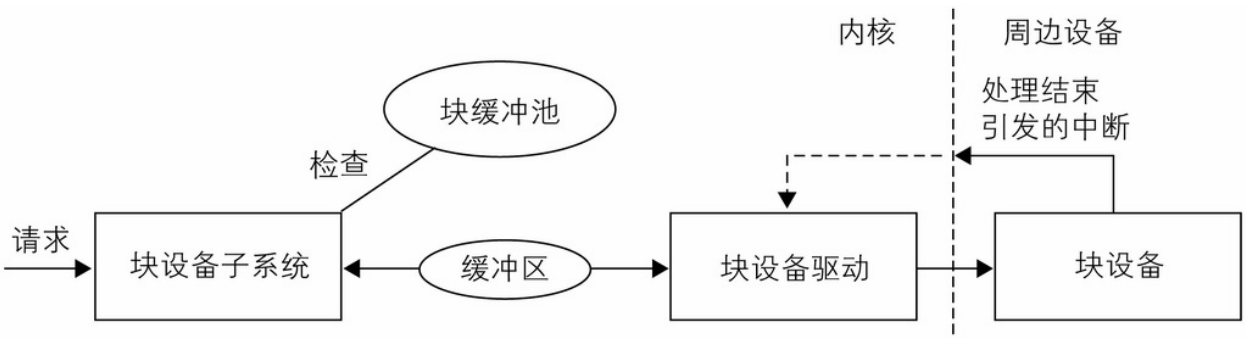


图 **7-3**　块设备子系统

缓冲区

块设备子系统通过缓冲区与块设备进行数据交换。缓冲区由设备编号和块编号命名 。对某个块进行处理时，首先检查是否存在所需的缓冲区，如果已经存在则使用该缓冲区，如果尚未存在则从未分配的缓冲区中获取新的缓冲区，对其命名后使用（图 7-4 ）。

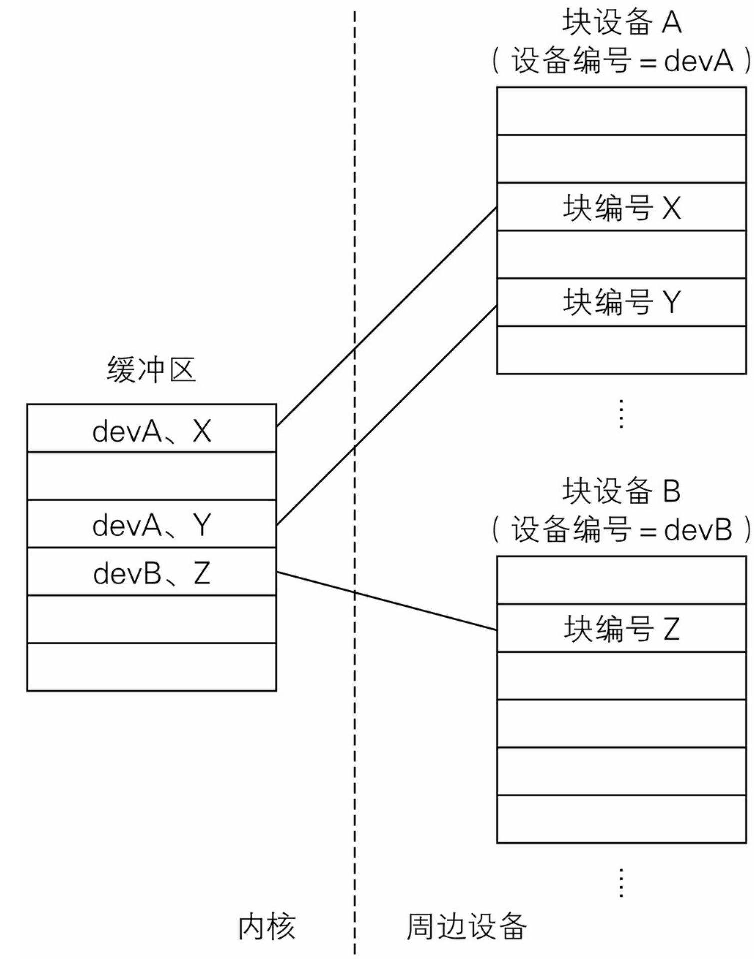


图 **7-4**　块设备缓冲区

使用缓冲区的目的主要有两个。首先是为了在多个进程同时访问同一个块时保持数据的一致性 。通过使用缓冲区和排他处理可以实现这一要求。其次，将需要经常进行读写操作的块的拷贝（缓存）保存在内存中以改善性能 。缓存能够减少对块设备的访问次数，由于块设备的访问速度与 CPU 的执行速度相比十分缓慢，因此减少访问次数将使系统性能得到相应提升。此外，还采取了通过将需要访问的数据预先读取至缓冲区，或是在写满缓冲区后再向块设备写出数据等方法，提高系统的性能。

缓冲区由 `buf` 结构体的数组 `buf[]` 管理（代码清单 7-2，代码清单 7-3，表 7-2 ）。通过将 `b_dev` 设定为设备编号、`b_blkno` 设定为块编号为缓冲区命名。

代码清单 **7-2**　**buf[]**（**buf.h)**

```
struct buf
{
    int       b_flags;
    struct    buf *b_forw;
    struct    buf *b_back;
    struct    buf *av_forw;
    struct    buf *av_back;
    int       b_dev;
    int       b_wcount;
    char      *b_addr;
    char      *b_xmem;
    char      *b_blkno;
    char      b_error;
```

```
    char      *b_resid;
} buf[NBUF];

#define    B_WRITE   0
#define    B_READ    01
#define    B_DONE    02
#define    B_ERROR   04
#define    B_BUSY    010
#define    B_PHYS    020
#define    B_MAP     040
#define    B_WANTED  0100
#define    B_RELOC   0200
#define    B_ASYNC   0400
#define    B_DELWRI  01000
```

代码清单 **7-3**　**NBUF**（**param.h**）

```
#define    NBUF    15
```

表 **7-2**　**buf** 结构体

成员	含义
b_flags	标志变量，参照表7-3
*b_forw	指向b-list前方的指针
*b_back	指向b-list后方的指针
*av_forw	指向av-list前方的指针
*av_back	指向av-list后方的指针
b_dev	设备编号
b_wcount	读写长度。在访问设备时以 2 的补数形式指定
*b_addr	用于放置设备数据拷贝的内存区域
*b_xmem	地址的扩张比特。用于保存位于物理地址第16比特之后的数据
*b_blkno	块编号
b_error	表示在访问设备时发生错误
*b_resid	用于 RAW 输入输出。保存因错误而无法传送的数据的长度（以字节为单位）

表 **7-3**　缓冲区的标志

标志	含义
B_WRITE	进行写入处理。与未设置B_READ时的含义相同
B_READ	进行读取处理。未被设置时表示进行写入处理
B_DONE	此缓冲区拥有最新的数据。表示已完成与设备的同步（读取或写入），或者拥有比设备更新的数据（写入）。设置此标志时，不会对设备进行读取数据的处理
B_ERROR	发生错误
B_BUSY	此缓冲区处于使用中的状态。在av-list中不存在
B_PHYS	表示处于 RAW 输入输出的状态
B_MAP	未在PDP-11/40环境中使用
B_WANTED	此缓冲区处于使用中的状态。第三者正等待该缓冲区被释放
B_RELOC	此标志未被使用（未定义）
B_ASYNC	进行预先读取，异步写入。不等待设备处理结束。处理结束后，在被调用的iodone()中对B_BUSY进行重置
B_DELWRI	进行延迟写入。并非立刻将数据写入设备，而是在通过getblk()对缓冲区进行再分配时，或是在bflush()被调用时将数据写入设备

`buf` 结构体用于保存缓冲区的状态等管理数据。块设备中数据的拷贝保存在 `buffers[]` 的数组元素中，`buf.b_addr` 则指向该元素（代码清单 7-4 ）。`buf.b_addr` 和 `buffers[]` 的关联通过 `binit()` 设定（图 7-5 ）。

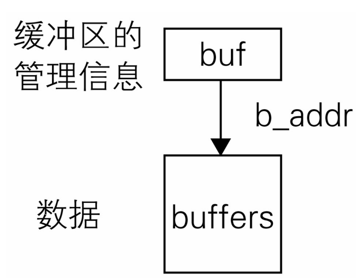


图 **7-5**　**buf** 和 **buffers**

代码清单 **7-4**　**buffers[]**（**dmr/bio.h**）

```
char    buffers[NBUF][514];
```

b-list 和 av-list

`buf[]` 通过 **b-list** 和 **av-list** 的双重环形队列 管理（图 7-6 ）。

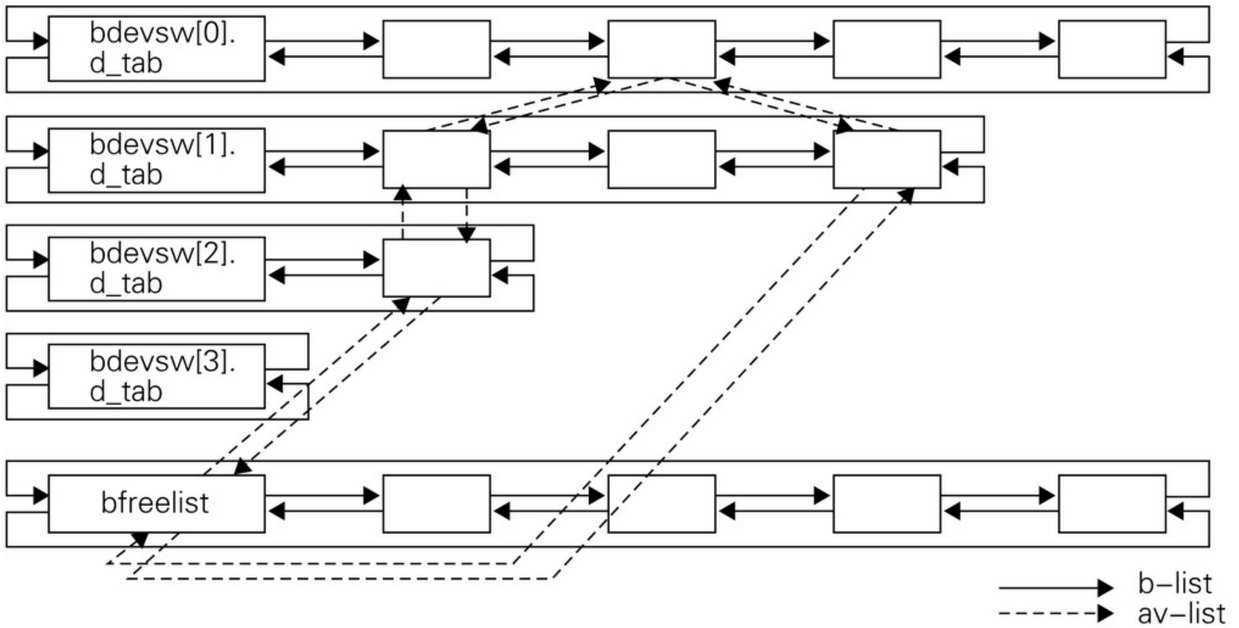


图 **7-6　b-list** 和 **av-list**

b-list 用来管理已分配给各块设备的缓冲区，通过 `buf.b_forw` 和 `buf.b_back` 构成环形队列。位于各 b-list 头部的元素为 `devtab` 结构体，由块设备驱动表 `bdevsw[]` 的 `d_tab` 指定（代码清单 7-5，表 7-4 ）。`devtab` 结构体也用于设备处理队列，详细请参照第 8 章。

但是，对尚未分配给设备（`NODEV` ）的缓冲区而言，位于 b-list 头部的元素为 `bfreelist` （代码清单 7-6 ）。

av-list 用来管理处于非使用状态（`B_BUSY` 以外）的缓冲区，通过 `buf.av_forw` 和 `buf.av_back` 构成环形队列。由 av-list 管理的缓冲区将被重新分配（给其他设备）。与尚未分配给设备的 b-list 相同，位于 av-list 头部的元素为 `bfreelist` 。无论是 b-list 还是 av-list，当队列为空时，队列头部的元素指向自身。

代码清单 **7-5　devtab** 结构体（**buf.h)**

```
struct devtab
{
    char      d_active;
    char      d_errcnt;
    struct    buf *b_forw;
    struct    buf *b_back;
    struct    buf *d_actf;
    struct    buf *d_actl;
};
```

表 **7-4　devtab** 结构体

成员	含义

d_active	设备处理中
d_errcnt	错误计数
*b_forw	指向b-list的头部
*b_back	指向b-list的末尾
*b_actf	指向设备处理队列的头部
*b_actl	指向设备处理队列的末尾

代码清单 **7-6　bfreelist（buf.h)**

```
struct    buf bfreelist;
```

RAW 输入输出

向块设备传送数据时可以不通过缓冲区，并且不受块长度（512 字节）的限制，这称为 RAW（无处理）输入输出，用于对块设备的数据进行整体备份等处理（图 7-7 ）。

一般情况下，输入输出功能需要通过缓冲区与虚拟地址空间交换数据。但是 RAW 输入输出可以直接传送数据，无需通过缓冲区。

为了使用 RAW 输入输出功能，系统管理者需要生成专用的特殊文件。首先生成供字符设备使用的特殊文件，然后将其注册到字符设备驱动表。

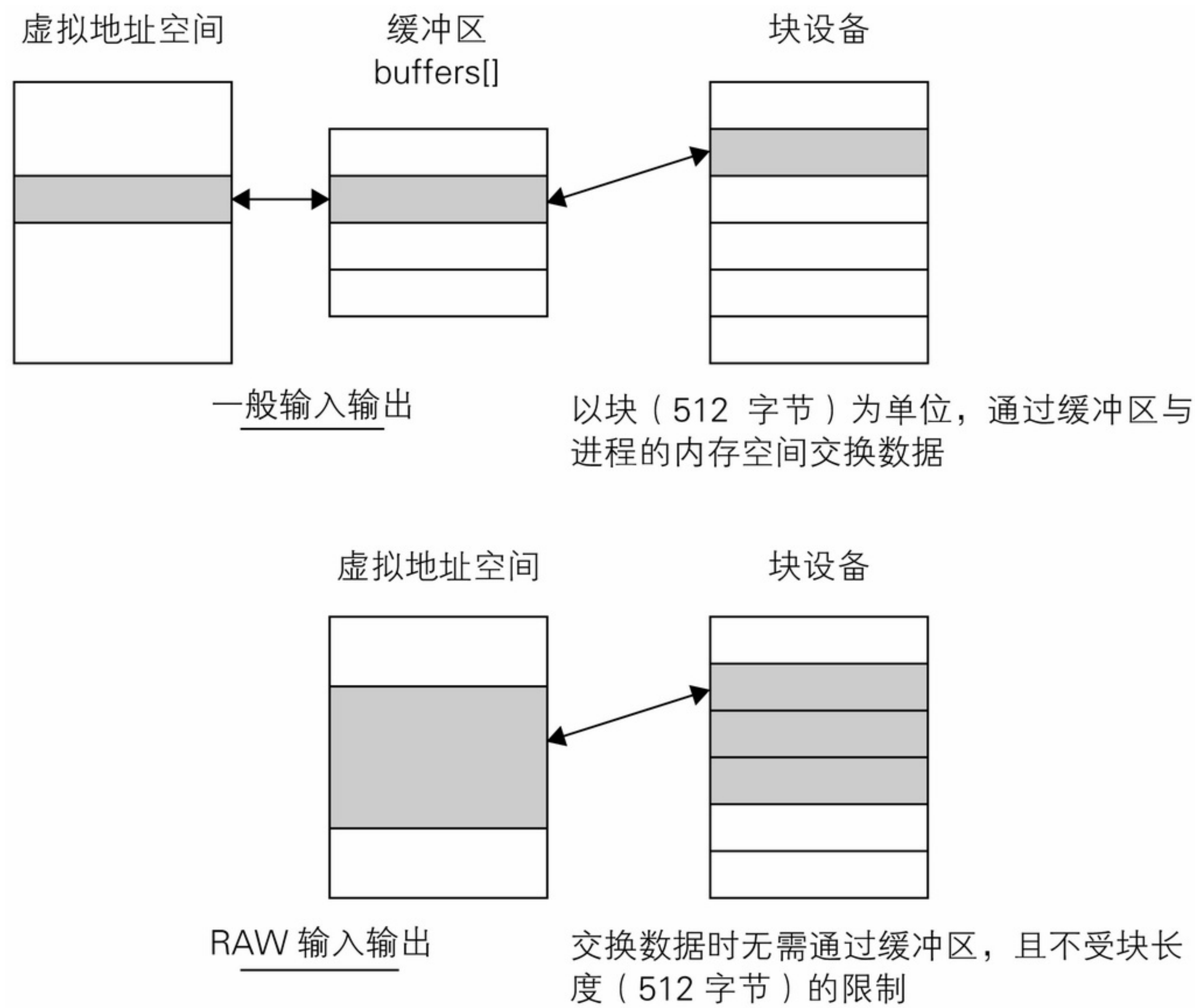


图 **7-7**　一般输入输出和 **RAW** 输入输出

7.3　缓冲区的初始化

binit()

`binit()` 是对缓冲区进行初始化的函数（代码清单 7-7 ）。在系统启动时由 `main()` 调用，且仅调用一次。

代码清单 **7-7**　**binit()**（**dmr/bio.c)**

```
binit()
{
    register struct buf *bp;
    register struct devtab *dp;
    register int i;
    struct bdevsw *bdp;

    bfreelist.b_forw = bfreelist.b_back =
        bfreelist.av_forw = bfreelist.av_back = &bfreelist;
    for (i=0; i<NBUF; i++) {
        bp = &buf[i];
        bp->b_dev = -1;
        bp->b_addr = buffers[i];
```

```
14          bp->b_back = &bfreelist;
15          bp->b_forw = bfreelist.b_forw;
16          bfreelist.b_forw->b_back = bp;
17          bfreelist.b_forw = bp;
18          bp->b_flags = B_BUSY;
19          brelse(bp);
20      }
21      i = 0;
22      for (bdp = bdevsw; bdp->d_open; bdp++) {
23          dp = bdp->d_tab;
24          if(dp) {
25              dp->b_forw = dp;
26              dp->b_back = dp;
27          }
28          i++;
29      }
30      nblkdev = i;
31  }
```

12　`b_dev = -1` 表示此缓冲区处于 `NODEV` 状态。

21~29　对赋予 `bdevsw[]` 的 `devtab.d_tab` 进行初始化处理。将位于各设备 b-list 头部的 `devtab` 结构体的成员变量 `b_forw` 和 `b_back` 指向自身。

30　将 `nblkdev` 设置为块设备（驱动）的数量。

`binit()` 对 `buf[]` 和 `buffers[]` 进行关联，将所有的缓冲区追加至 av-list 和处于 `NODEV` 状态的 b-list。此外，统计位于 `bdevsw[]` 中的设备驱动的数量，将其设置到 `nblkdev` 中（代码清单 7-8 ）。

代码清单 **7-8**　**nblkdev**（**conf.h)**

```
int    nblkdev;
```

clrbuf()

`clrbuf()` 将缓冲区的数据清 0（表 7-5，代码清单 7-9 ）。

表 **7-5**　**clrbuf()** 的参数

参数	含义
bp	缓冲区

代码清单 **7-9**　**clrbuf()**（**dmr/bio.c)**

```
clrbuf(bp)
int *bp;
{
    register *p;
    register c;

    p = bp->b_addr;
    c = 256;
    do
        *p++ = 0;
```

```
11     while (--c);
12 }
```

7.4　缓冲区的获取和释放

getblk()

`getblk()` 是取得根据设备编号和块编号命名的缓冲区的函数（表 7-6，代码清单 7-10 ）。按顺序遍历相应设备的 b-list，寻找是否存在所需的缓冲区。找到后将其从 av-list 中删除（设置标志位 `B_BUSY` ），并返回该缓冲区。如果已设置了该缓冲区的标志位 `B_BUSY` ，则设置标志位 `B_WANTED` 并进入睡眠状态。当正在使用该缓冲区的其他进程将其释放后，进程将被唤醒。

如果 b-list 中不存在所需的缓冲区，则取得位于 av-list 头部的缓冲区（设置其标志位为 `B_BUSY` ），对其重新命名，并追加到 b-list 的头部，然后返回该缓冲区。设置标志位 `B_BUSY` 意在表示该缓冲区正处于使用中的状态（图 7-8 ）。

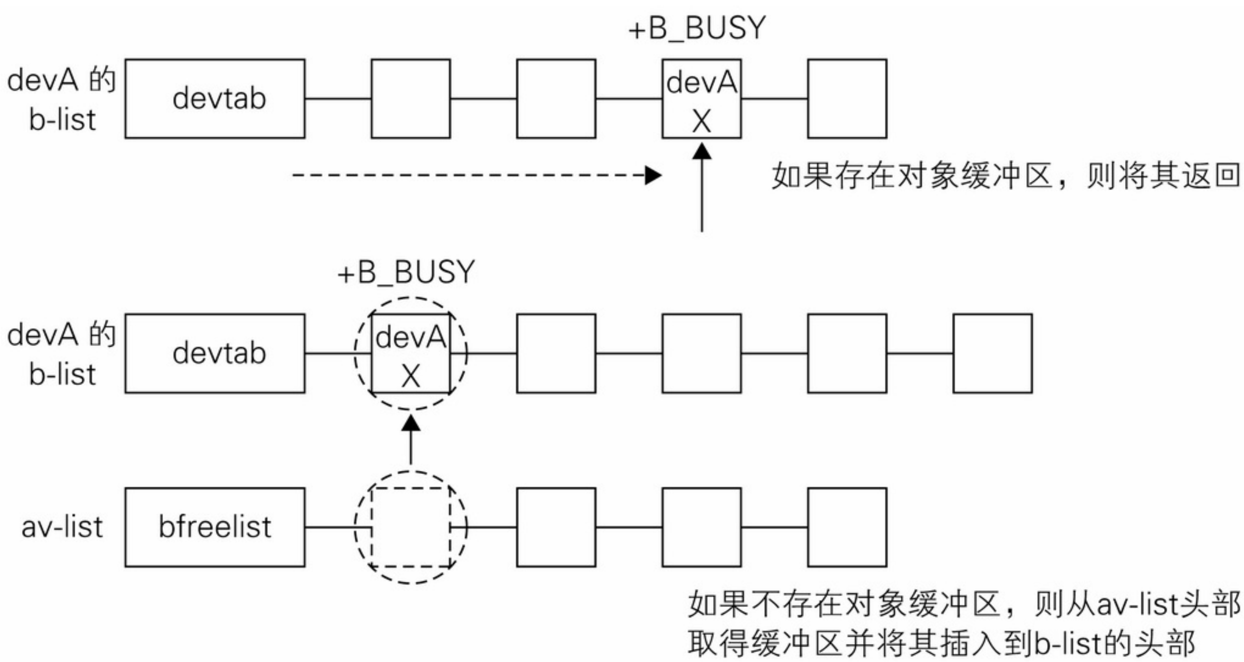


图 **7-8　getblk()**

在尝试从 av-list 取得缓冲区时，如果该缓冲区的类型为 `B_DELWRI` （延迟写入），则需要对设备进行异步读写。

表 **7-6　getblk()** 的参数

参数	含义
`dev`	设备编号
`blkno`	块编号

代码清单 **7-10　getblk()（dmr/bio.c)**

```
 1 getblk(dev, blkno)
 2 {
 3     register struct buf *bp;
 4     register struct devtab *dp;
 5     extern lbolt;
 6
 7     if(dev.d_major >= nblkdev)
 8         panic("blkdev");
 9
10     loop:
11     if (dev < 0)
12         dp = &bfreelist;
13     else {
14         dp = bdevsw[dev.d_major].d_tab;
15         if(dp == NULL)
16             panic("devtab");
17         for (bp=dp->b_forw; bp != dp; bp = bp->b_forw) {
18             if (bp->b_blkno!=blkno || bp->b_dev!=dev)
19                 continue;
20             spl6();
21             if (bp->b_flags&B_BUSY) {
22                 bp->b_flags =| B_WANTED;
23                 sleep(bp, PRIBIO);
24                 spl0();
25                 goto loop;
26             }
27             spl0();
28             notavail(bp);
29             return(bp);
30         }
31     }
32     spl6();
33     if (bfreelist.av_forw == &bfreelist) {
34         bfreelist.b_flags =| B_WANTED;
35         sleep(&bfreelist, PRIBIO);
36         spl0();
37         goto loop;
38     }
39     spl0();
40     notavail(bp = bfreelist.av_forw);
41     if (bp->b_flags & B_DELWRI) {
42         bp->b_flags =| B_ASYNC;
43         bwrite(bp);
44         goto loop;
45     }
46     bp->b_flags = B_BUSY | B_RELOC;
47     bp->b_back->b_forw = bp->b_forw;
48     bp->b_forw->b_back = bp->b_back;
49     bp->b_forw = dp->b_forw;
50     bp->b_back = dp;
51     dp->b_forw->b_back = bp;
52     dp->b_forw = bp;
53     bp->b_dev = dev;
54     bp->b_blkno = blkno;
55     return(bp);
56 }
```

7~8　如果大编号的值过大则调用 `panic()`。

11~12　NODEV（-1）时的处理。将 dp 设置为 NODEV 的 b-list 的起始元素。

13　非 NODEV 时的处理。

14~16　从 `bdevsw[]` 中取得相应设备的 `devtab` 结构体（b-list 的起始元素）。

17~19　遍历 b-list，检查是否存在所需的缓冲区。

20　如果成功找到，则将处理器优先级提升为 6，防止发生中断。由于块设备处理结束时引发的中断处理等会操作缓冲区，因此抑制中断可以避免在操作缓冲区时发生冲突。

21~26　如果此缓冲区正在使用，则设置 `B_WANTED` 标志位并进入睡眠状态。

27~29　将处理器优先级重置为 0，设置处于使用中的标志位，然后将缓冲区从 av-list 中删除，返回该缓冲区。

32　如果在 b-list 中没有找到所需的缓冲区，或是希望取得 `NODEV` 的缓冲区时，从 av-list 中取得缓冲区，并将处理器优先级提升至 6 防止发生中断。

33~38　av-list 为空时的处理。设置 av-list 的起始元素 `bfreelist` 的 `B_WANTED` 标志位并进入睡眠状态。

39　将处理器优先级重置为 0。

40　取得 av-list 的起始元素的缓冲区，将其从 av-list 中删除，并对其设置 `B_BUSY` 标志位。

41~45　如果取得的缓冲区设置了 `B_DELWRI`（延迟写入）标志位，则立刻对设备进行异步写入，且无需等待设备的处理结束。

46　`B_RELOC` 标志位在 UNIX V6 中未被使用。此时只设置了 `B_BUSY`（和 `B_RELOC`）标志位。请注意，不是 `bp->b_flags =|` 而是 `bp->b_flags =`。

47~48　将缓冲区从当前分配的 b-list 中删除。

49~52　将缓冲区追加至新的 b-list 的起始位置。

53~54　为缓冲区命名。

55　返回缓冲区。

notavail()

`notavail()` 是将缓冲区从 av-list 中删除，同时设置 `B_BUSY`（使用中）标志位的函数（表 7-7，代码清单 7-11）。

表 **7-7**　**notavail()** 的参数

参数	含义
bp	缓冲区

代码清单 **7-11**　**notavail()**（**dmr/bio.c**）

```
notavail(bp)
struct buf *bp;
{
    register struct buf *rbp;
    register int sps;

    rbp = bp;
```

```
 8     sps = PS->integ;
 9     spl6();
10     rbp->av_back->av_forw = rbp->av_forw;
11     rbp->av_forw->av_back = rbp->av_back;
12     rbp->b_flags =| B_BUSY;
13     PS->integ = sps;
14 }
```

9　将处理器优先级提升至 6，防止发生中断。由于块设备引发的中断处理等会操作缓冲区，因此抑制中断可以避免在操作缓冲区时发生冲突。

10~11　从 av-list 中删除对象缓冲区。

12　设置 B_BUSY 标志位，表明此缓冲区正在使用。

13　将处理器优先级恢复原值。

brelse()

`brelse()` 用来释放缓冲区（清除 B_BUSY 标志位），并将被释放的缓冲区追加至 av-list（表 7-8，代码清单 7-12 ）。已分配给 b-list 的缓冲区不会从 b-list 中删除。

表 **7-8　brelse()** 的参数

参数	含义
bp	缓冲区

代码清单 **7-12　brelse()**（**dmr/bio.c)**

```
brelse(bp)
struct buf *bp;
{
    register struct buf *rbp, **backp;
    register int sps;

    rbp = bp;
    if (rbp->b_flags&B_WANTED)
        wakeup(rbp);
    if (bfreelist.b_flags&B_WANTED) {
        bfreelist.b_flags =& ~B_WANTED;
        wakeup(&bfreelist);
    }
    if (rbp->b_flags&B_ERROR)
        rbp->b_dev.d_minor = -1;
    backp = &bfreelist.av_back;
    sps = PS->integ;
    spl6();
    rbp->b_flags =& ~(B_WANTED|B_BUSY|B_ASYNC);
    (*backp)->av_forw = rbp;
    rbp->av_back = *backp;
    *backp = rbp;
    rbp->av_forw = &bfreelist;
    PS->integ = sps;
}
```

8~9 唤醒正在等待当前缓冲区的进程。

10~13 唤醒正在等待缓冲区要补充到 av-list 的进程。清除 `bfreelist` 的 `B_WANTED` 标志位。

14~15 如果由于设备操作错误导致设置了缓冲区的错误标志，则将小编号设置为 -1。缓冲区的设备编号发生了变化，如果不通过 av-list 重新分配，此缓冲区将无法再次取得，因为其中容纳的数据有可能是错误的，需要防止错误数据被使用。

17~18 保存 PSW，将处理器优先级提升至 6 防止发生中断。

19 清除 `B_WANTED` 、`B_BUSY` 和 `B_ASYNC` 标志位。

20~23 将缓冲区返回至 av-list 的末尾。

24 将处理器优先级返回原值。

7.5 读取

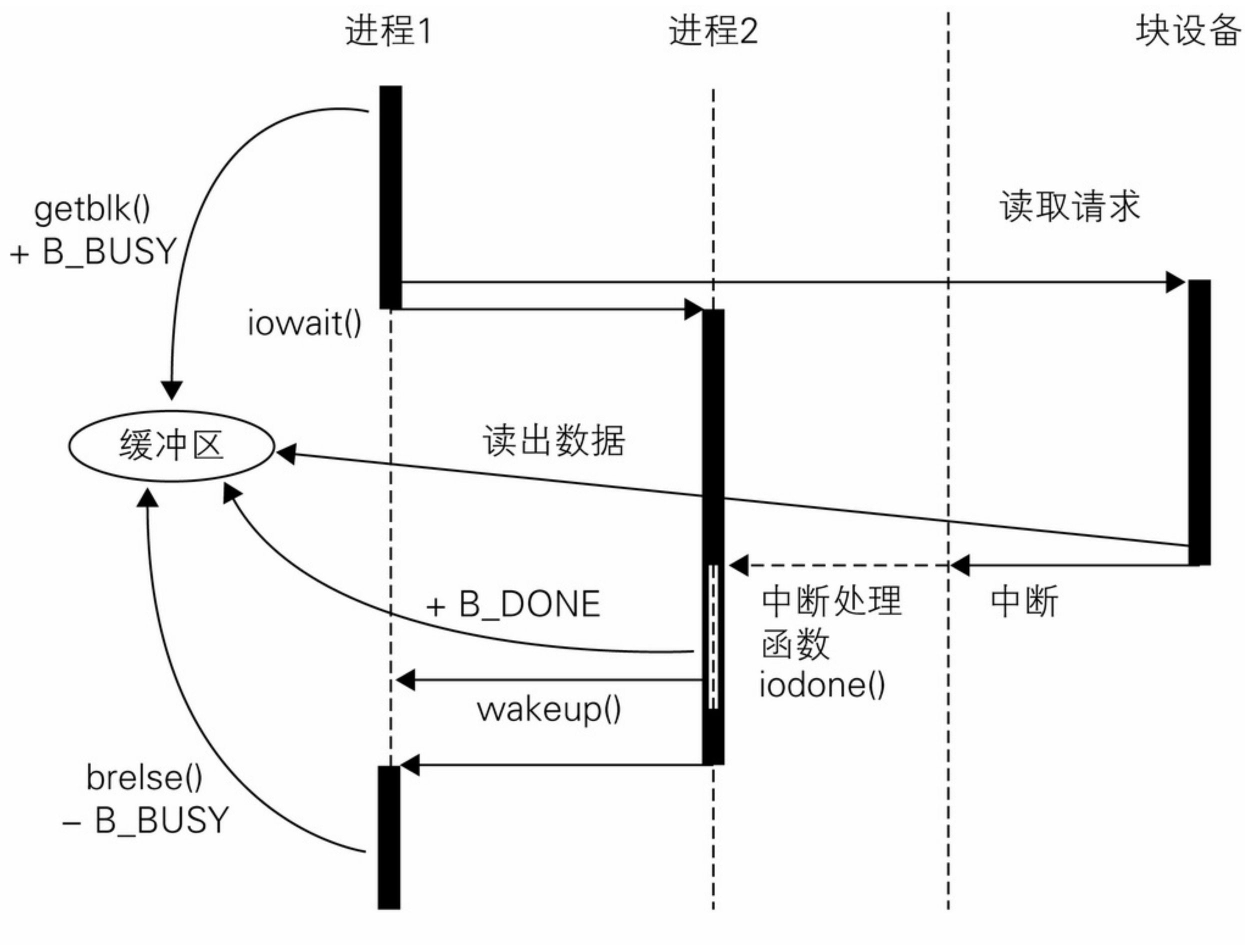
读取的种类

读取分为同步读取 和异步读取 两种类型。

同步读取时，进程首先使用 `getblk()` 取得缓冲区，然后对块设备提出读取请求，并执行 `iowait()` 进入睡眠状态。当块设备的处理完成后，由中断处理函数执行的 `iodone()` 唤醒入睡的进程。随后，该进程执行 `brelse()`，释放获取的缓冲区（图 7-9 ）。

异步读取时，进程继续原有处理，无需等待块设备的读取处理结束。缓冲区由中断处理函数执行的 `iodone()` 自动释放。当已设置了缓冲区的 `B_ASYNC` 标志位时，表示当前读取为异步读取（图 7-10 ）。

异步读取采用预读取 的方式。该方式指按顺序读取属于某个文件的块时，预先将下 一个块的数据读取至缓冲区。



进程1
进程2
块设备
getblk()
+ B_BUSY
读取请求
iowait()
缓冲区
读出数据
+ B_DONE
中断处理
函数
iodone()
中断
wakeup()
brelse()
– B_BUSY

图 **7-9**　同步读取

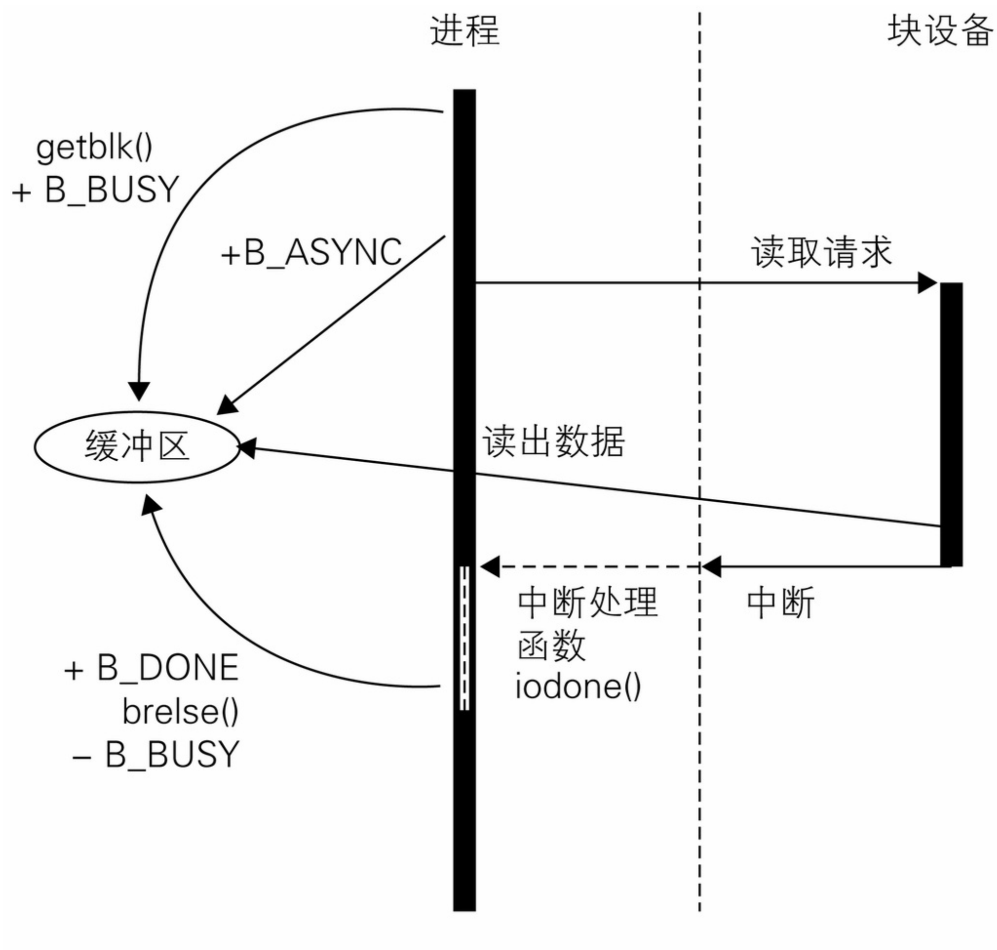


图 **7-10**　异步读取

bread()

`bread()` 是进行同步读取的函数（表 7-9，代码清单 7-13）。它通过检查由 `getblk()` 获取的缓冲区的 `B_DONE` 标志位来判断缓冲区内的数据是否为最新。如果为最新则不会访问设备，否则会对设备进行访问。

表 **7-9**　**bread()** 的参数

参数	含义
dev	设备编号
blkno	块编号

代码清单 **7-13**　**bread()**（**dmr/bio.c)**

```
bread(dev, blkno)
{
    register struct buf *rbp;

    rbp = getblk(dev, blkno);
```

```
 6     if (rbp->b_flags&B_DONE)
 7         return(rbp);
 8     rbp->b_flags =| B_READ;
 9     rbp->b_wcount = -256;
10     (*bdevsw[dev.d_major].d_strategy)(rbp);
11     iowait(rbp);
12     return(rbp);
13 }
```

8~10　设置 B_READ 标志位，将读取长度设定为 -256（表示 512 字节），然后执行设定于 bdevsw[] 中的设备访问函数。

11　执行 iowait()，进入等待状态直到设备的读取处理结束。

12　返回缓冲区，该缓冲区用来容纳从设备读取的数据。

iowait()

iowait() 是等待设备处理结束的函数（表 7-10，代码清单 7-14 ）。它使进程进入睡眠状态直至设置缓冲区的 B_DONE 标志位。

表 **7-10**　**iowait()** 的参数

参数	含义
bp	缓冲区

代码清单 **7-14**　**iowait()**（**dmr/bio.c)**

```
iowait(bp)
struct buf *bp;
{
    register struct buf *rbp;

    rbp = bp;
    spl6();
    while ((rbp->b_flags&B_DONE)==0)
        sleep(rbp, PRIBIO);
    spl0();
    geterror(rbp);
}
```

iodone()

iodone() 是在块设备的处理结束后被调用的函数，用来设置缓冲区的 B_DONE 标志位（表 7-11，代码清单 7-15 ）。它在块设备处理结束时执行的中断处理函数中被调用。

同步读写时，唤醒正在等待缓冲区的进程。异步读写时执行 brelse() 释放缓冲区。

表 **7-11**　**iodone()** 的参数

参数	含义

bp	缓冲区

代码清单 **7-15　iodone()**（**dmr/bio.c)**

```
1 iodone(bp)
2 struct buf *bp;
3 {
4     register struct buf *rbp;
5 
6     rbp = bp;
7     if(rbp->b_flags&B_MAP)
8         mapfree(rbp);
9     rbp->b_flags =| B_DONE;
10    if (rbp->b_flags&B_ASYNC)
11        brelse(rbp);
12    else {
13        rbp->b_flags =& ~B_WANTED;
14        wakeup(rbp);
15    }
16 }
```

7~8　在 PDP-11/40 的环境下不做任何处理。

geterror()

`geterror()` 是处理错误的函数（表7-12，代码清单7-16）。如果未设置 `buf.b_flags` 的错误标志位，则不做任何处理。

表 **7-12　geterror()** 的参数

参数	含义
abp	缓冲区

代码清单 **7-16　geterror()**（**dmr/bio.c)**

```
geterror(abp)
struct buf *abp;
{
    register struct buf *bp;

    bp = abp;
    if (bp->b_flags&B_ERROR)
        if ((u.u_error = bp->b_error)==0)
            u.u_error = EIO;
}
```

breada()

`breada()` 是从块设备读取数据，同时也负责预读取的函数（表 7-13，代码清单 7-17 ）。预读取是异步进行的，当设备的读取处理结束时，由块设备驱动执行的 `iodone()` 释放缓冲区。

表 **7-13　breada()** 的参数

参数	含义
adev	设备编号
blkno	块编号
rablkno	预读取块编号

代码清单 **7-17　breada()**（**dmr/bio.c)**

```
 1 breada(adev, blkno, rablkno)
 2 {
 3     register struct buf *rbp, *rabp;
 4     register int dev;
 5
 6     dev = adev;
 7     rbp = 0;
 8     if (!incore(dev, blkno)) {
 9         rbp = getblk(dev, blkno);
10         if ((rbp->b_flags&B_DONE) == 0) {
11             rbp->b_flags =| B_READ;
12             rbp->b_wcount = -256;
13             (*bdevsw[adev.d_major].d_strategy)(rbp);
14         }
15     }
16     if (rablkno && !incore(dev, rablkno)) {
17         rabp = getblk(dev, rablkno);
18         if (rabp->b_flags & B_DONE)
19             brelse(rabp);
20         else {
21             rabp->b_flags =| B_READ|B_ASYNC;
22             rabp->b_wcount = -256;
23             (*bdevsw[adev.d_major].d_strategy)(rabp);
24         }
25     }
26     if (rbp==0)
27         return(bread(dev, blkno));
28     iowait(rbp);
29     return(rbp);
30 }
```

8　执行 incore()，检查准备读取的设备的块的缓冲区是否存在。

9~14　如果缓冲区不存在，则执行 getblk() 获取缓冲区。如果尚未设置获取的缓冲区的 B_DONE 标志位，则启动从设备读取数据的处理。

16　准备进行预处理，且缓冲区不存在时的处理。

17~24　执行 getblk() 获取缓冲区。如果已设置了获取的缓冲区的 B_DONE 标志位，则表示已有数据被读取，因此要执行 brelse() 释放缓冲区。如果未设置 B_DONE 标志位，则开始从设备读取数据的处理。请注意，此处并没有等待读取处理执行结束。

26~27　如果准备读取（非预读取）的块的缓冲区已经存在，则执行 bread() 读取数据，并返回该缓冲区。

28 执行 `iowait()` 等待设备的同步读取处理结束。

29 返回同步读取的缓冲区。

incore()

`incore()` 检查分配给某个设备的某个块的缓冲区是否存在（表 7-14，代码清单 7-18 ）。如果存在则返回该缓冲区，如果不存在则返回 0。`incore()` 遍历该设备的 b-list，对 `buf` 结构体的 `b_blkno` 和 `b_dev` 分别进行检查。

表 **7-14　incore()** 的参数

参数	含义
adev	设备编号
blkno	块编号

代码清单 **7-18　incore()**（**dmr/bio.c)**

```
incore(adev, blkno)
{
    register int dev;
    register struct buf *bp;
    register struct devtab *dp;

    dev = adev;
    dp = bdevsw[adev.d_major].d_tab;
    for (bp=dp->b_forw; bp != dp; bp = bp->b_forw)
        if (bp->b_blkno==blkno && bp->b_dev==dev)
            return(bp);
    return(0);
}
```

7.6　写入

写入的种类

写入可以分为同步写入、异步写入、延迟写入 3 种类型。同步写入和异步写入的区别与读取处理的情况相同，前者等待设备处理结束，后者并不等待。

延迟写入时，首先执行 `getblk()` 取得缓冲区，随后将准备写入设备的数据写入该缓冲区。但是此时并不对设备提出写入请求，而是在设置缓冲区的 `B_ASYNC` 和 `B_ DELWRI` 标志位后，执行 `brelse()` 释放缓冲区。

经过一段时间后，当再次执行 `getblk()`（出于与上述延迟写入不同的原因），并试图再次分配已设置了 `B_DELWRI` 标志位的缓冲区时，将以异步执行的方式对设备提出写入数据的请求（图 7-11 ）。

此外，当执行刷新缓冲区的函数 `bflush()` 时，也会使用已设置了 `B_DELWRI` 标志位的缓冲区，对设备进行写入处理。

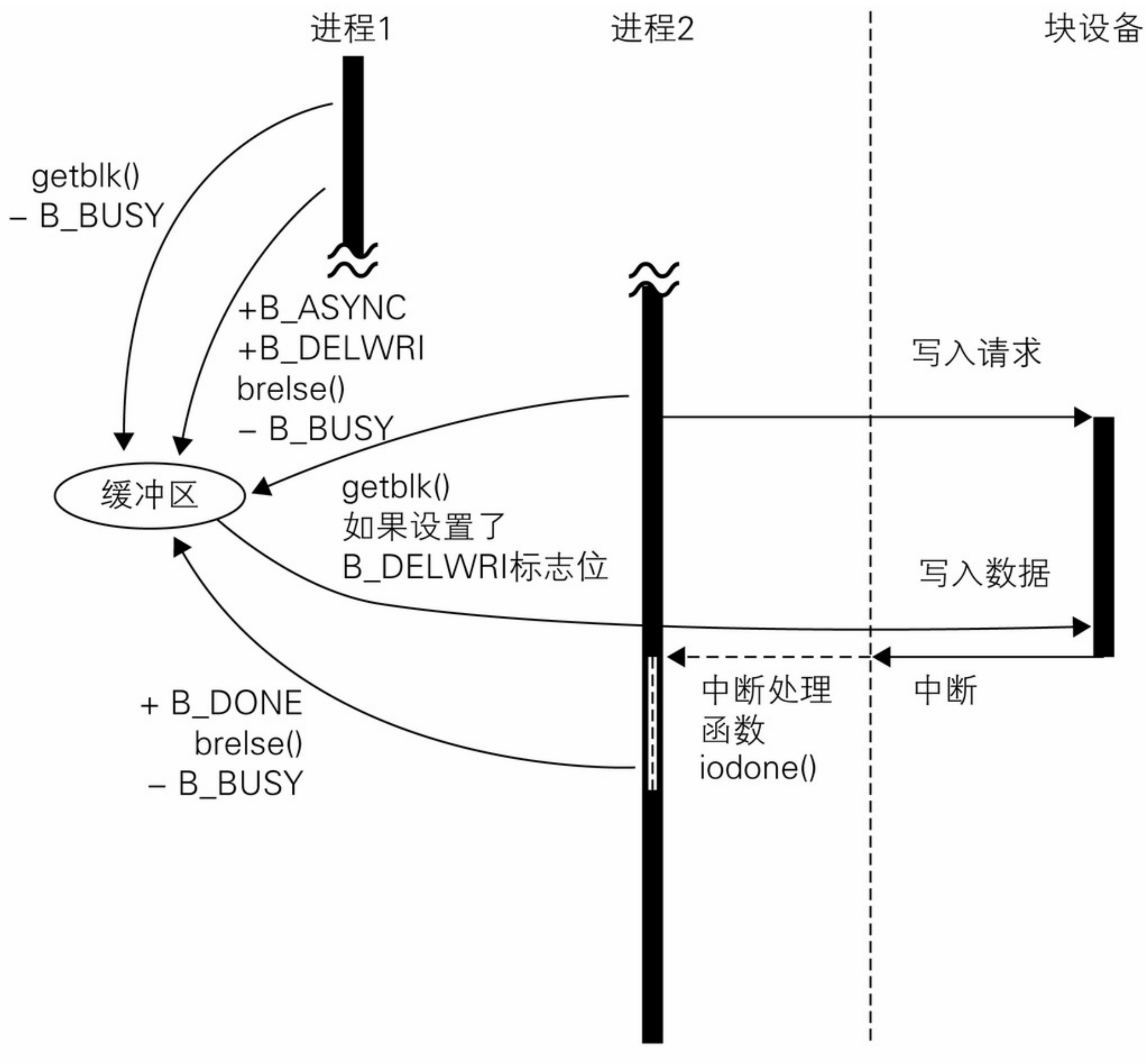


图 **7-11**　延迟写入

bwrite()

`bwrite()` 将缓冲区内的数据写入设备（表 7-15，代码清单 7-19 ）。如果未设置 `B_ASYNC` 标志位则进行同步写入，否则进行异步写入。

表 **7-15**　**bwrite()** 的参数

参数	含义
bp	缓冲区

代码清单 **7-19**　**bwrite()**（**dmr/bio.c)**

```
 1 bwrite(bp)
 2 struct buf *bp;
 3 {
 4     register struct buf *rbp;
 5     register flag;
 6
 7     rbp = bp;
 8     flag = rbp->b_flags;
 9     rbp->b_flags =& ~(B_READ | B_DONE | B_ERROR | B_DELWRI);
10     rbp->b_wcount = -256;
11     (*bdevsw[rbp->b_dev.d_major].d_strategy)(rbp);
12     if ((flag&B_ASYNC) == 0) {
13         iowait(rbp);
14         brelse(rbp);
15     } else if ((flag&B_DELWRI)==0)
16         geterror(rbp);
17 }
```

9　清除以下标志位：B_READ 、B_DONE 、B_ERROR 、B_DELWRI 。

11　执行在 bdevsw[] 中注册的设备访问函数。

12~14　如果未设置 B_ASYNC 标志位，则等待设备处理结束后，调用 brelse() 释放缓冲区。

15~16　如果已设置 B_ASYNC 标志位，则不等待设备处理结束。此外，如果未设置 B_DELWRI 标志位，则调用 geterror() 对错误进行检查。

bawrite()

bawrite() 是进行异步写入的函数（表 7-16，代码清单 7-20 ）。该函数首先设置 B_ASYNC 标志位，然后调用 bwrite() 。

表 **7-16**　**bawrite()** 的参数

参数	含义
bp	缓冲区

代码清单 **7-20**　**bawrite()**（**dmr/bio.c)**

```
bawrite(bp)
struct buf *bp;
{
    register struct buf *rbp;

    rbp = bp;
    rbp->b_flags =| B_ASYNC;
    bwrite(rbp);
}
```

bdwrite()

bdwrite() 进行延迟写入（表 7-17，代码清单 7-21 ）。该函数首先设置表示延迟写入的 B_DELWRI 标志位，然后释放缓冲区。

表 **7-17　bdwrite()** 的参数

参数	含义
bp	缓冲区

代码清单 **7-21　bdwrite()**（**dmr/bio.c)**

```
1 bdwrite(bp)
2 struct buf *bp;
3 {
4     register struct buf *rbp;
5     register struct devtab *dp;
6
7     rbp = bp;
8     dp = bdevsw[rbp->b_dev.d_major].d_tab;
9     if (dp == &tmtab || dp == &httab)
10        bawrite(rbp);
11    else {
12        rbp->b_flags =| B_DELWRI | B_DONE;
13        brelse(rbp);
14    }
15 }
```

8~10　如果写入的对象为磁带设备，则调用 `bawrite()` 并立刻进行写入处理。

bflush()

`bflush()` 将延迟写入缓冲区内的数据一次性写入设备（表 7-18，代码清单 7-22 ）。`bflush()` 被 `update()` 调用，而 `update()` 被 `panic()` 、`sync()` 、`sumount()` 调用。

在 UNIX V6 的环境中，守护程序 `/etc/update` 每隔 30 秒运行一次 `sync` 指令。

表 **7-18　bflush()** 的参数

参数	含义
dev	处理对象的设备编号。值为 -1（NODEV）时刷新所有设备的延迟写入缓冲区

代码清单 **7-22　bflush()**（**dmr/bio.c)**

```
bflush(dev)
{
    register struct buf *bp;

loop:
    spl6();
    for (bp = bfreelist.av_forw; bp != &bfreelist; bp = bp->av_forw) {
        if (bp->b_flags&B_DELWRI && (dev == NODEV||dev==bp->b_dev)) {
            bp->b_flags =| B_ASYNC;
            notavail(bp);
            bwrite(bp);
            goto loop;
        }
    }
```

```
    spl0();
}
```

7.7　RAW 输入输出

physio()

`physio()` 是进行 RAW 输入输出的函数（表 7-20，代码清单 7-23 ）。传送数据的地址、长度以及在块设备中的偏移量，都通过 `user` 结构体指定（表 7-19 ）。

`physio()` 由注册于字符设备驱动表中用于 RAW 输入输出的设备驱动调用。所使用的缓冲区不是 `buf[]`，而是供 RAW 输入输出专用的缓冲区（参见第 8 章）。

处理的流程为：读入用户 APR 的值，根据虚拟地址直接计算出物理地址，然后向缓冲区传递参数，再执行访问块设备的函数。

表 **7-19**　传递给 **physio()** 的 **user** 结构体成员变量

成员	含义
`u.u_base`	传送数据的虚拟地址（以字节为单位）
`u.u_offset`	块设备中的偏移量（以字节为单位）
`u.u_count`	传送数据长度（以字节为单位）

表 **7-20**　**physio()** 的参数

参数	含义
`strat`	指向块设备访问函数的指针
`abp`	缓冲区。使用的是 RAW 输入输出专用的缓冲区
`dev`	设备编号
`rw`	指定是读取还是写入

代码清单 **7-23**　**physio()**（**dmr/bio.c**）

```
physio(strat, abp, dev, rw)
struct buf *abp;
int (*strat)();
{
    register struct buf *bp;
    register char *base;
    register int nb;
    int ts;
```

```
 9
10     bp = abp;
11     base = u.u_base;
12     if (base&01 || u.u_count&01 || base>=base+u.u_count)
13         goto bad;
14     ts = (u.u_tsize+127) & ~0177;
15     if (u.u_sep)
16         ts = 0;
17     nb = (base>>6) & 01777;
18     if (nb < ts)
19         goto bad;
20     if ((((base+u.u_count)>>6)&01777) >= ts+u.u_dsize
21         && nb < 1024-u.u_ssize)
22         goto bad;
23     spl6();
24     while (bp->b_flags&B_BUSY) {
25         bp->b_flags =| B_WANTED;
26         sleep(bp, PRIBIO);
27     }
28     bp->b_flags = B_BUSY | B_PHYS | rw;
29     bp->b_dev = dev;
30     bp->b_addr = base&077;
31     base = (u.u_sep? UDSA: UISA)->r[nb>>7] + (nb&0177);
32     bp->b_addr =+ base<<6;
33     bp->b_xmem = (base>>10) & 077;
34     bp->b_blkno = lshift(u.u_offset, -9);
35     bp->b_wcount = -((u.u_count>>1) & 077777);
36     bp->b_error = 0;
37     u.u_procp->p_flag =| SLOCK;
38     (*strat)(bp);
39     spl6();
40     while ((bp->b_flags&B_DONE) == 0)
41         sleep(bp, PRIBIO);
42     u.u_procp->p_flag =& ~SLOCK;
43     if (bp->b_flags&B_WANTED)
44         wakeup(bp);
45     spl0();
46     bp->b_flags =& ~(B_BUSY|B_WANTED);
47     u.u_count = (-bp->b_resid)<<1;
48     geterror(bp);
49     return;
50     bad:
51     u.u_error = EFAULT;
52 }
```

12~22 检查作为参数传递进来的 `user` 结构体成员变量。

- 传送的虚拟地址为偶数
- 传送的数据长度为偶数
- 虚拟地址和数据长度之和是否溢出
- 是否试图传送位于代码段领域的数据
- 传送数据是否位于数据领域（下限）和栈领域（上限）之间

此时 `ts` 的值为代码段的长度（以 64 字节为单位，并以 128×64 字节为单位向上取整），`nb` 的值为传送数据的虚拟地址（以 64 字节为单位）。

24~27 如果供 RAW 输入输出使用的缓冲区正在使用，则设置 `B_WANTED` 标志位，然后进入睡眠状态直到缓冲区被释放。

28~37　将作为参数的 `user` 结构体传递给缓冲区。根据用户 PAR 的值从虚拟地址计算得到物理地址。将 `buf.b_xmem` 设定为物理内存第 16 位之后的部分。从 `u.u_offset` 计算出块编号，同时设定执行进程的 `SLOCK` 标志位，防止进程被换出至交换空间。

38　执行设备驱动的访问函数。

40~41　进入睡眠状态，等待块设备处理结束。

42　清除 `SLOCK` 标志位。

43~44　如果有进程在等待供 RAW 输入输出使用的缓冲区，则将其唤醒。

47　`buf.b_resid` 中保存有因出错而没能传送的数据长度，将其赋予 `u.u_count` 。

swap()

`swap()` 是进行交换处理的函数（表 7-21，代码清单 7-25 ）。该函数使用 `buf` 结构体类型的变量 `swbuf` （代码清单 7-24 ）进行 RAW 输入输出实现交换处理（图 7-12 ）。因为 `swbuf` 只有一个，无法同时交换两个以上的进程，所以需要进行排他处理。

`swap()` 为 `swbuf` 设定适当的参数，调用交换磁盘的设备驱动读写数据。因为参数的长度以 64 字节为单位，所以需要将地址变为字节单位，再将长度（字长）变为 2 的补数的形式。这些值最后都会赋予交换磁盘的寄存器。请参考 `physio()` 以及第 8 章中有关 RK 的寄存器的内容。

代码清单 **7-24**　**swbuf**（**bio.c)**

```
1 struct    buf    swbuf;
```

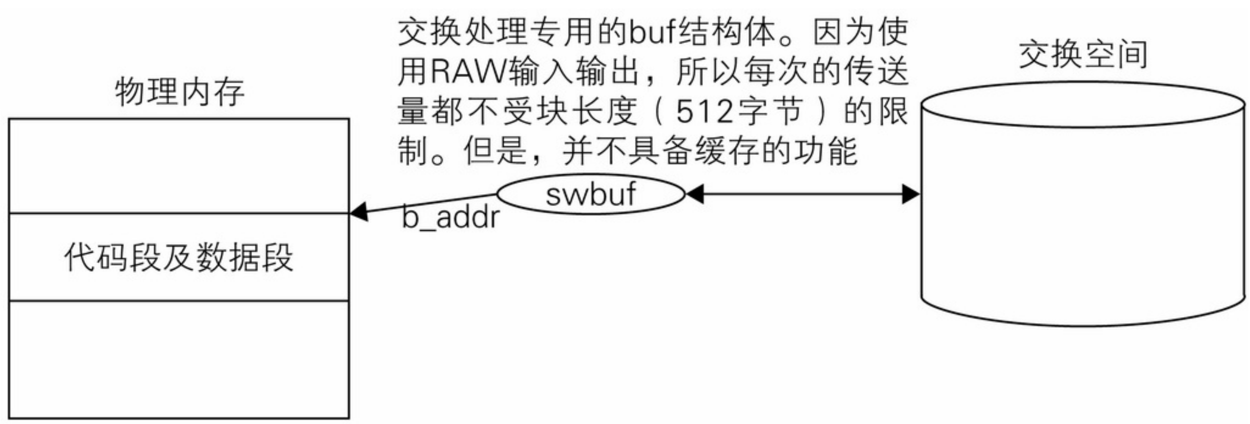


图 **7-12**　**swbuf**

表 **7-21**　**swap()** 的参数

参数	含义
`blkno` 优先级	交换磁盘中的块编号
`coreaddr`	交换对象的物理内存地址（以 64 字节为单位）

count	交换对象的长度（以 64 字节为单位）
rdflg	0：换出。1：换入

代码清单 **7-25　swap()**（**dmr/bio.c)**

```
 1 swap(blkno, coreaddr, count, rdflg)
 2 {
 3     register int *fp;
 4
 5     fp = &swbuf.b_flags;
 6     spl6();
 7     while (*fp&B_BUSY) {
 8         *fp =| B_WANTED;
 9         sleep(fp, PSWP);
10     }
11     *fp = B_BUSY | B_PHYS | rdflg;
12     swbuf.b_dev = swapdev;
13     swbuf.b_wcount = - (count<<5);
14     swbuf.b_blkno = blkno;
15     swbuf.b_addr = coreaddr<<6;
16     swbuf.b_xmem = (coreaddr>>10) & 077;
17     (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
18     spl6();
19     while((*fp&B_DONE)==0)
20         sleep(fp, PSWP);
21     if (*fp&B_WANTED)
22         wakeup(fp);
23     spl0();
24     *fp =& ~(B_BUSY|B_WANTED);
25     return(*fp&B_ERROR);
26 }
```

交换磁盘的设备编号通过 swapdev 指定（代码清单 7-26 ）。系统管理者可以根据实际环境修改这个值，但是需要对内核进行重新构筑。此例中大编号和小编号都被设为 0，第 8 章的 bdevsw[] 的例子（代码清单 8-1 ）表示的是 RK 磁盘。

代码清单 **7-26　swapdev**（**conf.c)**

```
1 int    swapdev    {(0<<8)|0};
```

7~10　如果有其他进程使用 swbuf，则设置 swbuf.b_flags 的 B_WANTED 标志位，然后进入睡眠状态。

11　如果可以使用 swbuf，则设置 swbuf.b_flags 的 B_BUSY 标志位（swbuf 使用中）、B_PHYS 标志位（RAW 输入输出）和 rdflg 标志位（读取或写入）。

12~16　设置 swbuf 的参数启动交换磁盘的输入输出。将通过参数设定的以 64 字节为单位的地址和长度分别转换为字节单位和 2 的补数的字单位。

17　调用交换磁盘的设备驱动。

19~20　进入睡眠状态等待交换磁盘的处理结束。

21~22 交换磁盘的处理结束后，唤醒正在等待 swbuf 的进程。

24 清除 swbuf 的 B_BUSY 标志位和 B_WANTED 标志位。

25 返回交换处理成功与否的标志。访问当对设备失败时，设备驱动（rkstrategy() 等）将设定 swbuf.B_ERROR 标志位。

7.8 小结

- 设备分为块设备和字符设备两种。
- 设备驱动群由设备驱动表进行管理。
- 设备由类别和设备编号进行管理。
- 为了使用设备，必须提前生成特殊文件。
- 对块设备的处理集中在块设备子系统中。
- 通过块设备缓冲区访问块设备。缓冲区保证了数据的一致性，同时也起到缓存的作用。
- 读取分为同步读取和异步读取。预读取功能采用了异步读取。
- 写入分为同步写入和异步写入。此外还有延迟写入。
- 通过 RAW 输入输出传输数据时无需缓冲区，也不受块长度（512 字节）的限制 。

第 8 章　块设备驱动

8.1　什么是块设备驱动

块设备驱动为操作块设备的程序 。每一种块设备都具有对应的驱动程序。驱动程序集中了对设备的各种操作功能，同时也包含了设备的规格参数。其他程序通过调用与设备对应的驱动程序，可以在无须了解设备的详情下进行操作。

本章主要对块设备驱动进行说明，字符设备驱动将在第 12 章中介绍。

块设备驱动表

块设备驱动通过块设备驱动表 `bdevsw[]` 进行管理（代码清单 8-1，表 8-1 ）。`bdevsw` 结构体包含打开、关闭、访问相应设备的函数地址，以及指向位于 b-list 起始位置的 `devtab` 结构体的地址。这些函数和与设备引发的中断相对应的处理函数**,**构成了设备驱动的实体 。

代码清单 **8-1　bdevsw[]**（**conf.h)**

```
struct    bdevsw
{
    int    (*d_open)();
    int    (*d_close)();
    int    (*d_strategy)();
    int    *d_tab;
} bdevsw[];
```

表 **8-1　bdevsw** 结构体

成员	含义
`(*d_open)()`	指向打开设备的函数的指针
`(*d_close)()`	指向关闭设备的函数的指针
`(*d_strategy)()`	指向访问设备的函数的指针。读取和写入共用一个函数，通过缓冲区的标志位判断进行何种操作
`*d_tab`	指向 devtab 结构体的指针

`bdevsw[]` 本身定义在 `conf.c` 中。`bdevsw[]` 中包含了与系统中所有块设备（的种类）相对应的设备驱动。系统管理者要根据系统构成编辑 `bdevsw[]`，并对内核进行重新构筑。

代码清单 **8-2　bdevsw[]**（**conf.c)**

```
int    (*bdevsw[])()
{
    &nulldev,    &nulldev,    &rkstrategy,    rktab,    /* rk */
    &nodev,      &nodev,      &nodev,         0,        /* rp */
    &nodev,      &nodev,      &nodev,         0,        /* rf */
    &nodev,      &nodev,      &nodev,         0,        /* tm */
    &nodev,      &nodev,      &nodev,         0,        /* tc */
    &nodev,      &nodev,      &nodev,         0,        /* hs */
```

```
	&nodev,		&nodev,		&nodev,		0,		/* hp */
	&nodev,		&nodev,		&nodev,		0,		/* ht */
	0
};
```

设备的大编号与 `bdevsw[]` 的数组下标相对应 。在代码清单 8-2 的示例中，大编号 0 对应 RK 磁盘设备，1 对应 RP 磁盘设备……以此类推。

设备处理队列

设备驱动拥有设备处理队列 。队列头部的元素与 b-list 相同，为 `devtab` 结构体（`bdevsw[].d_tab` ）。队列中包含缓冲区。`devtab` 结构体的 `d_actf` 和 `d_actl` 分别指向位于头部和末尾的缓冲区，而各缓冲区的 `buf.av_forw` 指向后方的缓冲区。末尾的缓冲区的 `buf.av_forw` 指向 `NULL` （0）。

设备驱动使用的缓冲区的状态为处理中（`B_BUSY` ），并从 av-list 中清除。因此，即使直接操作 `buf.av_forw` 的值，也不会和 av-list 发生冲突。

新的缓冲区被追加到队列的末尾，设备驱动从队列的头部开始处理（图 8-1 ）。

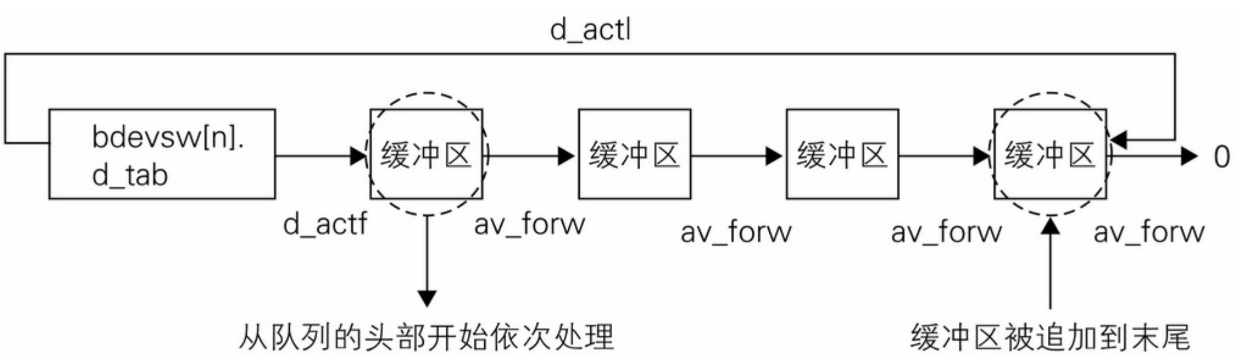


图 **8-1**　设备处理队列

处理流程

块设备的处理流程如下所示（图 8-2 ）。请注意，具体细节由设备驱动的实现方法决定，这里给出的只是比较典型的例子。

1. 内核（主要是块设备子系统）执行访问函数

2. 将传递进来的缓冲区追加至设备处理队列

3. 操作设备的寄存器，并开始对位于设备处理队列头部的缓冲区进行输入输出处理

4. 处理结束后设备将引发中断。如果设备处理队列中还留有缓冲区，中断处理函数将继续进行输入输出处理

当系统运行时，打开设备的函数会预先对设备进行初始化处理。当系统运行结束时，关闭设备的函数将终止设备的运行。根据设备的种类，这些处理也不总是必须进行的。

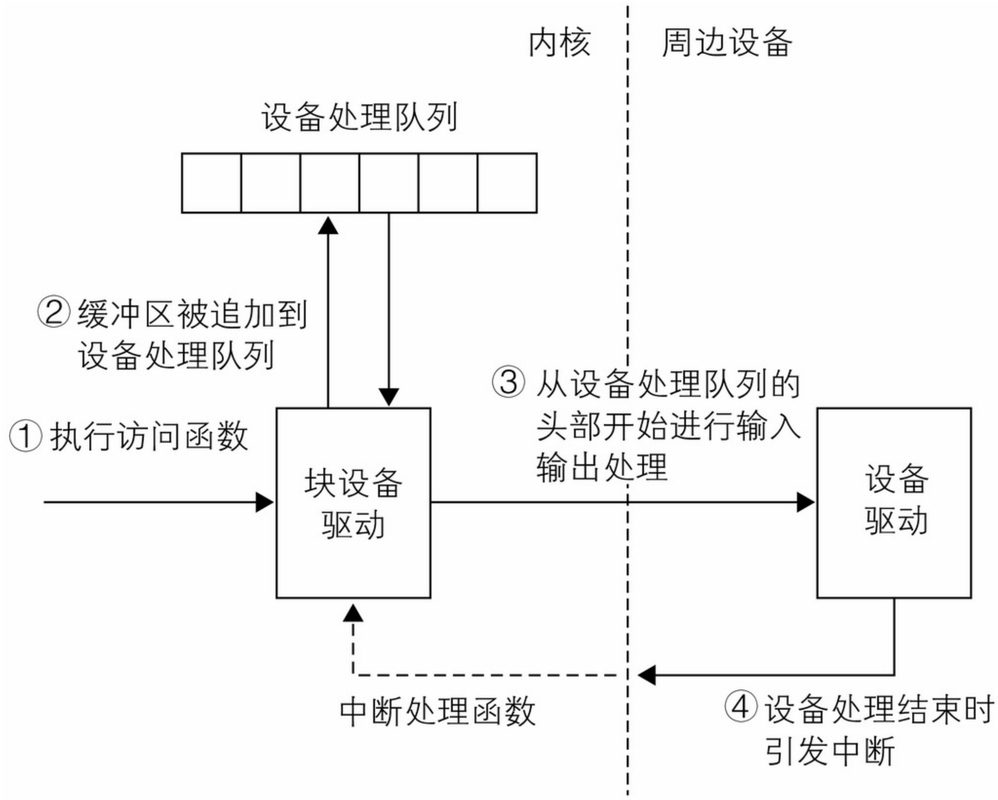


图 **8-2**　块设备驱动的处理流程

8.2　RK-11 磁盘驱动

下面以 DEC 公司开发的 RK-11 磁盘的设备驱动为例介绍块设备驱动。RK-11 磁盘具有多个型号，此处以 RK11-D 为对象。

RK11-D

RK11-D 是由 1 个磁盘控制器及 1 台磁盘构成的大容量块设备。此外还可以追加 7 台型号为 RK05 的磁盘，使 RK11-D 具备管理 8 台磁盘的能力。磁盘具有各自的小编号（图 8-3 ）。表 8-2 为磁盘的规格。

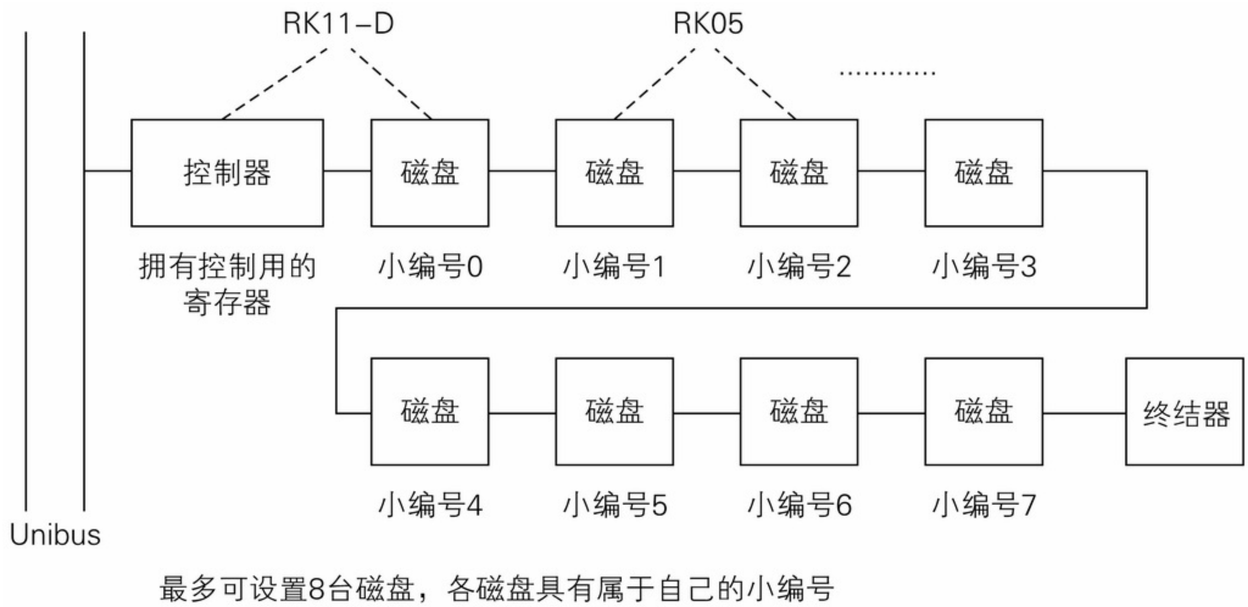


图 **8-3　RK11-D**

表 **8-2　RK11-D**

规格	值
磁道（柱面）/ 盘面	200+3（预备）
盘面 / 磁盘	2
扇区 / 磁道	12
字 / 扇区	256
块总数	4800+72（预备）
总容量（字）	1200K+18K（预备）
数据字长	16 比特

特殊文件

系统管理者在目录 /dev 下生成名为 rkn （n 为小编号）的特殊文件。

8 以上的小编号具有特殊意义。数据分散保存于 rk 0 至 rk (x-8)[1] 的各个磁盘中。小编号为 d（d 大于等于 8 ）的设备编号为 b 的块，实际对应小编号为 b%(d-7) 的设备 b/(d-7)（小数点的部分向下取整）的块。比如说小编号为 10 的设备编号为 10 的块，实际对应小编号为 1 的设备编号为 3 的块（表 8-3 ）。如果小编号不使用 0~7，只使用 8 以上的某个值，可以将所有磁盘模拟为一台大容量的磁盘。

[1] 此处的 x 表示任意大于等于 8 的小编号 。即磁盘 rkx 的数据将分散保存于 rk0 至 rk(x-8) 的各个磁盘中。参见表 8-3。——译者注

表 **8-3**　小编号为 **10** 的设备编号为 **10** 的块

设备	**rk0**	**rk1**	**rk2**
0	0	1	2
1	3	4	5
2	6	7	8
3	9	**10**	11
4	12	13	14

作为供 RAW 输入输出使用的特殊文件，在目录 `/dev` 下同时也生成名为 rrkn（n 为小编号）的特殊文件。对应的设备种类被设定为字符设备。

设定 **bdevsw[]**

以下的说明是假设 RK 磁盘的大编号为 0。将 `bdevsw[0]` 设定为 RK 的磁盘驱动（代码清单 8-3 ）。RK 磁盘因为无需做打开和关闭处理，所以相应的函数也会被设为 `nulldev()` 。访问函数为 `rkstrategy()` ，而 `bdevsw.d_tab` 被设定为 `rktab` （代码清单 8-4 ）。

代码清单 **8-3**　**bdevsw[]** 中的相关部分（**conf.c)**

```
    &nulldev,    &nulldev,    &rkstrategy,    rktab,    /* rk */
```

代码清单 **8-4**　 **rktab**（**dmr/rk.c)**

```
struct    devtab    rktab;
```

中断处理函数

RK 磁盘在处理结束后，会通过中断优先级 5、向量 0220 引发中断。将执行与向量对应的中断处理函数 `rkintr()` ，针对缓冲区进行结束输入输出的处理。当设备发生错误时将重试 **10** 次 ，重试的次数由 `devtab.d_errcnt` 管理。

RK11-D 的寄存器

RK11-D 具有 7 个控制寄存器（表 8-4 ）。RKCS 的第 0 比特位为 1 时，RKBA 和 RKCS 的 5~4 比特表示的物理内存地址和 RKDA 表示的磁盘地址之间，将传输 RKWC 定义的长度的数据。

表 **8-4**　**RK11-D** 的寄存器

寄存器	名称	含义

RKDS	Drive Status 寄存器	表示磁盘状态。只有在表示磁盘驱动中发生错误时才使用。参见表 8-5
RKER	Error 寄存器	表示错误状态。只有在表示磁盘驱动中发生错误时才使用。参见表 8-6
RKCS	Control Status 寄存器	用于控制磁盘。参见表 8-7
RKWC	Word Count 寄存器	表示待传送数据的长度。参见表 8-8
RKBA	Current Bus Address 寄存器	表示内存中待传送数据的地址。参见表 8-9
RKDA	Disk Address 寄存器	表示磁盘中待传送数据的地址。参见表 8-10
RKDB	Data Buffer 寄存器	在控制器和磁盘间传输数据时使用。设备驱动不对其进行操作。参见表 8-11

表 **8-5　RKDS**

比特位	含义
15~13	表示引发中断的磁盘
12	磁盘断电
11	连接的磁盘为 RK05
10	磁盘中发生异常
9	寻址（seek）尚未结束
8	表示 Sector Counter 是否准备完毕
7	被选择的磁盘是否处于 Ready 状态
6	表示磁盘的磁头是否处于未动作状态，即磁盘是否处于可以接受下一个请求的状态
5	被选择的磁盘是否处于只读模式
4	表示磁盘的磁头是否已经移动到扇区地址指定的扇区位置
3~0	被选择的磁盘的扇区地址

表 **8-6　RKER**

比特位	含义

15	发现磁盘已断电。或者选中的磁盘未处于 Ready 状态，或是发生了错误
14	在进行 Read、Write、Read Check、Write Check 处理时，发生磁盘地址溢出错误
13	试图向只读磁盘写入数据
12	在进行 Read、Write、Read Check、Write Check 处理时，磁盘的磁头并未移动到适当的位置
11	在进行 Read 或 Write 之外的处理时，设置 RKCS 的第 10 比特位
10	内存未应答
9	在 Write 或 Write Check 处理时，处理尚未结束但是已有多个缓冲区文件被清空。或是在 Read 处理时，处理尚未结束但是已有多个缓冲区文件被写满
8	发现无法检测出时钟脉冲
7	试图访问并不存在的磁盘
6	试图访问并不存在的柱面
5	试图访问并不存在的扇区
1	Read Check 或 Read 处理时发生校验错误
0	Write Check 时发生错误

表 **8-7　RKCS**

比特位	含义
15	当设置了 RKER 的任意一位时置 1
14	当设置了 RKER 的第 5~15 位时置 1
13	表示 Seek 或 Reset 处理结束并引发了中断
11	传送数据时防止 RKBA 的值递增
10	磁盘格式化时置 1
8	置 1 时表示当软件异常时终止对磁盘的操作
7	表示处于可接受操作的状态（Ready）

6	置 1 时表示当磁盘的处理结束或发生错误时，通过向量 0220 引发中断
5~4	扩展 RKBA 的地址。设定为内存中待传送数据的 18 比特物理地址的高位 2 比特的数据。当 RKBA 溢出时递增
3~1	000： Reset，001： Write，010： Read，011： Write Check，100： Seek，101： Read Check，110： Drive Reset，111 ： Write Lock
0	置 1 时表示启动磁盘处理

表 **8-8　RKWC**

比特位	含义
15~0	待传送数据的长度（字长）。设置为 2 的补数形式

表 **8-9　RKBA**

比特位	含义
15~0	内存中待传送数据的物理地址的低位 16 比特

表 **8-10　RKDA**

比特位	含义
15~13	磁盘编号（0~7）
12~5	柱面编号。（块编号 /12）>>1。例如，当块编号为 100 时，100/12=8(1000)，8(1000)>>1=4(100)
4	盘面编号。（块编号 /12）&1。例如，当块编号为 100 时，100/12=8(1000)，8(1000)&1=0
3~0	扇区地址（0~11）。块编号 %12。例如，当块编号为 100 时，100%12=4

表 **8-11　RKDB**

比特位	含义
15~0	在控制器和磁盘间传输数据时使用

rkstrategy()

`rkstrategy()` 是对 RK 磁盘进行读写的函数（表 8-12，代码清单 8-5 ）。首先将缓冲区追加至设备处理队列的末尾，然后执行 `rkstart()` 访问 RK 磁盘。

表 **8-12　rkstrategy()** 的参数

参数	含义
adp	缓冲区

代码清单 **8-5　rkstrategy()**（**dmr/rk.c**）

```
 1 rkstrategy(abp)
 2 struct buf *abp;
 3 {
 4     register struct buf *bp;
 5     register *qc, *ql;
 6     int d;
 7
 8     bp = abp;
 9     if(bp->b_flags&B_PHYS)
10         mapalloc(bp);
11     d = bp->b_dev.d_minor-7;
12     if(d <= 0)
13         d = 1;
14     if (bp->b_blkno >= NRKBLK*d) {
15         p->b_flags =| B_ERROR;
16         odone(bp);
17         eturn;
18     }
19     bp->av_forw = 0;
20     spl5();
21     if (rktab.d_actf==0)
22         rktab.d_actf = bp;
23     else
24         rktab.d_actl->av_forw = bp;
25     rktab.d_actl = bp;
26     if (rktab.d_active==0)
27         rkstart();
28     spl0();
29 }
```

9~10　在 PDP-11/40 的环境下不做任何处理。

11~18　如果块编号的值过大则引发错误。如果小编号小于 8，则与一个磁盘能够拥有的最大块编号 `NRKBLK`（4872）（代码清单 8-6）进行比较。大于等于 8 时与（小编号 -7）×4872 进行比较。

代码清单 **8-6　NRKBLK**（**dmr/rk.c**）

```
1 #define    NRKBLK    4872
```

19　将缓冲区的末尾设为 0。

20　将处理器优先级提升至 5，防止由 RK 引发的中断（中断优先级为 5）。

21~25　如果设备处理队列为空，则向队列的起始位置追加缓冲区。如果不为空，则将缓冲区追加至末尾位置。

26~27　如果 `rktab.d_active` 为 0（RK 未处于处理状态）则执行 `rkstart()`，开始对 RK 磁盘进行

读写操作。

28　将处理器优先级重置为 0。

rkstart()

rkstart() 是启动 RK 磁盘处理的函数（代码清单 8-7 ）。对处于设备处理队列起始位置的缓冲区进行读写操作。

代码清单 **8-7　rkstart()**（**dmr/rk.c)**

```
1 rkstart()
2 {
3     register struct buf *bp;
4
5     if ((bp = rktab.d_actf) == 0)
6         return;
7     rktab.d_active++;
8     devstart(bp, &RKADDR->rkda, rkaddr(bp), 0);
9 }
```

5~6　如果队列为空，则不做任何处理立即返回。

7　递增 rktab.d_active，表示 RK 磁盘处于处理中的状态。

8　执行 devstart() 启动磁盘处理。RKADDR 为 RK11-D 寄存器的基地址，rkda 表示基地址距 RKDA 地址的差值（代码清单 8-8，代码清单 8-9）。

代码清单 **8-8　RKADDR**（**dmr/rk.c)**

```
#define    RKADDR    0177400
```

代码清单 **8-9　RK** 寄存器参照用结构体（**dmr/rk.c)**

```
struct {
    int rkds;
    int rker;
    int rkcs;
    int rkwc;
    int rkba;
    int rkda;
};
```

rkaddr()

rkaddr() 利用小编号和块编号计算用于赋予 RKDA 的值（表 8-13，代码清单 8-10 ）。计算内容请参照表 8-10。

表 **8-13　rkaddr()** 的参数

参数	含义

bp	缓冲区

代码清单 **8-10　rkaddr()**（**dmr/rk.c)**

```
1 rkaddr(bp)
2 struct buf *bp;
3 {
4     register struct buf *p;
5     register int b;
6     int d, m;
7 
8     p = bp;
9     b = p->b_blkno;
10    m = p->b_dev.d_minor - 7;
11    if(m <= 0)
12        d = p->b_dev.d_minor;
13    else {
14        d = lrem(b, m);
15        b = ldiv(b, m);
16    }
17    return(d<<13 | (b/12)<<4 | b%12);
18 }
```

14　`lrem` 为用汇编语言编写的函数，用来计算余数。

15　`ldiv` 为用汇编语言编写的函数，用来计算商。

devstart()

`devstart()` 利用缓冲区中设置的参数，设置 RKDA、RKBA、RKWC、RKCS 并启动设备的处理（表 8-14，代码清单 8-11 ）。当 RKCS 的第 0 比特位变为 1 时，表示已启动 RK 磁盘的处理。

`devstart()` 对应多种磁盘设备，可以说它是在第 7 章介绍过的块设备子系统的一部分。为了便于说明，将其移至此处介绍。

表 **8-14　devstart()** 的参数

参数	含义
bp	缓冲区
devloc	RKDA 的地址
devblk	用于赋予 RKDA 的值
hbcom	用于赋予 RKCS 的值

代码清单 **8-11　devstart()**（**dmr/bio.c)**

```
devstart(bp, devloc, devblk, hbcom)
struct buf *bp;
int *devloc;
```

```
 4 {
 5     register int *dp;
 6     register struct buf *rbp;
 7     register int com;
 8 
 9     dp = devloc;
10     rbp = bp;
11     *dp = devblk;
12     *--dp = rbp->b_addr;
13     *--dp = rbp->b_wcount;
14     com = (hbcom<<8) | IENABLE | GO |
15         ((rbp->b_xmem & 03) << 4);
16     if (rbp->b_flags&B_READ)
17         com =| RCOM;
18     else
19         com =| WCOM;
20     *--dp = com;
21 }
```

9　将 dp 设定为 RKDA 的地址。

11　将 RKDA 设定为通过参数传递进来的值。

12　移动 dp 分别访问 RKBA、RKWC 和 RKCS。将 RKBA 设为缓冲区的地址。内核程序中的数据，其虚拟地址等于物理地址（参照第 14 章）。

13　设定 RKWC 的值。

14~20　设定 RKCS 的值。代码清单 8-12 为计算时使用的参数。图 8-4 表示计算的内容。因为内核程序使用数据的虚拟地址等于物理地址，且物理地址的前半部分被赋予内核程序，所以 buf.b_xmem 的值通常为 0。

代码清单 **8-12**　设置 **RKCS** 时使用的参数

```
#define    CTLRDY    0200
#define    IENABLE   0100
#define    WCOM      02
#define    RCOM      04
#define    GO        01
#define    RESET     0
```

```
 000000000 hbcom(0) << 8
| 001000000 IENABLE
| 000000001 GO
| 000xx0000 (rbp->b_xmem & 03) << 4
————————————
 001xx0001

读入时
 001xx0001
| 000000100 RCOM
————————————
 001xx0101

写入时
 001xx0001
| 000000010 WCOM
————————————
 001xx0011
```

图 **8-4**　赋予 **RKCS** 的值

rkintr()

`rkintr()` 为 RK 磁盘处理结束时引发中断的处理函数（代码清单 8-13 ）。它从设备处理队列删除位于头部的缓冲区，并再次执行 `rkstart()` 。当访问 RK 失败时进行 10 次重试。

对于已结束输入输出处理的缓冲区来说，`B_DONE` 标志位通过 `iodone()` 设定。异步访问时，在此处进行释放缓冲区的处理。

代码清单 **8-13**　**rkintr()**（**dmr/rk.c)**

```
rkintr()
{
    register struct buf *bp;

    if (rktab.d_active == 0)
        return;
    bp = rktab.d_actf;
    rktab.d_active = 0;
    if (RKADDR->rkcs < 0) {
        deverror(bp, RKADDR->rker, RKADDR->rkds);
        RKADDR->rkcs = RESET|GO;
        while((RKADDR->rkcs&CTLRDY) == 0) ;
        if (++rktab.d_errcnt <= 10) {
            rkstart();
            return;
        }
        bp->b_flags =| B_ERROR;
    }
    rktab.d_errcnt = 0;
    rktab.d_actf = bp->av_forw;
    iodone(bp);
```

```
22     rkstart();
23 }
```

5~6　如果 RK 磁盘未启动，则不直接返回。

7　从设备处理队列取得位于头部的缓冲区。

8　将 `rktab.d_active` 重置为 0。

9~18　如果 RKCS 的错误位（第 15 比特位）为 1，则进行下述处理。

- 调用 `deverror()` 输出错误信息
- 将 `RESET` 和 `GO` 赋予 RKCS，使磁盘进行 Reset 处理
- 等待设置 RKCS 的 Ready 位（即等待 RK 磁盘的 Reset 处理结束）
- 递增重试计数器
- 如果重试计数器的值小于等于 10 则重试，否则设置 `buf.b_flags` 的错误标志位

19　将重试计数器重置为 0。

20　从设备处理队列删除位于头部的缓冲区。

21　执行 `iodone()` 设置缓冲区的处理结束标志位（B_DONE）。

22　执行 `rkstart()`，开始处理设备处理队列的下一个缓冲区。

RAW 输入输出

`rkread()` 和 `rkwrite()` 用来进行 RAW 输入输出。这两个函数在字符设备驱动表中注册，使 RAW 输入输出使用的缓冲区 `rrkbuf`（代码清单 8-14）为参数，调用 `physio()`（代码清单 8-15）。

代码清单 **8-14　rrkbuf（dmr/rk.c)**

```
struct buf rrkbuf;
```

代码清单 **8-15　rkread()，rkwrite()（dmr/rk.c)**

```
rkread(dev)
{
    physio(rkstrategy, &rrkbuf, dev, B_READ);
}

rkwrite(dev)
{
    physio(rkstrategy, &rrkbuf, dev, B_WRITE);
}
```

8.3　小结

- 每种块设备都具有与其相对的块设备驱动
- 块设备驱动由 `bdevsw[]` 管理
- 大编号对应 `bdevsw[]` 的数组下标
- 块设备驱动拥有设备处理队列

第 V 部分　文件系统

内核使用户可以通过文件、目录等易于理解和管理的概念访问块设备上的数据。第 V 部分将介绍以下内容。

- 如何使用块设备上的区域
- 内核如何管理块设备上的空闲区域
- 文件的实体是什么
- 如何命@并管理文件
- 用户程序如何操作文件和目录

通过阅读本部分的内容,对读写文件时计算机内部发生的处理应该会有比较清晰的理解。

第 9 章　文件系统

9.1　什么是文件系统

文件系统是结构化管理块设备上的数据的机制 。它通过文件 和目录 等概念，使管理设备上的数据成为可能。

文件是为了管理块的集合而定义的概念。文件本身通过文件名管理。为了防止文件名的重复，同时给用户提供方便，UNIX V6 采用了树状结构的命名空间。目录通过文件名管理文件，它在本质上是与文件相同的概念。考虑到会有多个用户使用系统，可以设定文件和目录的访问权限。

由于存在文件系统，用户无需对块设备的种类、规格、数据的存放地址等复杂信息有任何了解。

在利用块设备的文件系统前需要对其进行挂载 （mount）。即使挂载了多个块设备的文件系统，对用户而言也是透明的，在数据操作方面并无任何不同（图 9-1 ）。

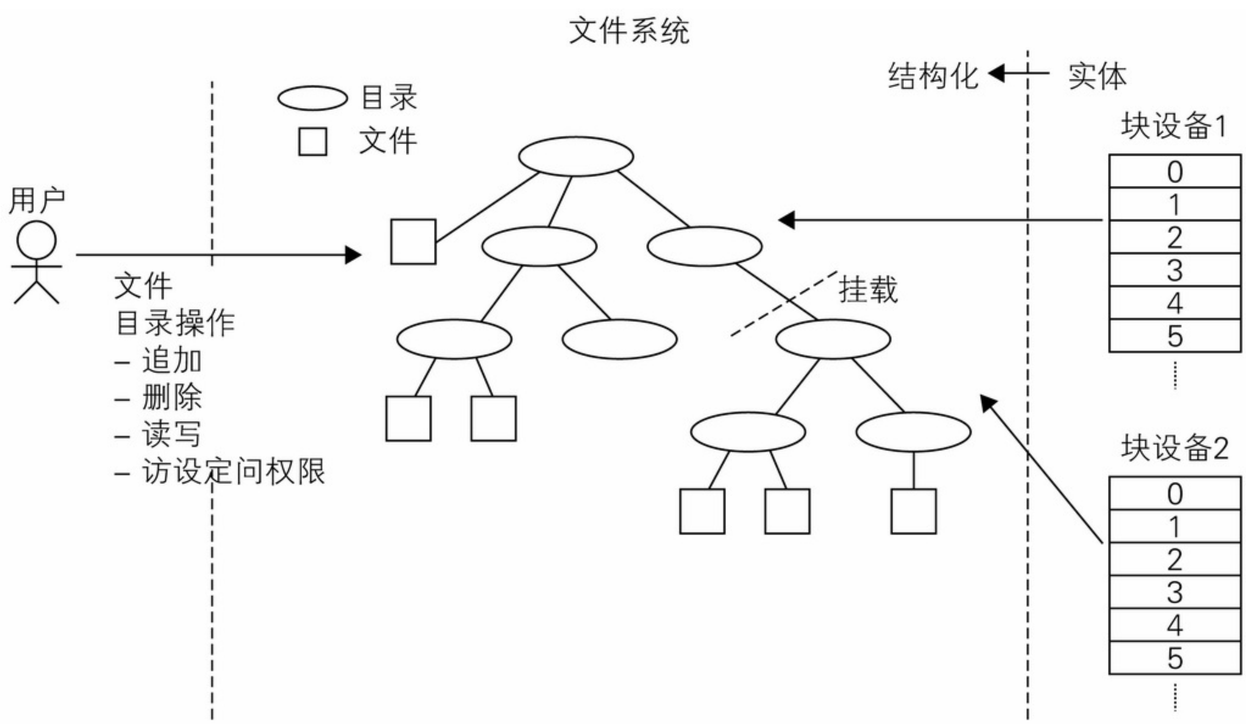


图 **9-1**　文件系统

inode

文件用来管理块设备上的块集合，它由两部分构成，定义文件的 inode 和该文件包含的数据。inode 管理文件大小、访问权限、保存数据的块设备的块编号等信息。操作文件时，系统内核需要首先取得对象文件的 inode。

inode 本身也保存在块设备中。系统内核将 inode 从块设备读取至内存时，为了便于操作，对其格式进行了适当的改变。虽然在阅读内核代码时着重考虑内存中 inode 的格式即可，但应该注意到它被写回块设备时，其数据格式是不同的，请不要忘记这一点。

树状结构的命名空间

系统利用树状结构的命名空间赋予文件唯一的文件名。文件和目录可以通过以根目录为起点的绝对路径，或以当前目录为起点的相对路径指定。

将位于树状结构顶点的目录称为根目录。由‘/’起始的路径为绝对路径。进程当前所在的目录称为当前目录，用 `user.u_cdir` 表示。如果路径的起始字符为‘/’以外的字符，则为相对路径。另外，‘.’表示对象目录本身，‘..’表示对象目录的父目录（图 9-2 ）。

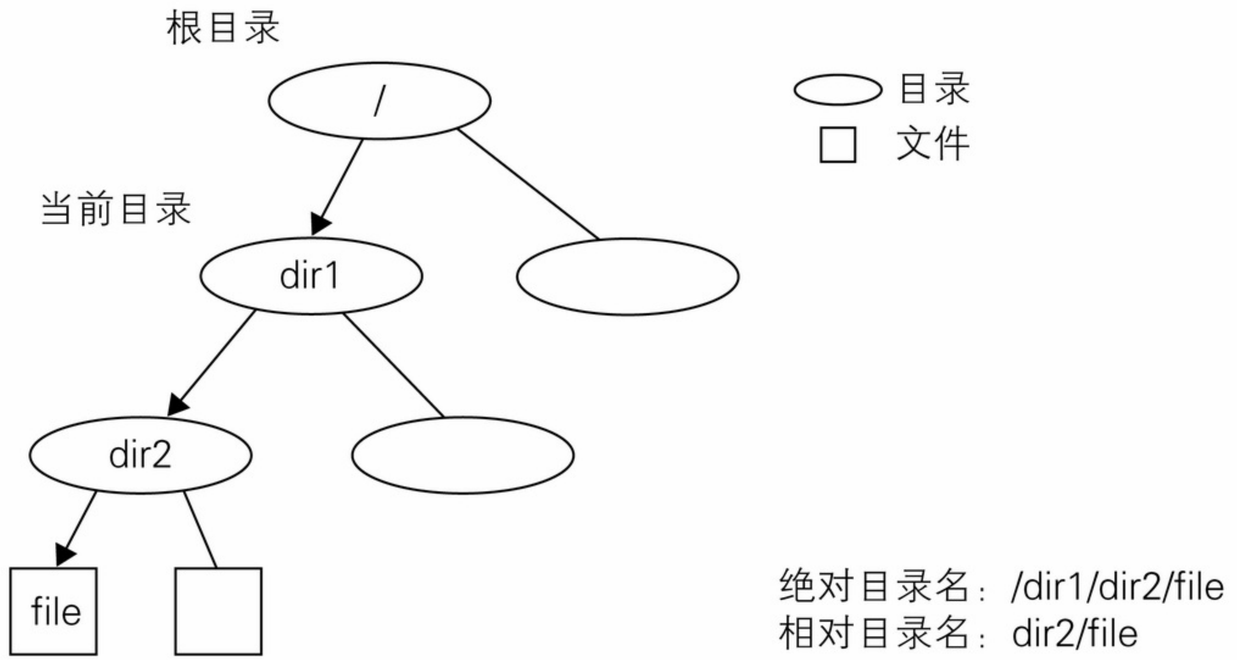


图 **9-2**　树状结构的命名空间

挂载

块设备的文件系统通过挂载操作被系统识别，反之，将其从系统中解除的操作即为卸载。利用挂载和卸载，用户可以将多个块设备的文件系统，根据需要自由地追加到系统中，或从系统中解除。

为了进行挂载，首先需要生成与块设备相对的特殊文件（special file），并利用系统调用 `mount` 将该文件与挂载点（mount point）进行关联。所谓挂载点其实是一个路径，被挂载的块设备的文件系统将以该路径为起点。已经被挂载的块设备（的文件系统）由 `mount[]` 管理。

访问权限

登录到系统的用户被赋予用户 ID。如果用户生成了文件，用户 ID 和该用户所属的组（group）的 ID 将会被记录在文件里。

文件的访问权限通过一组长度为 11 比特的数据（称为权限设定 ，permission）管理，其中的 9 比特分别对文件的所有者、所有者所属组的用户以及其他用户的读写和执行权限做了定义（表 9-1 ）。因为上述各类用户的权限均以 3 比特表示，所以文件的访问权限经常写成 0644 或 0755 这样的八进制的形式。

表 **9-1**　权限设定

比特位	含义
8	文件拥有者是否可读

7	文件拥有者是否可写
6	文件拥有者是否可以执行
5	文件拥有者所属组是否可读
4	文件拥有者所属组是否可写
3	文件拥有者所属组是否可以执行
2	其他用户是否可读
1	其他用户是否可写
0	其他用户是否可以执行

权限设定余下的 2 比特中，有 1 比特称为 SUID 。当它被置为 1 时，在文件执行时用户 ID 将暂时设置为文件拥有者的用户 ID。另有 1 比特称为 SGID ，当它被置为 1 时，在文件执行时组 ID 将暂时设置为文件拥有者所属组的 ID。

user 结构体的 u_uid 和 u_ruid 用来管理进程的用户 ID，u_gid 和 u_rgid 用来管理组 ID。SUID 和 SGID 的值为 1 时，在程序运行时 u_uid 和 u_gid 将改变，而 u_ruid 和 u_rgid 则维持原状。因此，u_uid 、u_gid 被称为实效用户 ID 和实效组 ID，u_ruid 、u_rgid 则被称为实际用户 ID 和实际组 ID。

user.u_uid 为 0 的进程称为超级用户。超级用户可以忽略权限设定操作所有文件。但是如果希望执行某个文件，必须事先对文件拥有者、所属组、其他用户的其中之一赋予可执行该文件的权限。

正因为对文件和目录采用了树状结构进行管理，权限管理也相应变得容易。比如说，可以很方便地指定除了某个用户以外的所有用户不得访问某个目录。假设没有采用树状结构进行管理的话，恐怕只能对每个文件手动设置访问权限。

根磁盘

对系统而言需要定义一个根磁盘 。根磁盘的文件系统在启动时被自动导入系统，它保存了系统内核程序和用于控制系统的用户程序（/etc/init 等），这些程序将在系统启动时执行。根磁盘以外的块设备的文件系统可在启动后挂载。

根磁盘通过 rootdev 来指定。系统管理者需要根据实际情况更改 rootdev 的设定，并重新生成系统内核。代码清单 9-1 的例子中将 rootdev 的大编号和小编号同时设定为 0。如果 bdevsw[] 按照代码清单 8-2 设置，那么小编号为 0 的 RK 磁盘（驱动器）将成为根磁盘。

代码清单 **9-1　rootdev**（**conf.c)**

```
int     rootdev     {(0<<8)|0};
```

9.2　块设备的区域

块设备被分为 4 个区域。编号为 0 的块将在系统启动时投入使用。编号为 1 的块称为超级块（superblock），它包含了该设备的信息。其后依次为 **inode** 区域 和存储区域 。inode 区域和存储区域的大小由超级块定义（图 9-3 ）。

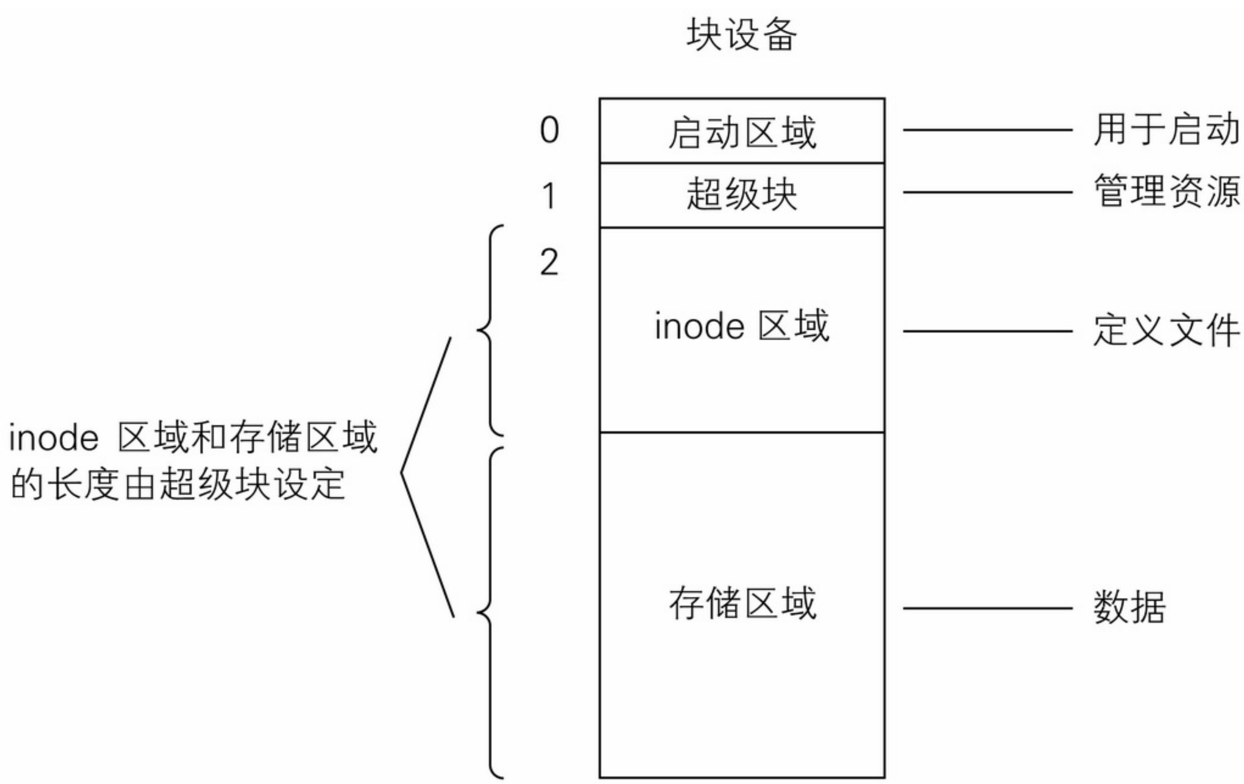


图 **9-3**　块设备区域

当系统管理者执行用户程序 `/etc/mkfs` 时，将对块设备的各区域进行初始化处理。

用于启动的区域

编号为 0 的块在系统启动时投入使用，而在其他的情况下不会被使用。详细请参照第 14 章。

超级块

编号为 1 的块容纳着超级块的信息。超级块对相应块设备的信息进行管理，用于获取资源及排他处理。此外，超级块通过空闲队列（free list）管理未使用的 inode 和属于未使用存储区域的块。新生成文件时，内核从空闲队列获取未使用区域的信息。反之，当删除文件时，将文件占用的区域返还给空闲队列。超级块的内容通过 `filsys` 结构体定义（代码清单 9-2，表 9-2 ）。

代码清单 **9-2**　**filsys** 结构体（**filsys.h)**

```
struct    filsys
{
    int    s_isize;
    int    s_fsize;
    int    s_nfree;
    int    s_free[100];
    int    s_ninode;
    int    s_inode[100];
    char   s_flock;
    char   s_ilock;
    char   s_fmod;
```

```
    char   s_ronly;
    int    s_time[2];
    int    pad[50];
};
```

表 **9-2**　**filsys** 结构体

成员	含义
`s_isize`	inode 区域的块数
`s_fsize`	存储区域的块数[1]
`s_nfree`	存储空闲队列中的有效元素个数
`s_free[]`	存储空闲队列
`s_ninode`	inode 空闲队列中的有效元素个数
`s_inode[]`	inode 空闲队列
`s_flock`	存储空闲队列的锁
`s_ilock`	inode 空闲队列的锁
`s_fmod`	更新标志
`s_ronly`	当前块设备为只读
`s_time[]`	当前时刻，或最后更新时刻
`pad[]`	填充字节

[1] 这里的原文有误。`s _ fsize` 指的应该是块设备全体（包括编号为 0 的块，超级块，inode 区域和存储区域）的块数。——译者注

inode 区域

从编号为 2 的块开始是长度为 `filsys.s_isize` 个块的 inode 区域。inode 区域用来保存 inode。inode 中有文件的定义，1 个 inode 对应 1 个文件。

inode 具有 inode 编号。从位于 inode 区域起始位置的 inode 开始，依次赋予 1、2、3……的编号。inode 的长度为 32 字节，1 个块（512 字节）可容纳 16 个 inode。具有某个 inode 编号的 inode 所在的块可表示为（inode 编号 +31 ）/16（向下取整），块中的偏移量可表示为 32×（（inode 编号 +31 ）%16 ）（图 9-4 ）。

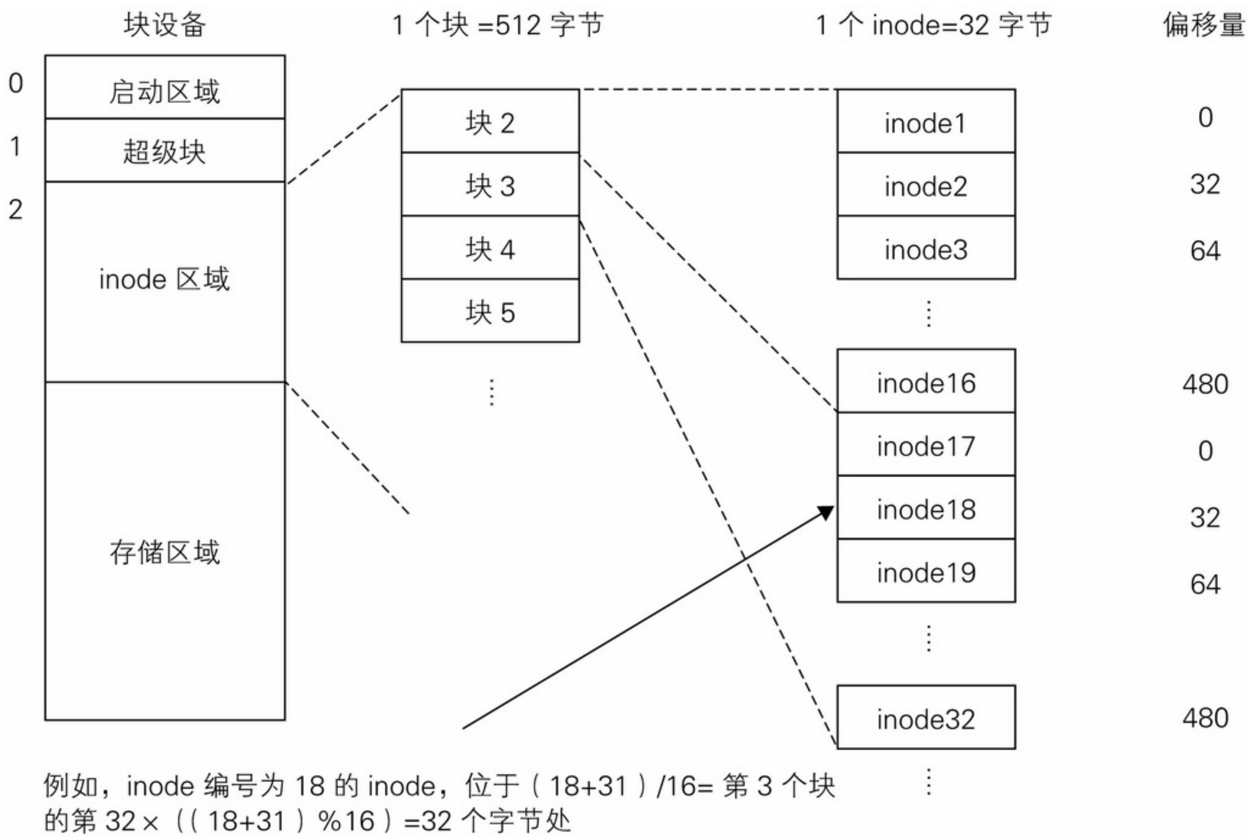


图 **9-4　inode** 区域

块设备中 inode 的数据形式由 `ino.h` 中的 `inode` 结构体定义（代码清单 9-3，表 9-3 ）。该结构体包括文件长度、权限、更新时间、数据所在地址等数据。请注意，inode 中不包含文件名。

如前所述，内存中 inode 的数据形式稍有不同。内存中的 inode 由 `inode.h` 中的 `inode` 结构体定义，详细请参照后文。内核不会使用 `ino.h` 中定义的 `inode` 结构体，而一部分对文件系统进行操作的用户程序将使用该结构体。

代码清单 **9-3　inode** 结构体（**ino.h)**

```
struct    inode
{
    int    i_mode;
    char   i_nlink;
    char   i_uid;
    char   i_gid;
    char   i_size0;
    char   *i_size1;
    int    i_addr[8];
    int    i_atime[2];
    int    i_mtime[2];
};

/* modes */
#define    IALLOC   0100000
#define    IFMT     060000
#define    IFDIR    040000
#define    IFCHR    020000
#define    IFBLK    060000
#define    ILARG    010000
#define    ISUID    04000
```

```
#define    ISGID    02000
#define    ISVTX    01000
#define    IREAD    0400
#define    IWRITE   0200
#define    IEXEC    0100
```

表 **9-3**　**inode** 结构体

成员	含义
i_mode	状态、控制信息。低位 9 比特表示权限。参见表 9-4
i_nlink	来自目录的参照数量
i_uid	用户 ID
i_gid	组 ID
i_size0	文件长度的高位 8 比特
*i_size1	文件长度的低位 16 比特
i_addr[]	使用的存储区域的块编号
i_atime[]	参照时间。inode.h 中的 inode 结构体不包含此成员
i_mtime[]	更新时间。inode.h 中的 inode 结构体不包含此成员

表 **9-4**　**inode** 的模式

模式	含义
IALLOC	当前 inode 已被分配
IFMT	调查格式时使用
IFDIR	目录
IFCHR	字符特殊文件
IFBLK	块特殊文件
ILARG	文件长度较大，使用间接参照

ISUID	SUID 比特
ISGID	SGID 比特
ISVTX	sticky 比特
IREAD	读取权限
IWRITE	写入权限
IEXEC	执行权限

存储区域

存储区域位于 inode 区域后方，其长度由 `filsys.s_fsize` [2] 定义。存储区域保存着文件中的数据。

> [2] 这里的原文有误。请参照 p190 的译者注。——译者注

`inode.i_addr[]` 表示某个文件究竟使用着存储区域的哪一个块。文件使用的块不一定是连续的，但是从用户的角度来看，文件可以被看做是占用了一块连续的区域（图 9-5 ）。

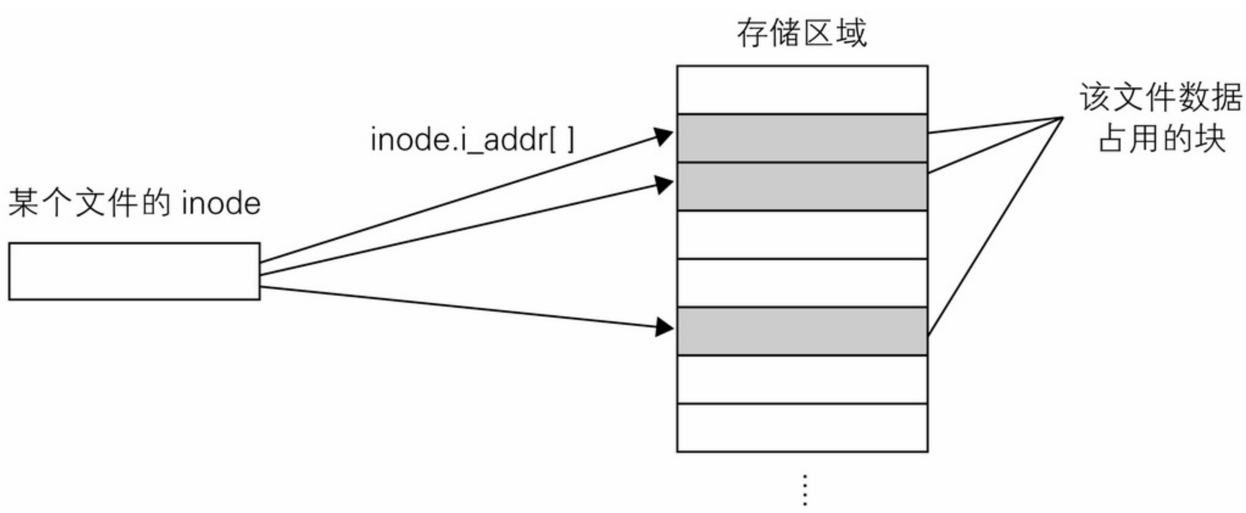


图 **9-5　inode** 和存储区域

9.3　挂载

mount 结构体

被挂载的块设备（的文件系统）通过 `mount` 结构体的数组 `mount[]` 进行管理（代码清单 9-4，代码清单 9-5，表 9-5 ）。

挂载块设备的文件系统之后，将被分配新的 `mount[]` 的元素。将 `mount.m_dev` 设定为该块设备的设备编号，代表挂载点的 `inode[]` 元素（`inode[]` 用于管理读取至内存的 inode。详细请参见下文）通过 `mount.m_inodep` 指定。设置代表挂载点的 `inode[]` 元素的 `IMOUNT` 标志位，表示该元素为挂载点。此外，超级块的数据拷贝至 `NODEV` 块设备缓冲区，通过 `mount.m_bufp` 可访问该缓冲区。

内核在遍历文件路径时，如果遇到挂载点，将取得相应的 mount[] 元素，然后移动到被挂载的块设备的根目录。mount[] 用于关联挂载点和被挂载的设备（图 9-6 ）。

代码清单 **9-4　mount**（**system.h)**

```
struct    mount
{
    int    m_dev;
    int    *m_bufp;
    int    *m_inodp;
} mount[NMOUNT];
```

代码清单 **9-5　NMOUNT**（**param.h)**

```
#define    NMOUNT    5
```

表 **9-5　mount** 结构体

成员	含义
m_dev	被挂载的块设备的设备编号
*m_bufp	指向块设备缓冲区的指针，该缓冲区容纳着被复制的超级块的数据
*m_inodp	代表挂载点的 inode[] 元素

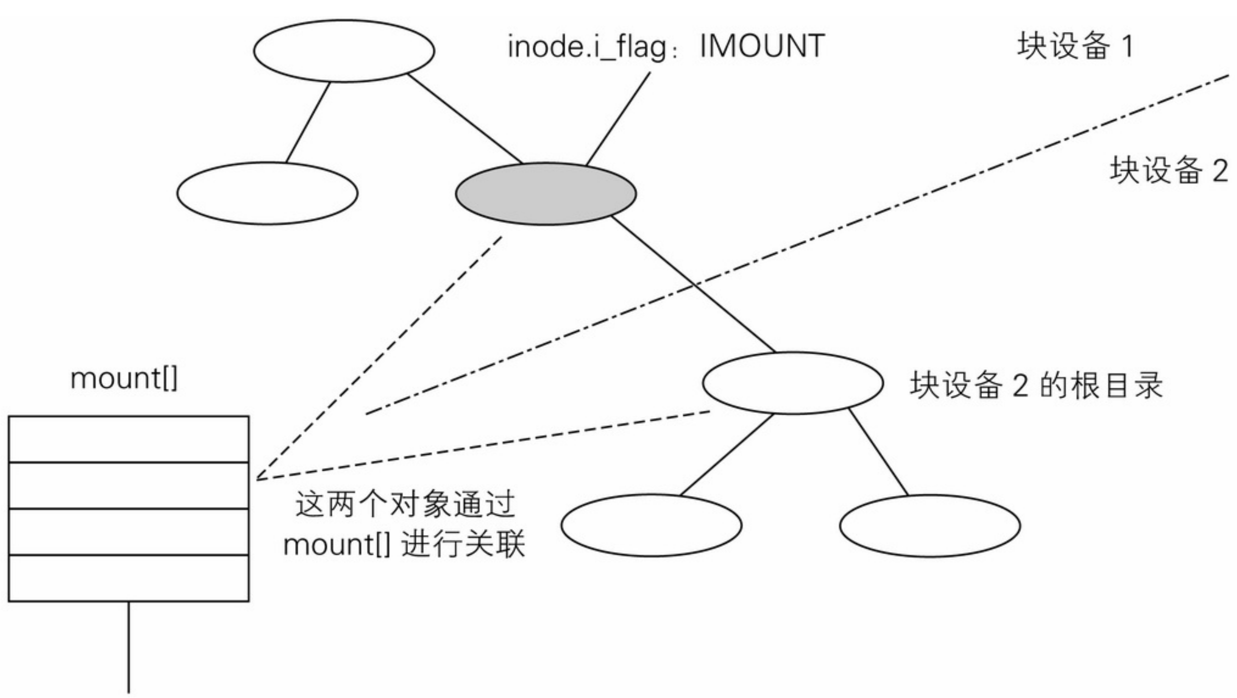


图 **9-6** 挂载

系统调用 **mount**

系统调用 mount 用于挂载块设备，并向 mount[] 元素追加块设备的信息。系统调用 mount 具有 3 个参数，分别为与挂载设备对应的特殊文件的路径、挂载点的路径和读写标志。

系统调用首先检查特殊文件的路径是否恰当，此处理调用 getmdev() 。如果路径恰当则返回设备编号。此外，确认挂载点当前没有任何人访问，并确认不是字符设备的特殊文件。

此后，从 mount[] 中寻找空的元素，并对该元素进行初始化处理。然后为代表挂载点的 inode[] 元素设置 IMOUNT 标志位。

smount() 是系统调用 mount 的处理函数（表 9-6，代码清单 9-6 ）。

表 **9-6** 系统调用 **mount** 的参数

成员	含义
u.u_arg[0]	与挂载设备相对应的特殊文件的路径
u.u_arg[1]	挂载点的路径
u.u_arg[2]	读写标志。如果为 1，表示以只读方式挂载

代码清单 **9-6** **smount()(ken/sys3.c)**

```
smount()
{
    int d;
    register *ip;
    register struct mount *mp, *smp;
    extern uchar;

    d = getmdev();
    if(u.u_error)
        return;
    u.u_dirp = u.u_arg[1];
    ip = namei(&uchar, 0);
    if(ip == NULL)
        return;
    if(ip->i_count!=1 || (ip->i_mode&(IFBLK&IFCHR))!=0)
        goto out;
    smp = NULL;
    for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++) {
        if(mp->m_bufp != NULL) {
            if(d == mp->m_dev)
                goto out;
        } else
        if(smp == NULL)
            smp = mp;
    }
    if(smp == NULL)
        goto out;
    (*bdevsw[d.d_major].d_open)(d, !u.u_arg[2]);
    if(u.u_error)
        goto out;
    mp = bread(d, 1);
    if(u.u_error) {
        brelse(mp);
```

```
34          goto out1;
35      }
36      smp->m_inodp = ip;
37      smp->m_dev = d;
38      smp->m_bufp = getblk(NODEV);
39      bcopy(mp->b_addr, smp->m_bufp->b_addr, 256);
40      smp = smp->m_bufp->b_addr;
41      smp->s_ilock = 0;
42      smp->s_flock = 0;
43      smp->s_ronly = u.u_arg[2] & 1;
44      brelse(mp);
45      ip->i_flag =| IMOUNT;
46      prele(ip);
47      return;
48
49 out:
50      u.u_error = EBUSY;
51 out1:
52      iput(ip);
53 }
```

8~10 执行 getmdev() 取得准备挂载的块设备的大编号。

11~14 取得代表挂载点的 inode[] 元素。

15~16 确认无人访问代表挂载点的 inode[] 元素，同时确认该元素不为字符设备或块设备的特殊文件。

17~27 寻找 mount[] 中未使用的元素。如果不存在未使用元素，或者准备挂载的设备已被挂载，则认为出错。

28~30 进行打开挂载设备的处理。

31~35 将挂载设备的超级块读入缓冲区。

36~43 对所取得的 mount[] 元素进行初始化处理。通过 getblk() 取得尚未分配给任何 设备的块设备缓冲区。将挂载设备的超级块的数据复制到该缓冲区，并注册到 mount[] 元素。

44 释放读取超级块的缓冲区。

45 设置代表挂载点的 inode[] 元素的 IMOUNT 标志位。

46 解除代表挂载点的 inode[] 元素的锁。

getmdev()

getmdev() 是执行 smount() 和 sumount() 共用处理的函数（代码清单 9-7 ）。该函数首先检查作为系统调用第 1 个参数的特殊文件的路径是否合适，如果合适，则返回该设备的设备编号。

代码清单 **9-7 getmdev()**（**ken/sys3.c)**

```
getmdev()
{
    register d, *ip;
    extern uchar;

    ip = namei(&uchar, 0);
    if(ip == NULL)
        return;
    if((ip->i_mode&IFMT) != IFBLK)
```

```
10         u.u_error = ENOTBLK;
11     d = ip->i_addr[0];
12     if(ip->i_addr[0].d_major >= nblkdev)
13         u.u_error = ENXIO;
14     iput(ip);
15     return(d);
16 }
```

6~8　取得与用户输入的第 1 个参数的路径相对应的 `inode[]` 元素。

9~10　如果不为块设备的特殊文件则出错。

11　特殊文件的 `inode.i_addr[0]` 中保存着设备编号。

12~13　如果设备的大编号的值过大则出错。

14~15　释放 `inode[]` 元素，返回设备编号。

系统调用**umount**

系统调用 `umount` 用于卸载指定块设备的文件系统。它从 `mount[]` 中释放相应元素，并清除代表挂载点的 `inode[]` 元素的 `IMOUNT` 标志位。但是，如果准备卸载的设备仍处于使用中的状态，则中断卸载处理。

`sumount()` 为系统调用 `umount` 的处理函数（表 9-7，代码清单 9-8 ）。系统调用 `umount` 的参数为挂载点的路径。

表 **9-7**　系统调用 **umount** 的参数

参数	含义
u.u_arg[0]	挂载点的路径

代码清单 **9-8**　**sumount()**（**ken/sys3.c)**

```
sumount()
{
    int d;
    register struct inode *ip;
    register struct mount *mp;

    update();
    d = getmdev();
    if(u.u_error)
        return;
    for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
        if(mp->m_bufp!=NULL && d==mp->m_dev)
            goto found;
    u.u_error = EINVAL;
    return;

found:
    for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
        if(ip->i_number!=0 && d==ip->i_dev) {
            u.u_error = EBUSY;
            return;
        }
    (*bdevsw[d.d_major].d_close)(d, 0);
    ip = mp->m_inodp;
```

```
25     ip->i_flag =& ~IMOUNT;
26     iput(ip);
27     ip = mp->m_bufp;
28     mp->m_bufp = NULL;
29     brelse(ip);
30 }
```

7　卸载前先将内存中的数据保存至块设备。

8~10　取得准备卸载的设备的设备编号。

11~15　在 `mount[]` 中寻找与卸载设备相对应的元素。

18~22　在 `inode[]` 中寻找属于卸载设备的元素。如果存在则说明该设备仍处于使用中的状态，此时将终止卸载处理。

23　进行关闭卸载设备的处理。

24~26　清除与挂载点相对的 `inode[]` 元素的 `IMOUNT` 标志位，并释放该元素。

27~28　释放 `mount[]` 元素。

29　释放块设备缓冲区，该缓冲区容纳着被卸载设备的超级块的数据。

9.4　inode 的获取和释放

inode[]

内存中的 inode 通过 `inode` 结构体的数组 `inode[]` 管理（代码清单 9-9，代码清单 9-10，表 9-8 ）。此处的 `inode` 结构体由 `inode.h` 定义，代表被读取至内存的 inode 的数据结构。

内核从块设备读取 inode 的数据并将其转换为 `inode[]` 数组元素的形式，通过操作该元素实现对 inode 的操作。`inode[]` 元素以设备编号和 inode 编号进行命名。所有块设备的 inode 由同一个 `inode[]` 管理。此外，`inode[]` 元素通过参照计数器来记录该元素是否处于使用中的状态（图 9-7 ）。

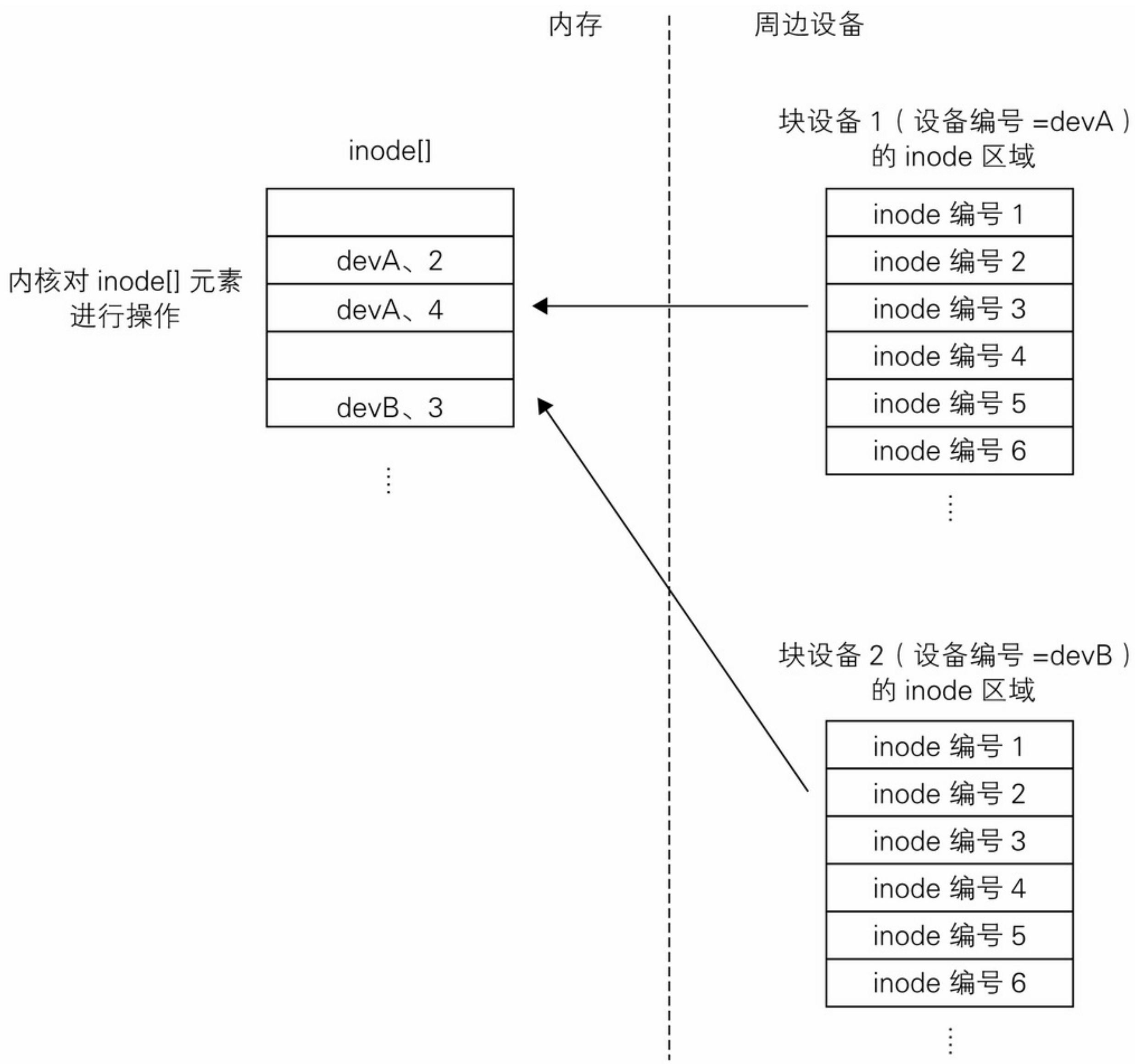


图 **9-7　inode[]**

`inode[]` 可以起到缓存的作用。`inode[]` 中的某个元素即使不再使用也不会马上被删除。在该元素再次被其他 inode 使用之前，其数据仍然保存在 `inode[]` 中。因此，`inode[]` 中保存着当前正在使用的 inode，以及最近曾经使用过的 inode 的数据。当所需的 inode 在 `inode[]` 中存在时，无需再从块设备中读取。

此外，通过对 `inode[]` 元素进行排他处理，可以防止多个进程操作同一文件时引发的冲突。

代码清单 **9-9　inode（inode.h)**

```
struct    inode
{
    char    i_flag;
    char    i_count;
    int     i_dev;
    int     i_number;
    int     i_mode;
    char    i_nlink;
    char    i_uid;
```

```
    char    i_gid;
    char    i_size0;
    char    *i_size1;
    int     i_addr[8];
    int     i_lastr;
} inode[NINODE];

/* flags */
#define    ILOCK     01
#define    IUPD      02
#define    IACC      04
#define    IMOUNT    010
#define    IWANT     020
#define    ITEXT     040

/* modes */
#define    IALLOC    0100000
#define    IFMT      060000
#define    IFDIR     040000
#define    IFCHR     020000
#define    IFBLK     060000
#define    ILARG     010000
#define    ISUID     04000
#define    ISGID     02000
#define    ISVTX     01000
#define    IREAD     0400
#define    IWRITE    0200
#define    IEXEC     0100
```

代码清单 **9-10　NINODE**（**param.h)**

```
#define    NINODE    100
```

表 **9-8　inode** 结构体

成员	含义
`i_flag`	标志。参照表 9-9。在 ino.h 中定义的 inode 结构体中不存在此成员
`i_count`	参照计数器。在 ino.h 中定义的 inode 结构体中不存在此成员
`i_dev`	设备编号。在 ino.h 中定义的 inode 结构体中不存在此成员
`i_number`	inode 编号。0 表示未使用的元素。在 ino.h 中定义的 inode 结构体中不存在此成员
`i_mode`	状态、控制信息。低位 9 比特表示权限。参照表 9-4
`i_nlink`	来自目录的参照数量
`i_uid`	用户 ID
`i_gid`	组 ID

i_size0	文件长度的高位 8 比特
*i_size1	文件长度的低位 16 比特
i_addr[]	使用的存储区域的块编号
i_lastr	在此之前读取的逻辑块编号。用于预读取功能。在 ino.h 中定义的 inode 结构体中不存在此成员

表 **9-9**　**inode** 的标志位

标志位	含义
ILOCK	已加锁
IUPD	已更新
IACC	已参照
IMOUNT	该 inode[] 元素为挂载点
IWANT	存在等待解锁该 inode[] 元素的进程
ITEXT	该 inode[] 元素作为代码段分配给进程

iget()

iget() 用于取得 inode[] 元素（表 9-10，代码清单 9-11 ）。

如果 inode[] 中不存在所需要的元素，则获取新的元素并命名，然后将从块设备读取的 inode 的数据复制到上述新元素，并返回该元素。

如果 inode[] 中存在所需要的元素，递增参照计数器并返回该元素。由于无需读取块设备处理速度也相应提高。但是如果该元素处于被加锁的状态，则进入睡眠状态直至元素被解锁。

如果设置了 inode[] 元素的 IMOUNT 标志位，则取得 mount[] 元素，并返回与对应设备的根目录相对应的 inode[] 元素。通过该处理，使得利用同一命名空间管理多个挂载设备成为可能。[3]

> [3] 即使得在同一个目录树下同时载入多个挂载设备的根目录成为可能。——审校者注

表 **9-10**　**iget()** 的参数

参数	含义
dev	设备编号
ino	inode 编号

代码清单 **9-11　iget()**（**ken/iget.c)**

```
1 iget(dev, ino)
2 {
3     register struct inode *p;
4     register *ip2;
5     int *ip1;
6     register struct mount *ip;
7
8 loop:
9     ip = NULL;
10     for(p = &inode[0]; p < &inode[NINODE]; p++) {
11         if(dev==p->i_dev && ino==p->i_number) {
12             if((p->i_flag&ILOCK) != 0) {
13                 p->i_flag =| IWANT;
14                 sleep(p, PINOD);
15                 goto loop;
16             }
17             if((p->i_flag&IMOUNT) != 0) {
18                 for(ip = &mount[0]; ip < &mount[NMOUNT]; ip++)
19                 if(ip->m_inodp == p) {
20                     dev = ip->m_dev;
21                     ino = ROOTINO;
22                     goto loop;
23                 }
24                 panic("no imt");
25             }
26             p->i_count++;
27             p->i_flag =| ILOCK;
28             return(p);
29         }
30         if(ip==NULL && p->i_count==0)
31             ip = p;
32     }
33     if((p=ip) == NULL) {
34         printf("Inode table overflow\n");
35         u.u_error = ENFILE;
36         return(NULL);
37     }
38     p->i_dev = dev;
39     p->i_number = ino;
40     p->i_flag = ILOCK;
41     p->i_count++;
42     p->i_lastr = -1;
43     ip = bread(dev, ldiv(ino+31,16));
44     if (ip->b_flags&B_ERROR) {
45         brelse(ip);
46         iput(p);
47         return(NULL);
48     }
49     ip1 = ip->b_addr + 32*lrem(ino+31, 16);
50     ip2 = &p->i_mode;
51     while(ip2 < &p->i_addr[8])
52         *ip2++ = *ip1++;
53     brelse(ip);
54     return(p);
55 }
```

10　从起始位置遍历 inode[]，寻找未使用的元素。同时确认对象元素是否在 inode[] 中已存在。

11　找到与参数 dev 、ino 相对应的元素时的处理。

12~16　如果对象元素被加锁，则设置 IWANT 标志位（表示存在等待该 inode[] 元素的进程）并进入睡眠状态。唤醒后返回 loop 再次尝试。

17~25　如果设置了 `IMOUNT` 标志位，则从 `mount[]` 找到对应的元素，将 `dev` 设定为被挂载设备的设备编号，将 `ino` 设定为 `ROOTINO`（1），然后返回 `loop` 并再次尝试。`ROOTINO` 表示根目录的 inode 编号（代码清单 9-12）。

代码清单 **9-12　ROOTINO**（**param.h)**

```
#define    ROOTINO    1
```

26~28　如果对象元素既未被加锁，也没有设置 `IMOUNT` 标志位的话，递增该元素的参照计数器并加锁，然后返回该元素。

30~31　将 `inode[]` 中距起始位置最近的未使用元素赋予 `ip` 。

33~37　如果在 `inode[]` 中未找到与参数 `dev` 、`ino` 相对应的元素，且 `inode[]` 中不存在未使用元素时，按出错处理。

38~42　为 `inode[]` 中未使用的元素命名。递增参照计数器并对该元素加锁（同时清除其他标志位），然后将预读取逻辑块编号设置为 -1（无效）。

43　读取块设备中该 inode 所在的块。

44~48　在读取块设备发生错误时进行错误处理。

49~52　将块设备中 inode 的 `i_mode` 至 `i_addr` 数据复制到 `inode[]` 元素。

53~54　释放缓冲区，返回 `inode[]` 元素。

iput()

`iput()` 用来递减 `inode[]` 元素的参照计数器的值（表 9-11，代码清单 9-13 ）。当参照计数器的值为 0 时，将 `inode[]` 元素的内容写回块设备。此外，当文件不再被任何目录参照时进行删除文件的处理。

文件的删除处理包含以下内容。

- 执行 `itrunc()` ，释放使用中的存储区域，将文件长度和 `inode.i_addr[]` 清 0
- 将 `inode.i_mode` 清 0
- 执行 `ifree()` ，将 inode 编号返还至空闲队列

表 **9-11　iput()** 的参数

参数	含义
p	`inode[]` 元素

代码清单 **9-13　iput()**（**ken/iget.c)**

```
iput(p)
struct inode *p;
{
    register *rp;
```

```
5
6     rp = p;
7     if(rp->i_count == 1) {
8         rp->i_flag =| ILOCK;
9         if(rp->i_nlink <= 0) {
10            itrunc(rp);
11            rp->i_mode = 0;
12            ifree(rp->i_dev, rp->i_number);
13        }
14        iupdat(rp, time);
15        prele(rp);
16        rp->i_flag = 0;
17        rp->i_number = 0;
18    }
19    rp->i_count--;
20    prele(rp);
21 }
```

7 递减 inode[] 元素参照计数器的值，使其变为 0 时的处理。

8 将 inode[] 元素加锁。

9~13 当文件不再被任何目录参照时进行删除文件的处理。

14 将 inode[] 元素的数据写回块设备。

15 将 inode[] 元素解锁。

16~17 将 inode[] 元素的 i_flag 和 i_number 清 0。

19 递减参照计数器的值。

20 将 inode[] 元素解锁。这是针对在 iput() 之外加锁的处理。

iupdat()

iupdat() 将 inode[] 元素的内容写入块设备（表 9-12，代码清单 9-14 ）。写入处理只在设置了 inode[] 元素的更新标志位（IUPD ）或参照标志位（IACC ）时才会进行。

表 **9-12 iupdat()** 的参数

参数	含义
p	inode[] 元素
tm	当前时间

代码清单 **9-14 iupdat()**（**ken/iget.c)**

```
iupdat(p, tm)
int *p;
int *tm;
{
    register *ip1, *ip2, *rp;
    int *bp, i;

```

```
 8     rp = p;
 9     if((rp->i_flag&(IUPD|IACC)) != 0) {
10         if(getfs(rp->i_dev)->s_ronly)
11             return;
12         i = rp->i_number+31;
13         bp = bread(rp->i_dev, ldiv(i,16));
14         ip1 = bp->b_addr + 32*lrem(i, 16);
15         ip2 = &rp->i_mode;
16         while(ip2 < &rp->i_addr[8])
17             *ip1++ = *ip2++;
18         if(rp->i_flag&IACC) {
19             *ip1++ = time[0];
20             *ip1++ = time[1];
21         } else
22             ip1 =+ 2;
23         if(rp->i_flag&IUPD) {
24             *ip1++ = *tm++;
25             *ip1++ = *tm;
26         }
27         bwrite(bp);
28     }
29 }
```

9　更新标志位或设置参照标志位时的处理。

10~11　执行 `getfs()` 读取超级块，如果为只读则直接返回。

12~13　从块设备中读取 inode 所在的块。

14~17　更新位于块设备中的 inode，更新的范围为 `i_mode` 至 `i_addr`。

18~20　如果设置了参照标志位，则更新 inode 的参照时间。

23~26　如果设置了更新标志位，则更新 inode 的更新时间。

27　执行 `bwrite()`，将包括已更新的 inode 在内的块写入块设备。

9.5　inode 与存储区域的对应关系

某个文件使用的存储区域的块由 `inode.i_addr[]` 管理。inode 与存储区域存在下述 3 种对应关系。

- 直接参照
- 间接参照
- 双重间接参照

上述 3 种参照方法能够管理的文件长度依次增大。直接参照在 inode 的 `ILARG` 标志位为 0 时使用，存储区域的块编号直接保存于 `inode.i_addr[]` 中。将文件中希望访问的字节偏移量 N 除以 512（= 块长度）时的商用 b 来表示。（`ILARG` 标志位为 0 时）b 一定小于或等于 7，`i_addr[b]` 中保存着相应的块编号。直接参照能够管理的最大文件长度为 512 字节 ×8 = 4KB（图 9-8 ）。

间接参照适用于 inode 的 `ILARG` 标志位为 1 的情况，且前述的 b 除以 256（1 个字 ×256 = 512 字节）的商小于等于 6。如果将 b 除以 256 的商记为 i，`inode.i_addr[i]` 则保存着作为间接参照块的存储区域中的块编号。前述 N 的除法运算得到的余数可以决定间接参照块中的偏移量，其中容纳着保存目标数据的存储区域的块编号。间接参照所能管理的最大文件长度为 512 字节 ×256×7 = 896KB。

双重间接参照适用于 inode 的 `ILARG` 标志位为 1，且前述的 i 的值为 7 的情况。`inode.i_addr[7]` 容纳着

间接参照块的起始地址，而间接参照块的各个数据又分别指向双重间接参照块的地址。双重间接参照在理论上可以管理的文件长度为 512 字节×256×256 = 32MB（再加上通过 `inode.i_addr[6-0]` 的间接参照管理的 896KB），但是由于文件长度是以 24 比特的形式保存的，因此实际上只能管理 16MB 的文件（图 9-9）。

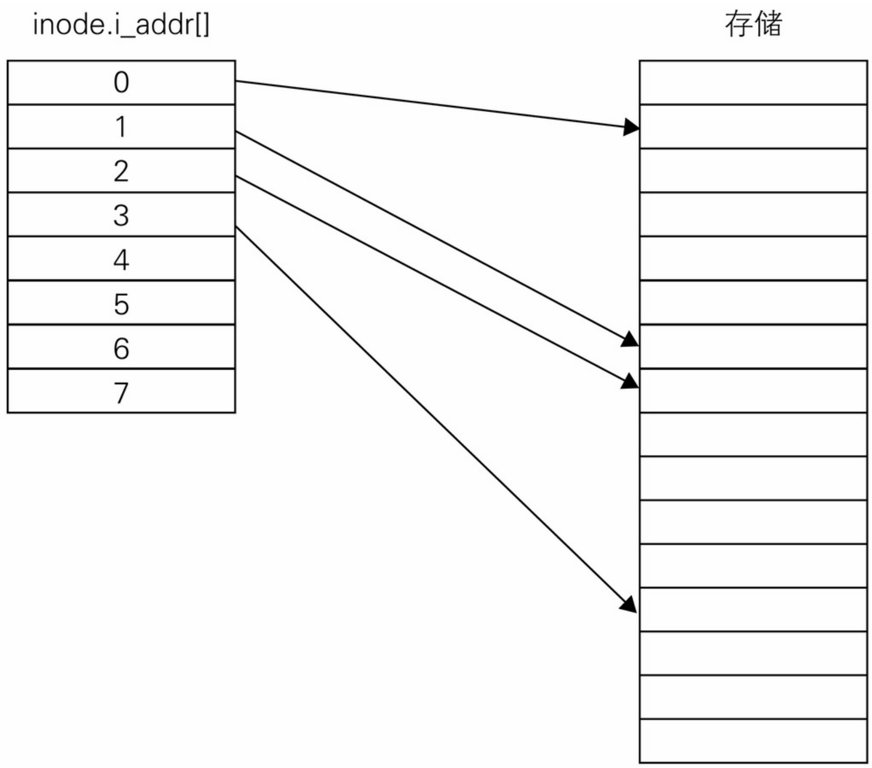
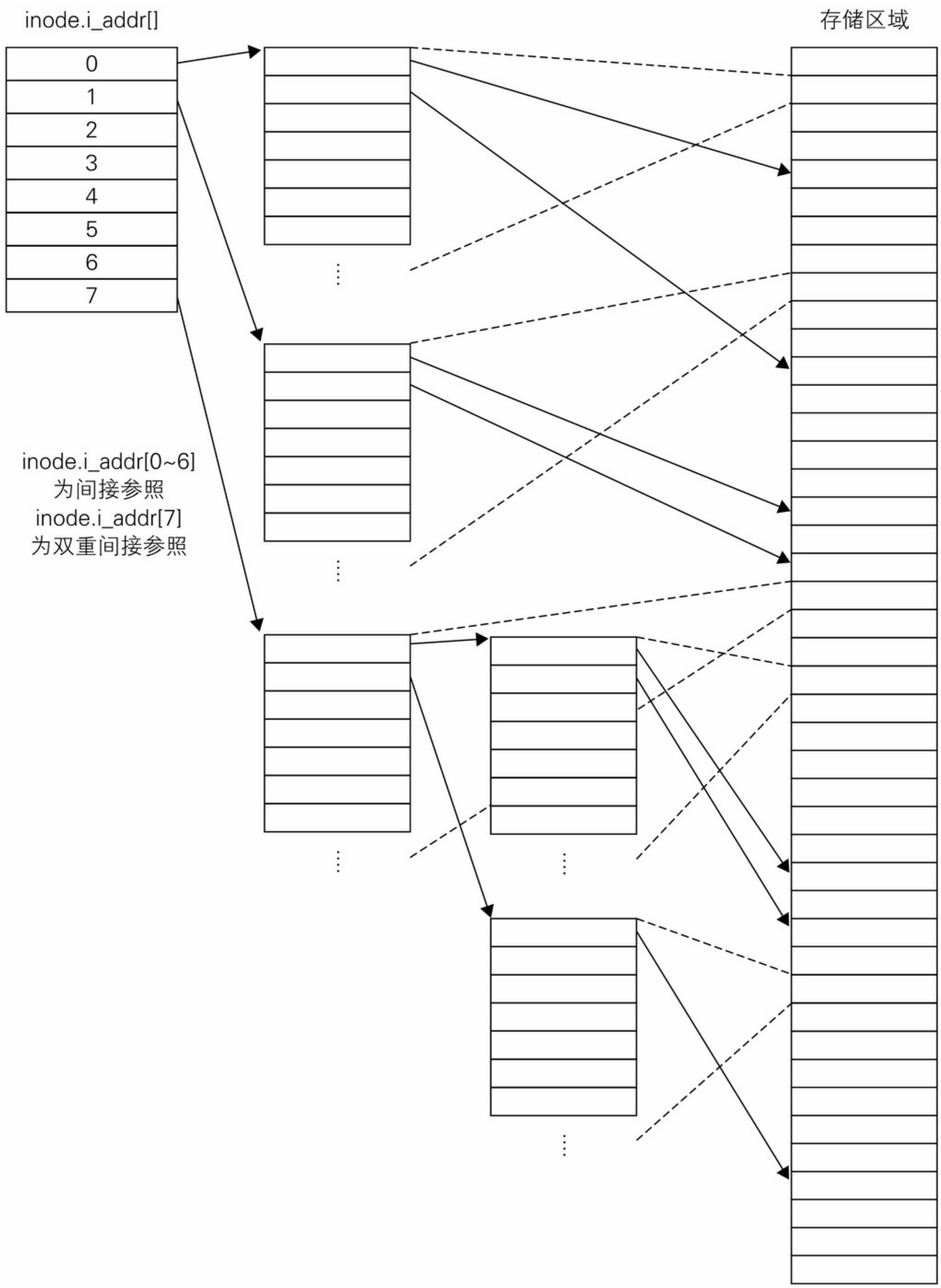


图 **9-8**　直接参照



inode.i_addr[]
存储区域
0
1
2
3
4
5
6
7
inode.i_addr[0~6]
为间接参照
inode.i_addr[7]
为双重间接参照

图 **9-9**　间接参照

在之后的说明中，会将通过文件的字节偏移量除以 512 的商得到的块编号称为逻辑块编号，将实际保存数据的存储区域的块编号称为物理块编号。

bmap()

`bmap()` 是将逻辑块编号变换为物理块编号的函数（表 9-13，代码清单 9-15 ）。

表 **9-13**　**bmap()** 的参数

参数	含义
ip	inode[] 元素
bn	逻辑块编号

代码清单 **9-15**　**bmap()**（**ken/subr.c)**

```
bmap(ip, bn)
struct inode *ip;
int bn;
{
    register *bp, *bap, nb;
    int *nbp, d, i;

    d = ip->i_dev;
    if(bn & ~077777) {
        u.u_error = EFBIG;
        return(0);
    }

    /* 直接参照 */
    if((ip->i_mode&ILARG) == 0) {
        if((bn & ~7) != 0) {
            if ((bp = alloc(d)) == NULL)
                return(NULL);
            bap = bp->b_addr;
            for(i=0; i<8; i++) {
                *bap++ = ip->i_addr[i];
                ip->i_addr[i] = 0;
            }
            ip->i_addr[0] = bp->b_blkno;
            bdwrite(bp);
            ip->i_mode =| ILARG;
            goto large;
        }
        nb = ip->i_addr[bn];
        if(nb == 0 && (bp = alloc(d)) != NULL) {
            bdwrite(bp);
            nb = bp->b_blkno;
            ip->i_addr[bn] = nb;
            ip->i_flag =| IUPD;
        }
        rablock = 0;
        if (bn<7)
            rablock = ip->i_addr[bn+1];
        return(nb);
    }

    /* 间接参照 */
    large:
    i = bn>>8;
```

```
45     if(bn & 0174000)
46         i = 7;
47     if((nb=ip->i_addr[i]) == 0) {
48         ip->i_flag =| IUPD;
49         if ((bp = alloc(d)) == NULL)
50             return(NULL);
51         ip->i_addr[i] = bp->b_blkno;
52     } else
53         bp = bread(d, nb);
54     bap = bp->b_addr;
55 
56     /* 双重间接参照 */
57     if(i == 7) {
58         i = ((bn>>8) & 0377) - 7;
59         if((nb=bap[i]) == 0) {
60             if((nbp = alloc(d)) == NULL) {
61                 brelse(bp);
62                 return(NULL);
63             }
64             bap[i] = nbp->b_blkno;
65             bdwrite(bp);
66         } else {
67             brelse(bp);
68             nbp = bread(d, nb);
69         }
70         bp = nbp;
71         bap = bp->b_addr;
72     }
73 
74     i = bn & 0377;
75     if((nb=bap[i]) == 0 && (nbp = alloc(d)) != NULL) {
76         nb = nbp->b_blkno;
77         bap[i] = nb;
78         bdwrite(nbp);
79         bdwrite(bp);
80     } else
81         brelse(bp);
82     rablock = 0;
83     if(i < 255)
84         rablock = bap[i+1];
85     return(nb);
86 }
```

9~12 如果参数的逻辑块编号的值过大，则认为出错。

15 如果没有设置 `inode[]` 元素的 `ILARG` 标志位，则按照直接参照的方式进行处理。

16 直接参照时，参数 bn 的值应该小于或等于 7（`inode.i_addr[]` 的元素数）。如果大于 7，则切换至间接参照方式。这种情况只有在满足下述条件时才会发生，即对文件的写入操作由 `writei()` 进行，且文件长度大于 4KB。

17~18 执行 `alloc()`，从存储区域分配新的块，并取得相应的缓冲区。

19~23 将 `inode.i_addr[]` 内的数据复制到刚取得的缓冲区，然后将 `inode.i_addr[]` 内的数据清 0。

24 将 `inode.i_addr[0]` 设置为刚取得的块的块编号。

25 执行 `bdwrite()` 对块进行写入操作。`inode.i_addr[]` 中原本容纳的数据被输出。

26~27 设置 `inode[]` 元素的 `ILARG` 标志位，然后跳转至 `large`，进行间接参照处理。

29 从此处开始为一般的直接参照处理。首先取得由参数指定的逻辑块编号指向的 `inode.i_addr[]`

的值。

30~35 由于文件长度变大，此处需要取得新的块。如果从 inode.i_addr[] 取得的值为 0，且通过 alloc() 成功取得新的块（的缓冲区），则将新取得的块的块编号注册到 inode.i_addr[]，并设置 inode[] 元素的更新标志位。

36~38 注册预读取块编号。如果逻辑块编号小于 7，则注册与下一个逻辑块编号相对应的物理块编号。如果是从头开始按顺序处理文件内容等情况，则很有可能会立即对下一个逻辑块进行处理，因此，此处将其注册为预处理块。

39 返回物理块编号。

42 此处开始为间接参照处理。

44~46 将逻辑块编号向右移动 8 比特后的值赋予 i。间接参照时，inode.i_addr[] 的每个元素对应 256 个块，向右移动 8 比特后的值（=除以 256 的商）相当于 inode.i_addr[] 的数组下标。如果逻辑块编号的值大于等于 0174000（=2048=256×8）则采用双重间接参照，此时需要使用 inode.i_addr[7]，因此将 i 的值设置为 7。

47~54 如果 inode.i_addr[i] 的值为 0，则设置 inode[] 元素的更新标志位。通过 alloc() 从存储区域取得新的块（的缓冲区），并将取得的块的块编号赋予inode.i_addr[i]。如果 inode.i_addr[i] 的值不为 0，则通过 bread() 读取该块的内容。

57 此处开始到第 72 行为双重间接参照处理。

58 计算第一级参照块中相应的块编号。从向右移动 8 比特后的值中减去 7，表示从逻辑块编号中减去 1792（=7×256）。通过这个计算可以取得在双重间接参照（inode.i_addr[7]）第一级参照块中的偏移量。

59~65 如果第一级参照块中相应元素的块编号为 0 时，通过 alloc() 从存储区域取得新的块（的缓冲区），并将取得的块的块编号分配给相应元素，然后执行 bdwrite()，对块设备进行延迟写入。

66~69 相应元素持有 0 以外的块编号时，执行 bread() 读取该块的内容。

70~71 用取得的块设备缓冲区更新变量 bp 和 bap。对第二级参照块的遍历处理由此后一般的间接参照处理进行。

74 此处开始为一般间接参照块的读取处理。i 被设定为逻辑块编号的低位比特，相当于间接参照块中的偏移量。

75~81 由间接参照块中的偏移量取得块编号。如果为 0，则尝试通过 alloc() 从存储区域取得新的块。将取得的块的块编号分配给偏移量，然后执行 bdwrite() 对取得的块和间接参照块进行延迟写入。如果块编号不为 0 则释放间接参照块。

82~84 将预读取块编号设定为与下一个逻辑块编号相对应的物理块编号。

85 返回物理块编号。

itrunc()

itrunc() 将参数 inode[] 元素使用的存储区域的块编号返还给空闲队列（表 9-14，代码清单 9-16），然后将文件长度和 inode.i_addr[] 全部清 0。

间接参照时，将相应的间接块也一并返还到空闲队列。文件数据本身仍保存在块设备的存储区域中，直到该区域被别的数据覆盖。

表 **9-14　itrunc()** 的参数

参数	含义
ip	inode[] 元素

代码清单 **9-16　itrunc()**（**ken/iget.c)**

```
 1 itrunc(ip)
 2 int *ip;
 3 {
 4     register *rp, *bp, *cp;
 5     int *dp, *ep;
 6 
 7     rp = ip;
 8     if((rp->i_mode&(IFCHR&IFBLK)) != 0)
 9         return;
10     for(ip = &rp->i_addr[7]; ip >= &rp->i_addr[0]; ip--)
11     if(*ip) {
12         if((rp->i_mode&ILARG) != 0) {
13             bp = bread(rp->i_dev, *ip);
14             for(cp = bp->b_addr+512; cp >= bp->b_addr; cp--)
15             if(*cp) {
16                 if(ip == &rp->i_addr[7]) {
17                     dp = bread(rp->i_dev, *cp);
18                     for(ep = dp->b_addr+512; ep >= dp->b_addr; ep--)
19                     if(*ep)
20                         free(rp->i_dev, *ep);
21                     brelse(dp);
22                 }
23                 free(rp->i_dev, *cp);
24             }
25             brelse(bp);
26         }
27         free(rp->i_dev, *ip);
28         *ip = 0;
29     }
30     rp->i_mode =& ~ILARG;
31     rp->i_size0 = 0;
32     rp->i_size1 = 0;
33     rp->i_flag =| IUPD;
34 }
```

8~9　如果是特殊文件，则不做任何处理立即返回。

10~11　遍历 inode.i_addr[] 。

12　如果设置了 ILARG 标志位，则遍历间接参照块。

13　读取与 inode.i_addr[] 元素指向的块编号相对应的块。

14~26　参照块容纳着块编号的队列。从后向前遍历队列，通过执行 free() ，每次都返还一个块编号给空闲队列。因为 inode.i_addr[7] 供双重间接参照使用，首先读取各块编号所指向的块，再通过 free() 将其中保存的块编号队列返还给空闲队列。

27~28　执行 free() ，将 inode.i_addr[] 元素指向的块编号返还给空闲队列，并将 inode.i_addr[] 元素的值置为 0。

30~33 重置 `inode[]` 元素的 `ILARG` 标志位，将文件长度设为 0，并设置更新标志位。

9.6 分配块设备中的块

块设备中的 inode 和存储区域中的块，分别由超级块中的 **inode** 空闲队列（**`filsys. s_inode[]`**）和存储空闲队列（**`filsys.s_free[]`**） 管理。`filsys.s_inode[]` 和 `filsys.s_free[]` 的元素数为 100，分别保存着未分配的 inode 编号和未分配的存储区域的块编号。而 `filsys.s_ninode` 和 `filsys.s_nfree` 分别容纳 inode 空闲队列和存储空闲队列中编号的数量。这两个变量的值变为 0 时，需要从块设备取得未分配的 inode 编号和块编号补充空闲队列。

ialloc()

`ialloc()` 用来分配块设备中 inode 区域的未分配 inode（表 9-15，代码清单 9-17）。首先取得由 `filsys.s_ninode` 指向的 `filsys.s_inode[]` 元素，然后通过 inode 编号取得 `inode[]` 元素，同时递减 `filsys.s_ninode`。将 `filsys.s_inode[]` 看做存放未使用 inode 编号的栈，将 `filsys.s_ninode` 看做栈指针，可能更容易理解（图 9-10）。

`filsys.s_inode[]` 为空时，从块设备的 inode 区域的头部（块编号 2）开始按顺序检查 inode，将未使用的 inode 编号追加至 `filsys.s_inode[]`（图 9-11）。

由于当 `filsys.s_inode[]` 为空时才需要访问块设备，这与每当收到分配块设备 inode 的请求时则立即访问块设备的做法相比，在性能上具有优势。

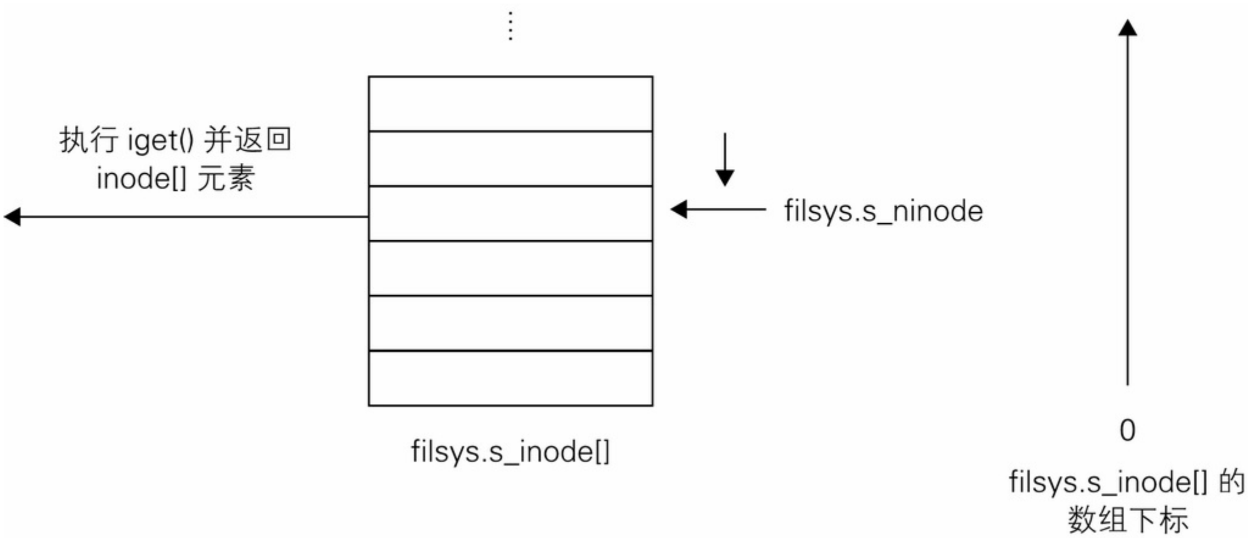


图 **9-10** **ialloc()**

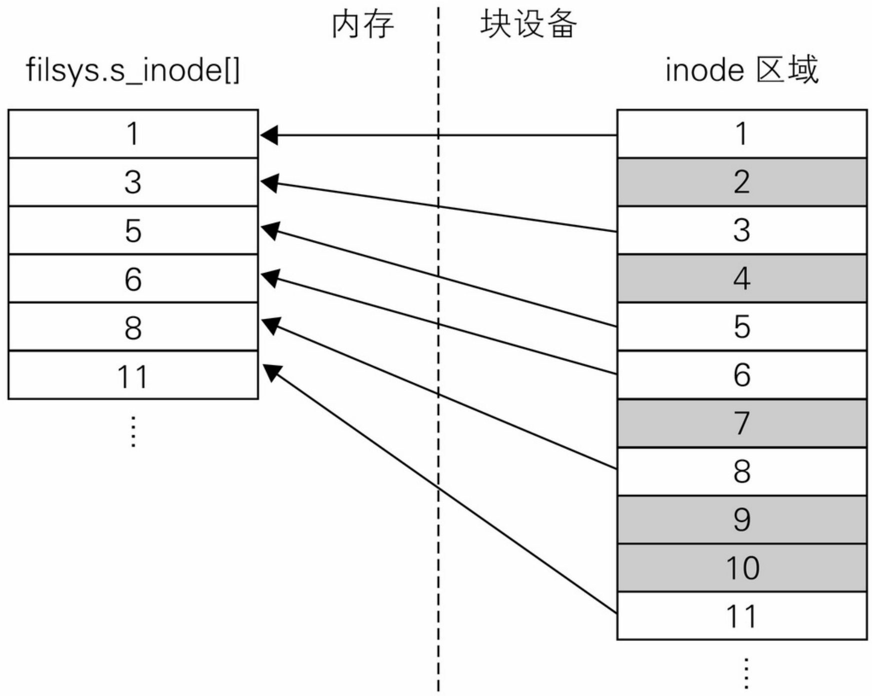


图 **9-11**　对 **inode** 空闲队列进行补**}**

表 **9-15**　**ialloc()** 的参数

参数	含义
dev	设备编号

代码清单 **9-17**　**ialloc()**（**ken/alloc.c)**

```
ialloc(dev)
{
    register *fp, *bp, *ip;
    int i, j, k, ino;

    fp = getfs(dev);
    while(fp->s_ilock)
        sleep(&fp->s_ilock, PINOD);
loop:
    if(fp->s_ninode > 0) {
        ino = fp->s_inode[--fp->s_ninode];
        ip = iget(dev, ino);
        if (ip==NULL)
            return(NULL);
        if(ip->i_mode == 0) {
            for(bp = &ip->i_mode; bp < &ip->i_addr[8];)
                *bp++ = 0;
            fp->s_fmod = 1;
            return(ip);
        }
        iput(ip);
        goto loop;
    }
    fp->s_ilock++;
    ino = 0;
```

```
26     for(i=0; i<fp->s_isize; i++) {
27         bp = bread(dev, i+2);
28         ip = bp->b_addr;
29         for(j=0; j<256; j=+16) {
30             ino++;
31             if(ip[j] != 0)
32                 continue;
33             for(k=0; k<NINODE; k++)
34             if(dev==inode[k].i_dev && ino==inode[k].i_number)
35                 goto cont;
36             fp->s_inode[fp->s_ninode++] = ino;
37             if(fp->s_ninode >= 100)
38                 break;
39         cont:;
40         }
41         brelse(bp);
42         if(fp->s_ninode >= 100)
43             break;
44     }
45     fp->s_ilock = 0;
46     wakeup(&fp->s_ilock);
47     if (fp->s_ninode > 0)
48         goto loop;
49     prdev("Out of inodes", dev);
50     u.u_error = ENOSPC;
51     return(NULL);
52 }
```

6 取得与参数的设备编号相对应的 `filsys` 结构体（超级块）。

7~8 进入睡眠状态直至解锁 `filsys` 结构体。

10 当 inode 空闲队列中还存在未分配的 inode 编号时的处理。

11 取得位于 inode 空闲队列（栈）头部的 inode 编号。

12~14 利用所取得的 inode 编号调用 `iget()` 以取得 `inode[]` 元素。

15 取得的 `inode[]` 元素的 `i_mode` 为 0 时的处理（释放 inode 时相应的 `inode[]` 元素的 `i_mode` 被清 0，如果 `i_mode` 不为 0 则说明该 `inode[]` 元素仍处于使用中的状态，或者 inode 的释放处理未能正常执行）。

16~17 将 `inode[]` 元素从 `i_mode` 至 `i_addr` 的部分清 0。

18 将 `filsys.s_fmod` 置 1 以设置 `filsys` 结构体的更新标志位。

19 返回 `inode[]` 元素。

21~22 如果 `inode.i_mode` 的值不为 0，则释放取得的 `inode[]` 元素，然后返回 `loop` 处，并再次尝试取得 inode。

24 如果空闲队列内不存在尚未分配的 inode 编号，则取得 `filsys` 结构体的锁。

26 遍历块设备的 inode 的块（#2~）。

29 遍历块中所有的 inode。

31~32 如果 `inode.i_mode` 不为 0，则表示该 inode 处于使用中的状态，执行 `continue` 。

33~35 如果 `inode[]` 中存在相应元素则跳转至 `cont` 。似乎是在块设备和内存中都确认了该 inode 未

被分配，所以才将其视作未分配 inode。

36　将 inode 编号追加至空闲队列。

37~38　如果空闲队列已满，则执行 `break` 退出循环。

42~43　如果空闲队列已满，则执行 `break` 退出循环，终止补充空闲队列的处理。

45　释放 `filsys` 结构体的锁。

46　唤醒正在等待解锁 `filsys` 结构体的进程。

47~48　如果向空闲队列至少补充一个未分配的 inode 编号，则返回 `loop` 执行分配 inode 的处理。

ifree()

`ifree()` 用于释放 inode（表 9-16，代码清单 9-18 ）。将被释放的 inode 的 inode 编号追加至空闲队列。如果空闲队列已满，则丢弃 inode 编号（图 9-12 ）。丢弃的 inode 编号在通过 `ialloc()` 补充空闲队列时被回收。

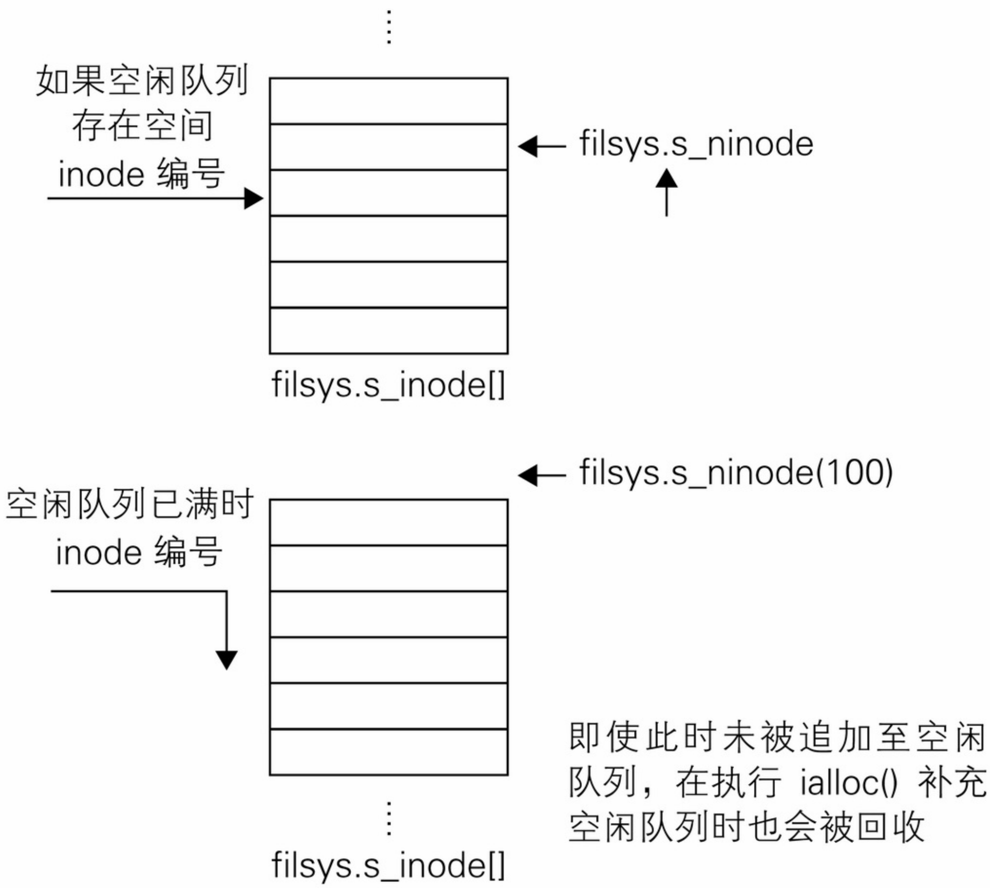


图 **9-12**　**ifree()**

表 **9-16**　**ifree()** 的参数

参数	含义

dev	设备编号
ino	inode 编号

代码清单 **9-18　ifree()**（**ken/alloc.c)**

```
1 ifree(dev, ino)
2 {
3     register *fp;
4
5     fp = getfs(dev);
6     if(fp->s_ilock)
7         return;
8     if(fp->s_ninode >= 100)
9         return;
10    fp->s_inode[fp->s_ninode++] = ino;
11    fp->s_fmod = 1;
12 }
```

5　取得与参数指定的设备编号相对应的 `filsys` 结构体。

6~7　如果 `filsys` 结构体被加锁，则不做任何处理立即返回。尽管此时未能将当前的 inode 编号回收至空闲队列，但在通过 `ialloc()` 补充空闲队列时，一定会进行回收。

8~9　当空闲队列已满时，不做任何处理立即返回。出于与上述同样的原因，此处也不存在回收的问题。

10　将 inode 编号返回至空闲队列。

11　设置 `filsys` 结构体的更新标志位。

alloc()

`alloc()` 是分配块设备存储区域中未使用的块的函数（表 9-17，代码清单 9-19 ）。首先取得 `filesys.s_nfree` 指向的 `filsys.s_free[]` 元素，然后使用该块编号取得块设备的缓冲区。

如果空闲队列已空，则从块设备存储区域取得未使用的块编号，将其补充至空闲队列。此处的补充处理与针对 inode 的补充处理有很大不同。块设备中的每 99 个未使用的块编号用 1 个队列管理，队列头部的元素保存着队列中的元素数和下一个队列的块编号（图 9-13 ）。这个未使用块编号的队列在执行 `/etc/mkfs` 时生成，通过复制该队列补充存储空闲队列（图 9-14 ）。

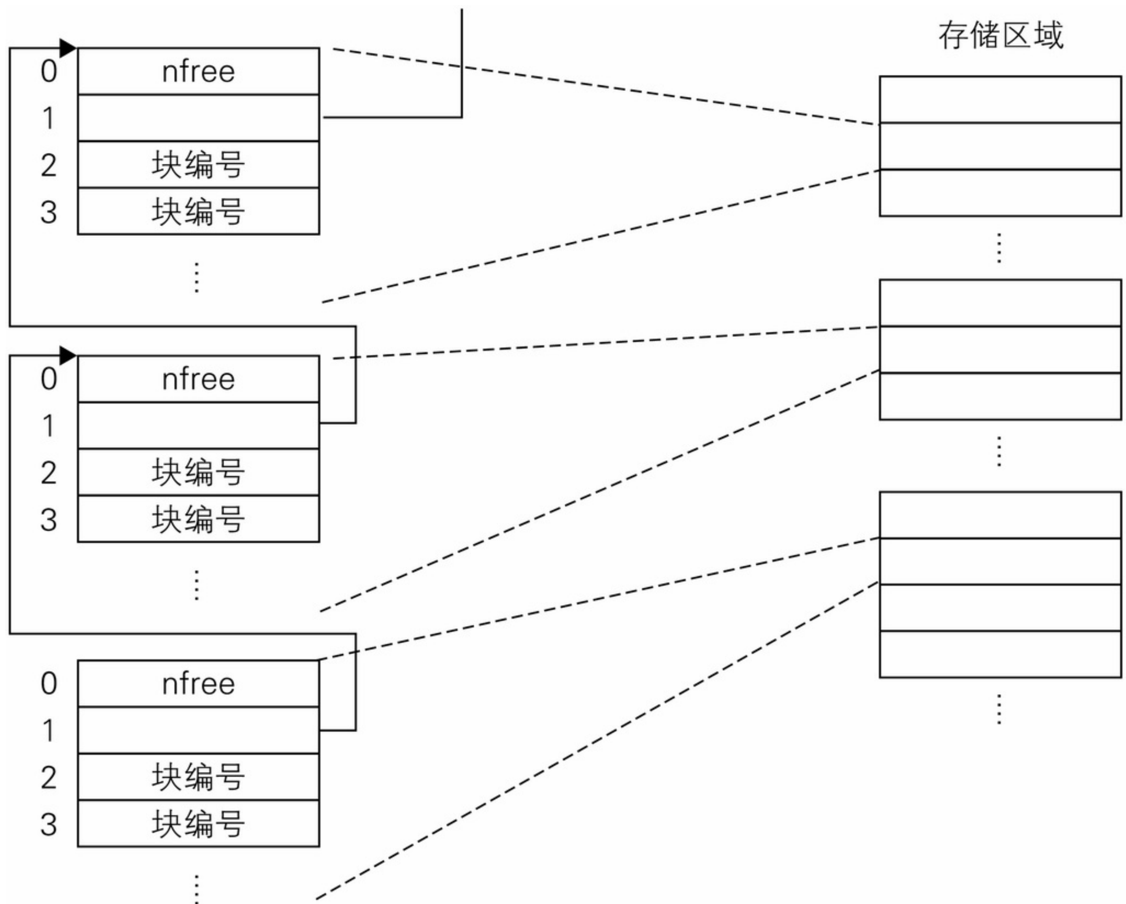


图 **9-13**　未使用块编号的队列

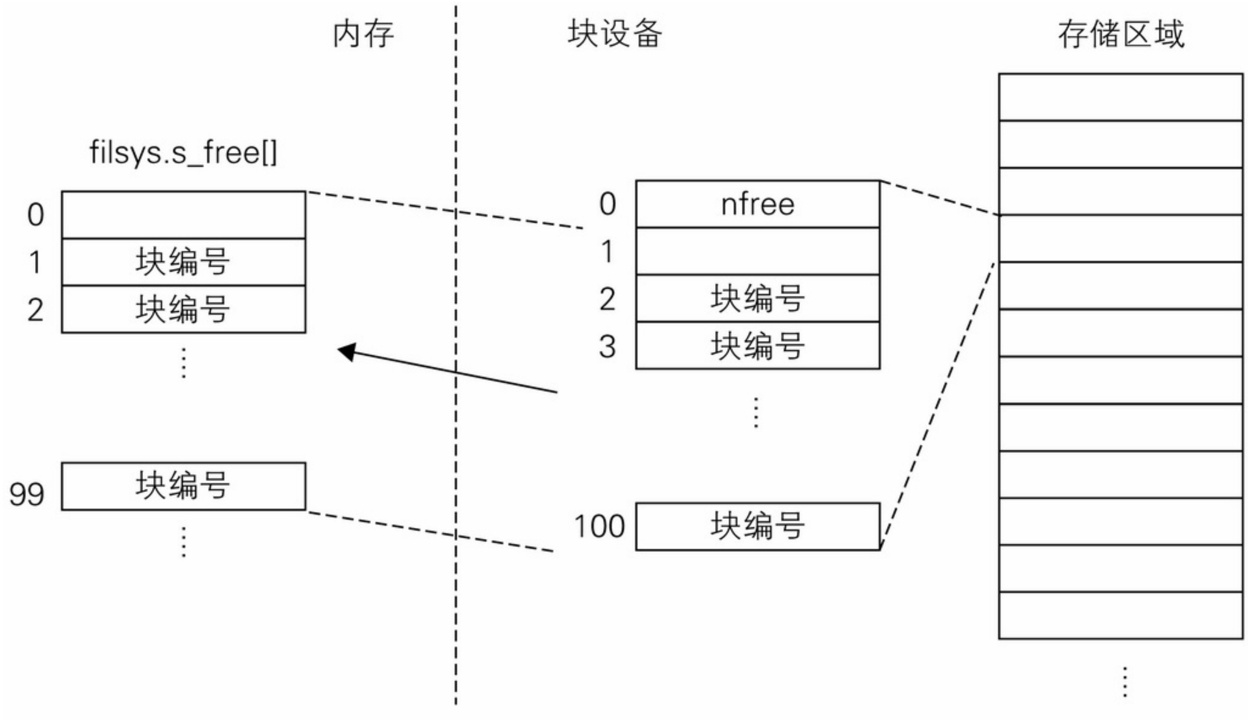


图 **9-14**　补}存储空闲队列

表 **9-17**　**alloc()** 的参数

参数	含义
dev	设备编号

代码清单 **9-19**　**alloc()**（**ken/alloc.c)**

```
alloc(dev)
{
    int bno;
    register *bp, *ip, *fp;

    fp = getfs(dev);
    while(fp->s_flock)
        sleep(&fp->s_flock, PINOD);
    do {
        if(fp->s_nfree <= 0)
            goto nospace;
        bno = fp->s_free[--fp->s_nfree];
        if(bno == 0)
            goto nospace;
    } while (badblock(fp, bno, dev));
    if(fp->s_nfree <= 0) {
        fp->s_flock++;
        bp = bread(dev, bno);
        ip = bp->b_addr;
        fp->s_nfree = *ip++;
        bcopy(ip, fp->s_free, 100);
        brelse(bp);
        fp->s_flock = 0;
        wakeup(&fp->s_flock);
    }
```

```
26     bp = getblk(dev, bno);
27     clrbuf(bp);
28     fp->s_fmod = 1;
29     return(bp);
30
31 nospace:
32     fp->s_nfree = 0;
33     prdev("no space", dev);
34     u.u_error = ENOSPC;
35     return(NULL);
36 }
```

6　取得与参数指定的设备编号相对应的 `filsys` 结构体。

7~8　如果 `filsys` 结构体被加锁，则进入睡眠状态直至解锁。

9　进行循环直至取得合适的块编号。如果取得的块编号既未指向块设备的 inode 区域，也未指向存储区域，`badblock()` 将返回 1。[4]

10~11　如果空闲队列为空则跳转至 `nospace` 。

12~14　从空闲队列取得块编号。如果为 0 则跳转至 `nospace` 。

16　如果取得了合适的块编号，且空闲队列变为空时，对其进行补充处理。

17　对 `filsys` 结构体加锁。

18　取得的块编号指向容纳着下一个未使用块编号队列的块，执行 `bread()` 读取该块。

20　在读取的块的头部容纳着该队列中块编号的数量，将其复制到 `filsys.s_nfree` 。

21　执行 `bcopy()` ，将队列整体拷贝到 `filsys.s_free` 。

22　释放缓冲区。

23　将 `filsys` 结构体解锁。

24　唤醒正在等待解锁 `filsys` 结构体的进程。

26　利用取得的块编号，执行 `getblk()` ，取得对应的缓冲区。

27　将取得的缓冲区清 0。

28　设置 `filsys` 结构体的更新标志位。

> [4] 这里的原文有误。只有当块编号未指向存储区域时，`badblock()` 才返回 1。——译者注

free()

`free()` 用于释放存储区域的块（表 9-18，代码清单 9-20 ），并将块编号追加至空闲队列。当空闲队列已满时，将空闲队列的内容写入准备释放的块。写入的内容为，以当前的 `filsys.s_nfree` 为起始元素，其后跟随 `filsys.s_free[]` 元素。随后将 `filsys.s_nfree` 清 0，并将 `filsys.s_free[0]` 设定为准备释放的块的块编号（图 9-15）。

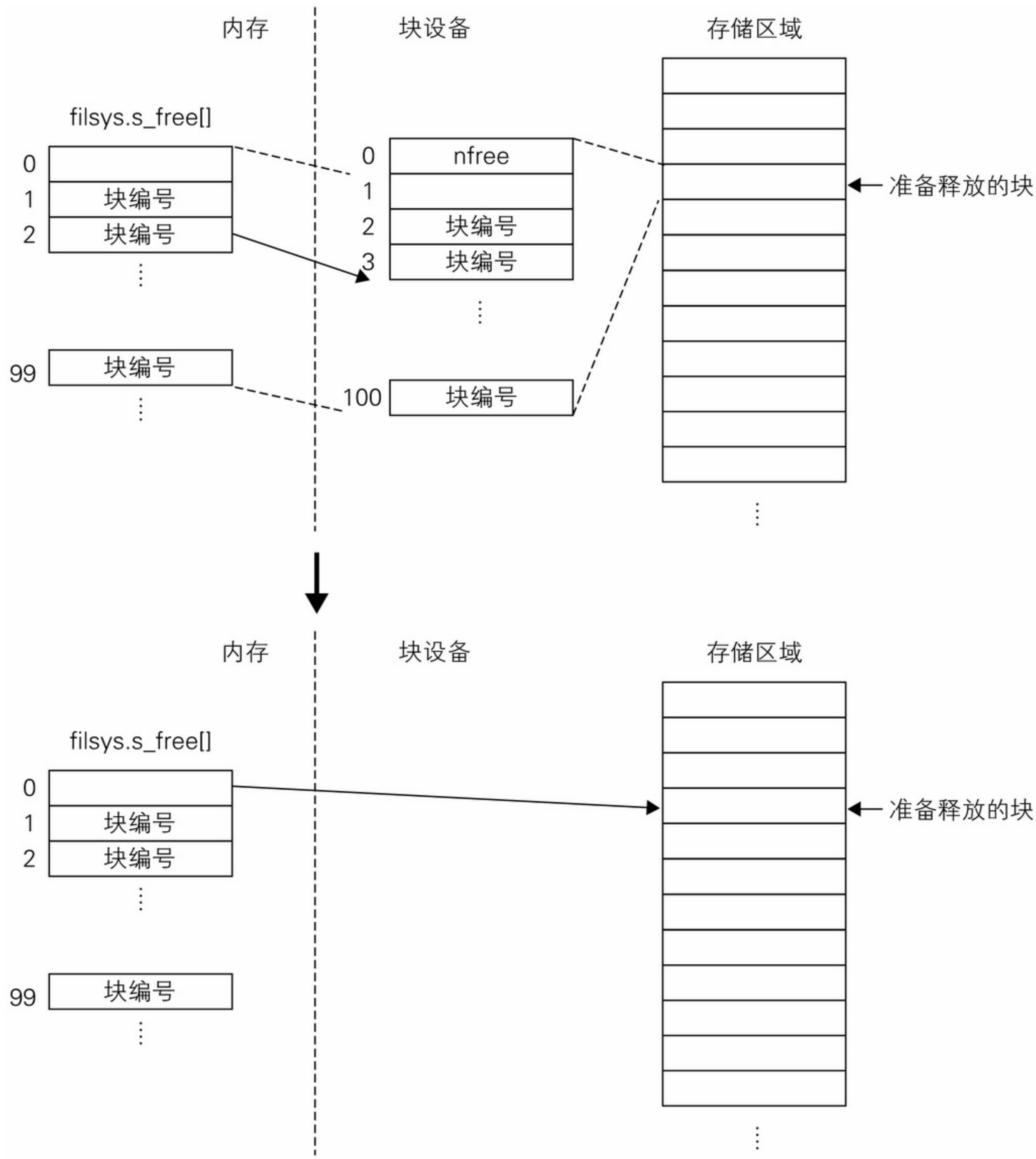


图 **9-15　free()**

表 **9-18　free()** 的参数

参数	含义
dev	设备编号

bno	块编号

代码清单 **9-20　free()**（**ken/alloc.c)**

```
 1 free(dev, bno)
 2 {
 3     register *fp, *bp, *ip;
 4
 5     fp = getfs(dev);
 6     fp->s_fmod = 1;
 7     while(fp->s_flock)
 8         sleep(&fp->s_flock, PINOD);
 9     if (badblock(fp, bno, dev))
10         return;
11     if(fp->s_nfree <= 0) {
12         fp->s_nfree = 1;
13         fp->s_free[0] = 0;
14     }
15     if(fp->s_nfree >= 100) {
16         fp->s_flock++;
17         bp = getblk(dev, bno);
18         ip = bp->b_addr;
19         *ip++ = fp->s_nfree;
20         bcopy(fp->s_free, ip, 100);
21         fp->s_nfree = 0;
22         bwrite(bp);
23         fp->s_flock = 0;
24         wakeup(&fp->s_flock);
25     }
26     fp->s_free[fp->s_nfree++] = bno;
27     fp->s_fmod = 1;
28 }
```

5　取得与参数指定的设备编号相对应的 `filsys` 结构体。

6　设置 `filsys` 结构体的更新标志位。

7~8　进入睡眠状态直至解锁 `filsys` 结构体。

9~10　如果参数的块编号的值有误则返回。

11~14　如果 `filsys.s_nfree` 的值小于等于 0，则将其设为 1，并将 `filsys.s_free[0]` 设为 0。

15　如果空闲队列已满，则将空闲队列写入块设备，并重置空闲队列。

16　将 `filsys` 结构体加锁。

17　取得与参数的块编号相对应的缓冲区。

19　将 `filsys.s_nfree` 保存至缓冲区的起始位置。

20　将空闲队列整体复制到缓冲区。

21　将 `filsys.s_nfree` 设定为 0（将空闲队列清空）。

22　将保存空闲队列的缓冲区写入块设备。

23　将 `filsys` 结构体解锁。

24　唤醒正在等待解锁 `filsys` 结构体被进程。

26　向空闲队列追加块编号。如果第 15 行的值为真，空闲队列的头部将被设定为下一个队列的块编号。

27　设置 `filsys` 结构体的更新标志位。

getfs()

`getfs()` 用来取得与设备编号相对应的 `filsys` 结构体（超级块）（表 9-19，代码清单 9-21 ）。该函数在 `mount[]` 中寻找与参数指定的设备编号相对应的元素。

表 **9-19**　**getfs()** 的参数

参数	含义
dev	设备编号

代码清单 **9-21**　**getfs()**（**ken/alloc.c)**

```
 1 getfs(dev)
 2 {
 3     register struct mount *p;
 4     register char *n1、*n2;
 5
 6     for(p = &mount[0]; p < &mount[NMOUNT]; p++)
 7     if(p->m_bufp != NULL && p->m_dev == dev) {
 8         p = p->m_bufp->b_addr;
 9         n1 = p->s_nfree;
10         n2 = p->s_ninode;
11         if(n1 > 100 || n2 > 100) {
12             prdev("bad count", dev);
13             p->s_nfree = 0;
14             p->s_ninode = 0;
15         }
16         return(p);
17     }
18     panic("no fs");
19 }
```

9~15　如果 `filsys.s_nfree` 或 `filsys.n_ninode` 的值大于 100，可认为是错误值，此时返回 0。

badblock()

`badblock()` 用于检查块编号是否合适（表 9-20，代码清单 9-22 ）。如果块编号指向块设备中的 inode 区域或存储区域，则返回 0，否则返回 1。[5]

> [5] 这里的原文有误。只有当块编号指向存储区域时 `badblock()` 才返回 0，否则将返回 1。——译者注

表 **9-20**　**badblock()** 的参数

参数	含义
afp	超级块（filsys 结构体）

abn	块编号
dev	设备编号

代码清单 **9.22　badblock()**（**ken/alloc.c)**

```
badblock(afp, abn, dev)
{
    register struct filsys *fp;
    register char *bn;

    fp = afp;
    bn = abn;
    if (bn < fp->s_isize+2 || bn >= fp->s_fsize) {
        prdev("bad block", dev);
        return(1);
    }
    return(0);
}
```

9.7　将路径变为 inode

目录的内容

文件和目录可通过路径访问。目录拥有文件名和 inode 编号的对应表。该表中一条记录的长度为 16 字节，头部的 2 字节是与文件相对应的 inode 编号，其后的 14 字节为文件或目录名（表 9-21 ）。请注意文件或 inode 本身并没有文件名。

表 **9-21**　目录内容例

inode 编号（**2** 字节**)**	文件或目录名（**14** 字节**)**
0x0010	.（表示父目录）
0x0044	..（表示当前目录）
0x0060	fuga
0x0000（0 表示无效记录）	hoge
0x0092	homu

以根目录或当前目录为起点，由路径的起始位置开始逐个取得与路径的各个要素相对应的 `inode[]` 元素。如果元素为目录，则取得与该目录相对应的 `inode[]`，然后再重复上述处理，这样就可以取得与目标文件或目录相对应的全部 `inode[]`（图 9-16 ）。

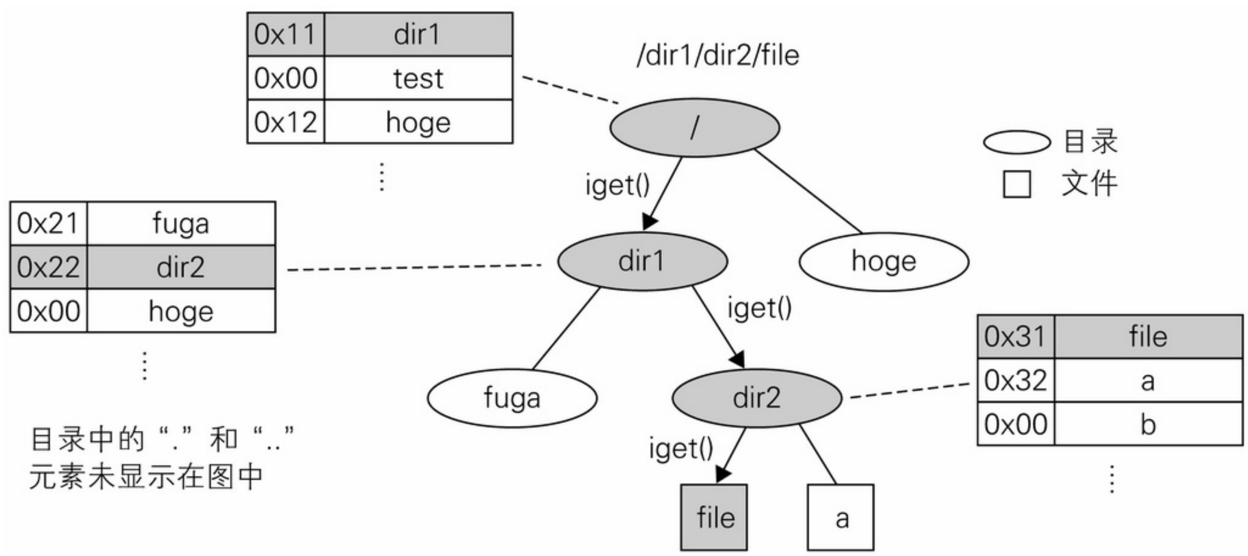


图 **9-16**　遍历文件路径

namei()

namei() 遍历文件路径，取得与文件路径名相对应的文件或目录的 inode[] 元素（表 9-22，代码清单 9-24 ）。

如果给出的路径名的起始字符为‘/’，可判断该路径为绝对路径，并将根目录作为遍历的起点。否则可判断为相对路径，并将当前路径作为遍历的起点。

从路径名的起始位置开始逐个取得构成路径的各个元素，然后遍历各个目录对应表所包含的记录，再使用 iget() 取得与各记录相对应的 inode[] 元素，并重复上述处理。

遍历目录的同时检查路径是否恰当和访问权限。如果对某个目录不具备执行权限，是无法访问该目录中的数据的。

namei() 的参数中的 flag 代表准备对路径表示的文件进行的操作，可指定的操作包括寻找、生成或是删除。根据指定的操作，会相应的更新 user 结构体的内容。

- u.u_pdir：只有当准备生成新的文件或目录时，才会设定为与对象文件的父目录相对应的 inode[] 元素。如果准备生成名为“/home/hoge/fuga”的文件，则将 u.u_pdir 设定为与“/home/hoge”相对应的 inode[] 元素
- u.u_offset：只有当准备生成新的文件或目录时，才会设定为指向对象文件父目录中空记录的偏移量
- u.u_dbuf：文件名或目录名。如果路径名为“/home/hoge/fuga”，则将 u.u_dbuf 设定为“fuga”

很多函数都是使用上述更新后的数据进行自身处理的，因此在执行这些函数之前，需要首先执行 namei() 。

namei() 通过执行 schar() 和 uchar() 取得路径名（代码清单 9-23 ）。schar() 和 uchar() 的区别在于路径名是保存在用户空间还是内核空间。

一般来说，用户程序通过系统调用对文件进行操作。此时在 trap() 中，将 u.u_dirp 设定为 u.u_arg[0] 。u.u_arg[0] 是赋予系统调用的参数，被设定为路径名的地址。uchar() 通过一边对 u.u_dirp 进行位移操作，一边执行 fubyte()，在用户空间逐个处理构成数据的字符。

当内核程序希望独自操作文件时，也采取将 u.u_dirp 设定为路径名地址的方式。此时不会发生模式的切换，而是通过递增 u.u_dirp 对字符进行逐个处理。

代码清单 **9-23**　**schar()** 和 **uchar()**

```
schar()
    return(*u.u_dirp++ & 0377);
}

uchar()
{
    register c;

    c = fubyte(u.u_dirp++);
    if(c == -1)
        u.u_error = EFAULT;
    return(c);
}
```

表 **9-22**　**namei()** 的参数

参数	含义
func	&uchar 或 &schar 。区别在于路径名是存在于用户空间还是内核空间
flag	0：寻找路径名所示的 inode。1：生成路径名所示的 inode。2：删除路径名所示 的 inode

代码清单 **9-24**　**namei()** 和 **ken/namei.c**

```
namei(func, flag)
int (*func)();
{
    register struct inode *dp;
    register c;
    register char *cp;
    int eo, *bp;

    dp = u.u_cdir;
    if((c=(*func)()) == '/')
        dp = rootdir;
    iget(dp->i_dev, dp->i_number);
    while(c == '/')
        c = (*func)();
    if(c == '\0' && flag != 0) {
        u.u_error = ENOENT;
        goto out;
    }

cloop:
    if(u.u_error)
        goto out;
    if(c == '\0')
        return(dp);

    if((dp->i_mode&IFMT) != IFDIR) {
        u.u_error = ENOTDIR;
        goto out;
    }
    if(access(dp, IEXEC))
        goto out;
```

```
    cp = &u.u_dbuf[0];
    while(c!='/' && c!='\0' && u.u_error==0) {
        if(cp < &u.u_dbuf[DIRSIZ])
            *cp++ = c;
        c = (*func)();
    }
    while(cp < &u.u_dbuf[DIRSIZ])
        *cp++ = '\0';
    while(c == '/')
        c = (*func)();
    if(u.u_error)
        goto out;

    u.u_offset[1] = 0;
    u.u_offset[0] = 0;
    u.u_segflg = 1;
    eo = 0;
    u.u_count = ldiv(dp->i_size1, DIRSIZ+2);
    bp = NULL;

eloop:

    if(u.u_count == 0) {
        if(bp != NULL)
            brelse(bp);
        if(flag==1 && c=='\0') {
            if(access(dp, IWRITE))
                goto out;
            u.u_pdir = dp;
            if(eo)
                u.u_offset[1] = eo-DIRSIZ-2; else
                dp->i_flag =| IUPD;
            return(NULL);
        }
        u.u_error = ENOENT;
        goto out;
    }

    if((u.u_offset[1]&0777) == 0) {
        if(bp != NULL)
            brelse(bp);
        bp = bread(dp->i_dev,
            bmap(dp, ldiv(u.u_offset[1], 512)));
    }

    bcopy(bp->b_addr+(u.u_offset[1]&0777), &u.u_dent, (DIRSIZ+2)/2);
    u.u_offset[1] =+ DIRSIZ+2;
    u.u_count--;
    if(u.u_dent.u_ino == 0) {
        if(eo == 0)
            eo = u.u_offset[1];
        goto eloop;
    }
    for(cp = &u.u_dbuf[0]; cp < &u.u_dbuf[DIRSIZ]; cp++)
        if(*cp != cp[u.u_dent.u_name - u.u_dbuf])
            goto eloop;

    if(bp != NULL)
        brelse(bp);
    if(flag==2 && c=='\0') {
        if(access(dp, IWRITE))
        goto out;
            return(dp);
    }
    bp = dp->i_dev;
    iput(dp);
    dp = iget(bp, u.u_dent.u_ino);
    if(dp == NULL)
        return(NULL);
    goto cloop;

```

```
104 out:
105     iput(dp);
106     return(NULL);
107 }
```

9~11　rootdir 表示与根目录相对应的 inode[] 元素，在系统启动时设定（代码清单 9-25）。如果路径名以‘/’开始，则将 dp 设定为与根目录相对应的 inode[] 元素，否则将其设定为 u.u_cdir（当前目录）。dp 将作为遍历的起点。

代码清单 **9-25**　**rootdir**（**system.h)**

```
1 int    *rootdir;
```

12　执行 iget() 并等待解锁 inode[] 元素，然后确认相关的文件系统已被挂载，递增参照计数器，再对 inode[] 元素加锁。

13~14　如果路径名中的指针指向‘/’则忽略该字符，移动指针使其指向‘/’的下一个字符，并将该字符赋予 c 。结束循环时，如果路径名为“////home/fuga”则 c 的值为‘h’。如果路径名为“foo/var”则 c 的值为‘f’。

15~18　当路径名为‘’（空字符）、‘/’或‘///’时，此处 if 语句中的 c == ‘\0’ 的值为真。根目录是无法生成或删除的。

20　以 cloop 为起点的代码从 u.u_dirp 指向的元素中取出一个元素，之后将对其进行各种处理。此时 dp 指向与最后处理的要素相对应的 inode[] 元素。

21~22　如果发生错误则跳转至 out 。u.u_error 中可能容纳由 iget() 等生成的错误代码。

23~24　如果已到路径名的末尾，则返回 dp 。假设路径名为“/home/hoge/fuga”时，此处 dp 的值为与“fuga”对应的 inode[] 元素。

26~29　如果 dp 不为目录则进行错误处理。假设路径名为“/home/hoge/fuga”时，如果“hoge”不为目录将引发错误。

30~31　如果对 dp 不具备执行权限则进行错误处理。假设路径名为“/home/hoge/fuga”时，如果对目录“hoge”不具备执行权限将引发错误。

33~40　u.u_dbuf 容纳着 dp 指向的元素的下一个元素名。假设路径名为“/home/hoge/fuga”，且 dp 指向与“home”相对应的 inode[] 元素时，u.u_dbuf 的值为“hoge”。由于 u.u_dbuf 只能容纳 DIRSIZ（=14。代码清单 9-26）个字符，之后的字符将被忽略。如果不到 14 个字符，那么剩余的字符将用 NULL（0）填充。

代码清单 **9-26**　**DIRSIZ**（**param.h)**

```
1 #define    DIRSIZ    14
```

41~42　忽略重复的‘/’字符。

43~44　如果设定了 u.u_error ，则进行错误处理。u.u_error 中可能容纳由 uchar() 等生成的错误

代码。

假设路径名为“/home/hoge/fuga”，且 dp 指向与“home”相对应的 inode[] 元素时，此时 u.u_dbuf 的值为“hoge\0\0\0...”，而 u.u_dirp 指向“fuga”头部的‘f’。

46~51　由于 dp 指向与目录相对应的 inode[] 元素，此后将从 dp 代表的目录的对应表中寻找与 u.u_dbuf 容纳的元素名相对应的记录，因此在此处首先进行初始化设定。u.u_count 表示目录对应表中的记录数。因为目录的文件长度等于记录数 ×16（字节），所以将文件长度除以 16（DIRSIZ +2）即可得到记录数。

53　此后为检查目录对应表中一条记录的处理。

55　当检查完所有记录也未找到与 u.u_dbuf 容纳的元素名相对应的记录时的处理。每处理一条记录后将递减 u.u_count （第 80 行）。

56~57　如果 bp 有块设备缓冲区，则释放该缓冲区。第 74 行的 bp 有可能被赋予块设备缓冲区。

58　如果 flag 为 1（试图生成文件或目录时），且 u.u_dirp 指向路径末尾时的处理。假设路径名为“/home/hoge/fuga”，且 dp 指向与 hoge 相对应的 inode[] 元素。u.u_dbuf 的值为“fuga\0\0...”，而 u.u_dirp 指向路径末尾（“fuga”之后）。此时的状态表示试图在目录“/home/hoge”中生成名为“fuga”的文件或目录。

59~60　如果 dp （与目录相对应的 inode[] 元素，在此目录中试图生成新的文件或目录）不具备写入权限将引发错误。

61　将 u.u_pdir 设定为 dp ，此处的 u.u_pdir 将使用在别的函数中。

62~64　如果 eo 的值不为 0，则将 u.u_offset[1] 设为由 eo 减去 16（字节）后的值。因为 eo 指向位于 dp 代表的目录对应表中空记录之后的记录，减去 16（一条记录的长度）后将指向空记录的起始位置。如果 eo 的值为 0，即当目录对应表中不存在空记录时，设置 dp 的 inode 更新标志位。

67~68　如果在目录的记录中未找到对象记录，将引发 ENOENT 错误。

71　当 u.u_offset[1] 指向块（512 字节）的边界时的处理。请注意，在首次遍历某个目录的记录时一定会执行此处理。

72~73　如果 bp 已持有块设备缓冲区，则释放。

74~75　读取下一个块。

78　从块设备缓冲区向 u.u_dent 复制目录对应表中的一条记录。u.u_dent.u_ino 表示 inode 编号，u.u_dent.u_name 表示文件或目录名。

79　将 u.u_offset[1] 与 16（一条记录的长度）相加。每处理一条记录，就将 u.u_offset[1] 的值增加 16。

80　递减 u.u_count 。每处理一条记录，就将 u.u_count 的值减去 1。

81　u.u_dent.u_ino 为 0（即空记录）时的处理。

82~84　如果 eo 的值为 0，则将 eo 设定为 u.u_offset[1] ，使 eo 指向当前目录对应表中空记录之后的一条记录。返回至 eloop 检查下一条记录。

86~88　将 u.u_dbuf 中保存的字符串和 u.u_dent.u_name 中的字符串进行比较，如果不一致则返回至 eloop 检查下一条记录。u.u_dent.u_name - u.u_dbuf 用来计算 u.u_dent.u_name 和

u.u_dbuf 地址之差，加上 *cp （指向 u.u_dbuf 中的第 x 个字符）后即可指向 u.u_dent.u_name 中的第 x 个字符。如果字符串一致则表示在 dp 代表的目录中找到了与 u.u_dbuf 相对应的记录。

90~91　如果 bp 已有块设备缓冲区，则将其释放。

92~96　如果准备删除文件或目录，且已到达路径名的末尾，则检查是否具有对该文件或目录的父目录的写入权限，如果具有权限则返回 dp 。

假设路径名为“/home/hoge/fuga”，且准备删除“fuga”时，确认“hoge”目录下存在名为“fuga”的文件或目录。如果对“hoge”具有写入权限则返回与“hoge”相对应的 inode[] 元素。调用 namei() 的函数，通过清除父目录对应表中相应记录的 inode 编号，达到删除文件或目录的目的。

97~101　释放 dp 指向的 inode[] 元素。将在目录对应表中找到的记录相对应的 inode[] 元素赋予 dp 。

102　返回至 cloop 。

access()

access() 用来检查文件的权限设定（表 9-23，代码清单 9-27 ）。将由参数指定的 mode （读取、写入、执行）与 inode[] 元素的 i_mode 的低位 9 比特进行比较，确认是否具有相应的权限。inode.i_mode 的低位 9 比特表示文件的权限设定，从高位开始以 3 比特为 1 组，分别表示文件拥有者、组用户以及其他用户具有的权限。

如果是写入，那么不能将文件系统本身设置为只读状态。此外，作为代码段被使用的文件也不能写入。

超级用户（user.u_uid 为 0 ）一定会有权限。但是如果没有为文件拥有者、组用户以及其他用户的其中之一设定执行权限，那么即使是超级用户，也无法执行该文件。

如果具有权限则返回 0，否则返回 1。此外，权限不足时 u.u_error 被设置为错误代码。

表 **9-23**　**access()** 的参数

参数	含义
aip	inode[] 元素
mode	处理种类：读取、写入或执行

代码清单 **9-27**　**access()**（**ken/fio.c)**

```
access(aip, mode)
int *aip;
{
    register *ip, m;

    ip = aip;
    m = mode;
    if(m == IWRITE) {
        if(getfs(ip->i_dev)->s_ronly != 0) {
            u.u_error = EROFS;
            return(1);
        }
        if(ip->i_flag & ITEXT) {
            u.u_error = ETXTBSY;
            return(1);
```

```
16         }
17     }
18     if(u.u_uid == 0) {
19         if(m == IEXEC && (ip->i_mode &
20             (IEXEC | (IEXEC>>3) | (IEXEC>>6))) == 0)
21                 goto bad;
22         return(0);
23     }
24     if(u.u_uid != ip->i_uid) {
25         m =>> 3;
26         if(u.u_gid != ip->i_gid)
27             m =>> 3;
28     }
29     if((ip->i_mode&m) != 0)
30         return(0);
31
32 bad:
33     u.u_error = EACCES;
34     return(1);
35 }
```

8~17　检查是否具有写入权限时的处理。利用 getfs() 取得超级块，检查是否为只读状态，同时也确认该文件不是作为代码段的文件。

18~23　超级用户的处理。

24~30　根据权限检查对象的 inode[] 元素与当前用户的所属关系（文件拥有者、组用户或其他用户），调整参数 mode 的比特位置。对 inode[] 元素的 i_mode 与参数 mode 进行与（&）运算，如果结果不为 0 则可确定用户具有访问权限。

9.8　初始化与同步

iinit()

iinit() 读取根磁盘的超级块，并将其赋予 mount[] 的第一个元素（代码清单 9-28 ）。该函数在系统启动时被 main() 调用，且仅调用一次。

代码清单 **9-28**　**iinit()**（**ken/alloc.c)**

```
iinit()
{
    register *cp, *bp;

    (*bdevsw[rootdev.d_major].d_open)(rootdev, 1);
    bp = bread(rootdev, 1);
    cp = getblk(NODEV);
    if(u.u_error)
        panic("iinit");
    bcopy(bp->b_addr, cp->b_addr, 256);
    brelse(bp);
    mount[0].m_bufp = cp;
    mount[0].m_dev = rootdev;
    cp = cp->b_addr;
    cp->s_flock = 0;
    cp->s_ilock = 0;
    cp->s_ronly = 0;
    time[0] = cp->s_time[0];
    time[1] = cp->s_time[1];
}
```

5　打开根磁盘的处理。如果是 RK 磁盘则不做任何处理。

6　读取超级块的内容。

7~11　取得 `NODEV` 块设备的缓冲区，将超级块的内容复制到此缓冲区，并释放用来读取超级块的缓冲区。

12~13　将根磁盘注册到 `mount[0]` 。

14~17　对超级块解锁，并清除只读标记。

18~19　将表示时间的 `time` 复制到超级块的 `filsys.s_time` 。

update()

`update()` 用来同步内存中的数据和块设备中的数据（代码清单 9-29 ）。该函数将尚未写入块设备的 `mount[]` 、`inode[]` 和 `buf[]` 的内容写入块设备。

代码清单 **9-29**　**update()**（**ken/alloc.c)**

```
 1 update()
 2 {
 3     register struct inode *ip;
 4     register struct mount *mp;
 5     register *bp;
 6 
 7     if(updlock)
 8         return;
 9     updlock++;
10     for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
11         if(mp->m_bufp != NULL) {
12             ip = mp->m_bufp->b_addr;
13             if(ip->s_fmod==0 || ip->s_ilock!=0 ||
14                ip->s_flock!=0 || ip->s_ronly!=0)
15                 continue;
16             bp = getblk(mp->m_dev, 1);
17             ip->s_fmod = 0;
18             ip->s_time[0] = time[0];
19             ip->s_time[1] = time[1];
20             bcopy(ip, bp->b_addr, 256);
21             bwrite(bp);
22         }
23     for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
24         if((ip->i_flag&ILOCK) == 0) {
25             ip->i_flag =| ILOCK;
26             iupdat(ip, time);
27             prele(ip);
28         }
29     updlock = 0;
30     bflush(NODEV);
31 }
```

7~9　取得 `updlock` 的锁。如果已被加锁，则不做任何处理立即返回。`updlock` 是用于排他处理的变量（代码清单 9-30）。

代码清单 **9-30**　**updlock**（**system.h)**

```
int updlock;
```

10 遍历 `mount[]`。

11~15 如果未设置元素的更新标志位，或已被加锁，或处于只读状态，则对该元素不做任何处理。

16 读取超级块的内容。

17 清除更新标志位。

18~19 更新超级块的 `filsys.s_time`。

20~21 将超级块的内容复制到缓冲区，并将缓冲区的内容写入块设备。

23~28 遍历 `inode[]`。如果元素未被加锁，则加锁后调用 `iupdat()`，将 `inode[]` 元素的内容写入块设备。

29 释放 `updlock` 上的锁。

30 调用 `bflush()` 刷新块设备的缓冲区。因为参数为 `NODEV`，所以会刷新所有设备的块设备缓冲区。

9.9 小结

- 块设备的结构通过文件系统得以抽象化。
- 数据通过文件、目录以及树状结构的命名空间进行管理。
- 文件用来表现数据本身，而目录是用来管理下一层文件或目录的容器。
- 目录具有与文件相同的实体。目录持有下一层文件或目录的 inode 编号和文件名。
- 可以对文件或目录设定访问权限。
- 通过挂载或卸载可以导入或除去多个块设备的文件系统。
- 块设备的区域可分为以下 4 种：启动用区域、超级块、inode 区域和存储区域。
- 超级块用来管理块设备的控制信息。
- inode 用来管理文件的定义信息。
- 存储区域的块用来保存数据本身。
- 文件的实体由 1 个 inode 和多个存储区域的块组成。
- 文件使用的存储区域的块可通过 inode 的映射获取。
- 从 inode 到存取区域的映射分为以下 3 种：直接参照、间接参照和双重间接参照。
- 未使用 inode 和存储区域的未使用块由超级块的空闲队列管理。
- 通过重复下述处理可以从路径名获取相应文件（的 inode）：首先取得路径名中的一个元素，然后取得目录对应表相应记录中的 inode，再取得路径名中的下一个元素。

第 **10** 章　文件处理

10.1　用户程序对文件的处理

用户程序处理文件时，流程如下所示。

1. 使用路径名打开准备进行处理的文件，如果成功，则可以获取文件描述符。

2. 使用文件描述符读写文件，或调整文件的偏移量（读写文件的位置）。

3. 处理完毕后关闭文件。

如果用 C 语言进行描述，上述流程可以表现为代码清单 10-1 的代码。

代码清单 **10-1**　文件处理的示例代码

```
 1 main() {
 2     int fd;
 3     char buf[100];
 4     fd = open("filename.txt", 2);
 5     read(fd, buf, 10);
 6     buf[0] = "a";
 7     seek(fd, 0, 0);
 8     write(fd, buf, 10);
 9     close(fd);
10 }
```

4　将文件路径作为参数，以读写模式打开，如果成功则返回文件描述符。

5　将位于文件头部的 10 字节数据读入 `buf` 。

6　更新读取的数据。

7　将文件偏移量移动至文件起始位置。

8　将更新后的数据写入文件。

9　关闭文件，结束对文件的处理。

`open()` 、`read()` 、`seek()` 、`write()` 、`close()` 是由 C 语言库提供的操作文件的函数。这些函数通过执行系统调用处理文件。

10.2　**3** 个结构体

文件处理通过 3 个结构体管理。`user.u_ofile[]` 负责管理由进程打开的文件，`file` 结构体的数组 `file[]` 用来管理已被打开的文件的处理信息（代码清单 10-2，表 10-1 ）。`inode[]` 表示读取到内存的 inode 信息。

因为每一个 `file[]` 元素都各自持有文件偏移量和文件访问模式，所以即使有两个进程同时打开同一文件，且其中一个进程将文件偏移量向前移动，也不会对由另一个进程打开的文件的文件偏移量产生影响。

当进程打开某个文件之后，该进程被赋予 `user.u_ofile[]` 和 `file[]` 的元素，且 `user.u_ofile[]` 元素指向 `file[]` 元素，而 `file[]` 元素指向 `inode[]` 元素（图 10-1 ）。`user.u_ofile[]` 的数组下标将返还

给用户程序，该下标被称为文件描述符。用户程序通过文件描述符对文件进行操作。

关闭文件时，与打开文件时相反，将释放 `user.u_ofile[]`、`file[]` 和 `inode[]` 元素。

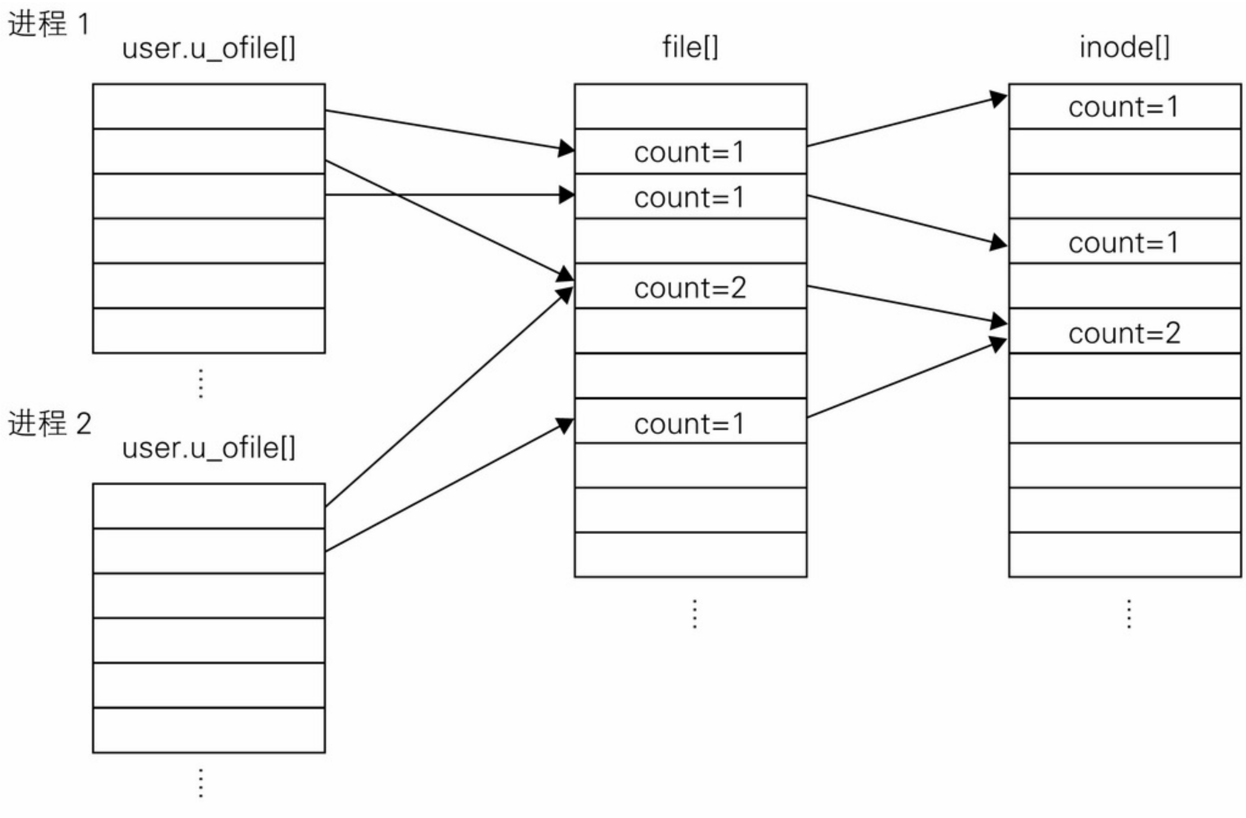


图 **10-1**　打开文件

代码清单 **10-2**　**file (file.h)**

```
struct    file
{
    char    f_flag;
    char    f_count;
    int     f_inode;
    char    *f_offset[2];
} file[NFILE];

/* flags */
#define    FREAD     01
#define    FWRITE    02
#define    FPIPE     04
```

表 **10-1**　**file** 结构体

成员	含义
`f_flag`	读取、写入、管道。参见表 9-2

f_count	参照计数器
f_inode	相对应的 inode[] 元素
*f_offset[]	文件偏移量

表 **10-2**　**file** 的标志位

标志位	含义
FREAD	以读取模式打开。不具备文件的读取权限时无法以此模式打开文件
FWRITE	以写入模式打开。不具备文件的写入权限时无法以此模式打开文件
FPIPE	作为管道使用（参见第 11 章）

标准输入输出

当系统启动和用户登录时，执行打开终端的处理。user.u_ofile[] 的第 0 个元素被用作标准输入、第 1 个元素被用作标准输出，而第 2 个元素被用作标准错误输出（图 10-2 ）

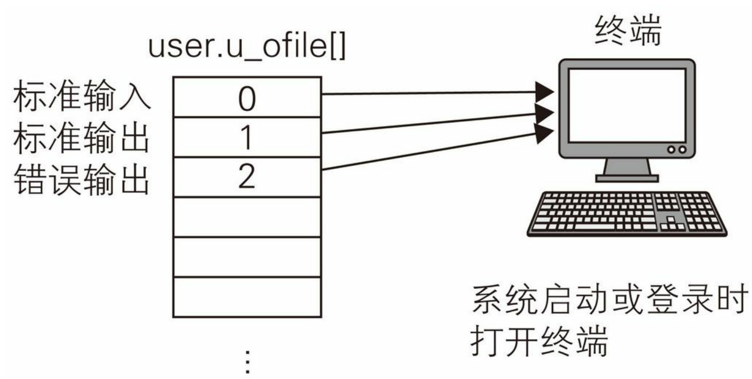


图 **10-2**　标准输入输出

通过更改 user.u_ofile[] 的第 0~2 个元素，可以实现重定向或管道功能。比如关闭 user.u_ofile[1] ，并将其设定为普通文件后，程序向标准输出写入的内容，将被输出至该文件（图 10-3 ）。管道的内容请参照第 11 章。类似这种改变输出对象的处理，一般通过 Shell 程序进行。

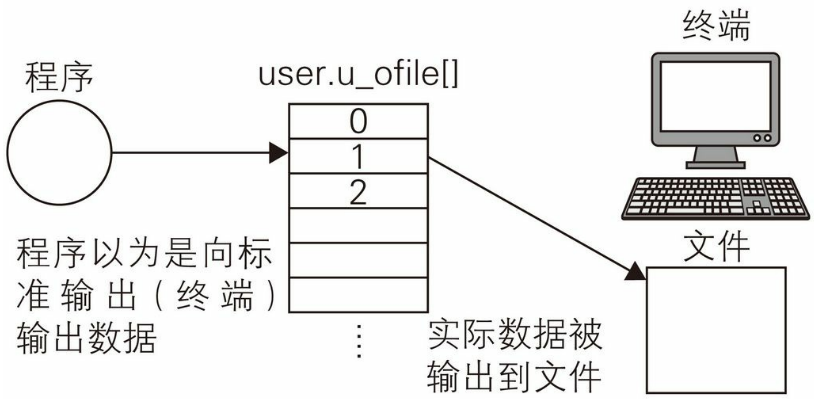


图 **10-3**　重定向

10.3　文件的生成和打开处理

文件的生成和打开基本由共用处理实现。

系统调用 **creat**

系统调用 `creat` 生成新的文件，并将其打开。该文件已存在时，将文件长度设定为 0 并将其打开。`creat()` 为系统调用 `creat` 的处理函数（表 10-3，代码清单 10-3 ）。

表 **10-3**　系统调用 **creat** 的参数

参数	含义
`u.u_arg[0]`	文件的路径名
`u.u_arg[1]`	文件打开模式。与 inode.i_mode 对应。文件已存在时不使用

代码清单 **10-3**　**creat()**（**ken/sys2.c)**

```
 1 creat()
 2 {
 3     register *ip;
 4     extern uchar;
 5
 6     ip = namei(&uchar, 1);
 7     if(ip == NULL) {
 8         if(u.u_error)
 9             return;
10         ip = maknode(u.u_arg[1]&07777&(~ISVTX));
11         if (ip==NULL)
12             return;
13         open1(ip, FWRITE, 2);
14     } else
15         open1(ip, FWRITE, 1);
16 }
```

6　尝试获取与用户程序设定的路径相对应的 `inode[]` 元素。

7~13　如果通过 namei() 无法取得 inode[] 元素，则尝试生成新的文件。将 maknode() 的参数设定为用户程序指定的模式。但是，由于清除了模式的高位比特（010000 以上），因此无法生成目录（IFDIR=040000），也无法设定 Sticky bit。

生成文件后执行 open1() 。open1() 是生成并打开文件的共用函数，第 3 个参数的 2 表示正在生成新的文件。

14~15　如果通过 namei() 成功取得 inode[] 元素，则使用该元素。open1() 的第 3 个参数的 1 表示正在对已存在的文件进行初始化处理。

maknode()

maknode() 用来生成新的文件（表 10-4，代码清单 10-4 ）。首先执行 ialloc() 从空闲队列中获取 inode[] 元素，执行 wdir() 向目录的对应表追加记录，然后将 inode[] 元素返还给调用者。

表 **10-4**　**maknode()** 的参数

参数	含义
mode	准备生成的文件的模式，对应 inode.i_mode

代码清单 **10-4**　**maknode()**（**ken/iget.c)**

```
 1 maknode(mode)
 2 {
 3     register *ip;
 4
 5     ip = ialloc(u.u_pdir->i_dev);
 6     if (ip==NULL)
 7         return(NULL);
 8     ip->i_flag =| IACC|IUPD;
 9     ip->i_mode = mode|IALLOC;
10     ip->i_nlink = 1;
11     ip->i_uid = u.u_uid;
12     ip->i_gid = u.u_gid;
13     wdir(ip);
14     return(ip);
15 }
```

5~7　u.u_pdir 指向准备在其中生成新文件的目录。u.u_pdir 由在 maknode() 之前执行的 namei() 设定。

wdir()

wdir() 向目录的对应表中追加新的记录（表 10-5，代码清单 10-5）。通过 namei() ，将追加对象的父目录、追加对象的文件或目录名，以及父目录对应表中的偏移量设置在 user 结构体中（图 10-4 ）。

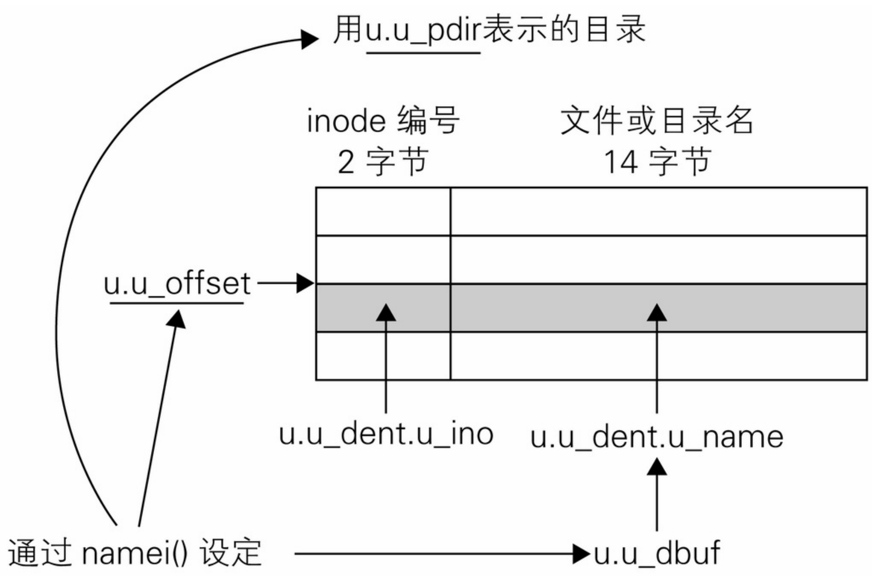


图 **10-4　wdir()**

由于目录的实体是文件，因此采取了与文件相同的方法调用 `writei()` 更新目录数据。

表 **10-5　wdir()** 的参数

参数	含义
ip	inode[] 元素

代码清单 **10-5　wdir()**（**ken/iget.c)**

```
 1 wdir(ip)
 2 int *ip;
 3 {
 4     register char *cp1, *cp2;
 5 
 6     u.u_dent.u_ino = ip->i_number;
 7     cp1 = &u.u_dent.u_name[0];
 8     for(cp2 = &u.u_dbuf[0]; cp2 < &u.u_dbuf[DIRSIZ];)
 9         *cp1++ = *cp2++;
10     u.u_count = DIRSIZ+2;
11     u.u_segflg = 1;
12     u.u_base = &u.u_dent;
13     writei(u.u_pdir);
14     iput(u.u_pdir);
15 }
```

14　对于在其中追加文件或目录的目录，`namei()` 递增了它的参照计数器，此处则递减该计数器。

系统调用 **open**

系统调用 open 用于打开文件。`open()` 是系统调用 open 的处理函数（表 10-6，代码清单 10-6 ）。

表 **10-6**　系统调用 **open** 的参数

参数	含义
u.u_arg[0]	文件路径
u.u_arg[1]	打开文件时的模式。如果不具备该模式的权限则无法打开文件

代码清单 **10-6　open()**（**ken/sys2.c)**

```
 1 open()
 2 {
 3     register *ip;
 4     extern uchar;
 5
 6     ip = namei(&uchar, 0);
 7     if(ip == NULL)
 8         return;
 9     u.u_arg[1]++;
10     open1(ip, u.u_arg[1], 0);
11 }
```

6~8　尝试取得与用户程序指定的路径名相对应的 `inode[]` 元素。

9　系统调用 open 的参数 mode ，它的可选值如下：0 为读取，1 为写入。对 `file` 结构体的标志变量而言，则是 1 为读取，2 为写入，因此需要在此处加 1 进行调整。

10　执行 `open1()` ，进行生成及打开文件的共用处理。第 3 个参数的 0 表示试图打开已存在的文件。

open1()

`open1()` 用来执行生成及打开文件的共用处理（表 10-7，代码清单 10-7 ）。

试图打开已存在的文件时执行 `access()` ，以检查是否具有访问权限。生成新文件时，执行 `itrunc()` 将文件的长度设置为 0。

随后，执行 `falloc()` 从 `user.u_ofile[]` 和 `file[]` 中取得新的元素，将 `file[]` 元素指向与准备打开的文件相对应的 `inode[]` 元素。

表 **10-7　open1()** 的参数

参数	含义
ip	inode[] 元素
mode	表示是准备读取文件，还是写入文件
trf	0：准备打开已存在的文件。1：准备生成新的文件，且该文件已经存在。2：准备生成新的文件，但该文件不存在

代码清单 **10-7　open1()**（**ken/sys2.c)**

```
open1(ip, mode, trf)
```

```
 2 int *ip;
 3 {
 4     register struct file *fp;
 5     register *rip, m;
 6     int i;
 7
 8     rip = ip;
 9     m = mode;
10     if(trf != 2) {
11         if(m&FREAD)
12             access(rip, IREAD);
13         if(m&FWRITE) {
14             access(rip, IWRITE);
15             if((rip->i_mode&IFMT) == IFDIR)
16                 u.u_error = EISDIR;
17         }
18     }
19     if(u.u_error)
20         goto out;
21     if(trf)
22         itrunc(rip);
23     prele(rip);
24     if ((fp = falloc()) == NULL)
25         goto out;
26     fp->f_flag = m&(FREAD|FWRITE);
27     fp->f_inode = rip;
28     i = u.u_ar0[R0];
29     openi(rip, m&FWRITE);
30     if(u.u_error == 0)
31         return;
32     u.u_ofile[i] = NULL;
33     fp->f_count--;
34
35 out:
36     iput(rip);
37 }
```

10~18 文件已存在时检查相应的访问权限。准备读取文件时检查是否具有读取权限，准备写入文件时则检查是否具有写入权限。但是不允许对目录进行写入。

19~20 没有访问权限时，在 access() 内将错误代码赋予 u.u_error 。

21~22 如果系统调用 creat 处于执行中的状态，则执行 itrunc() 将文件长度设定为 0。

23 将 inode[] 元素解锁。该元素在 ialloc() 或 namei() 执行的 iget() 中被加锁。

24~27 从 user.u_ofile[] 和 file[] 取得新的元素。

28 在 falloc() 执行的 ufalloc() 中，将用户进程的 r0 设定为从 user.u_ofile[] 中取得的元素的数组下标。这个值将被返还给用户进程。如果在此后的处理中发生错误，则需要在第 32 行释放所取得的 user.u_ofile[] 元素，因此，此处将下标暂时保存于 i 中。

29 执行 openi() 。如果是一般文件则不做任何处理。

30~31 如果在 openi() 中未发生错误，则认为处理成功，返回调用者。

32~36 如果在 openi() 中发生错误，则释放从 user.u_ofile[] 中取得的元素，并递减 file[] 元素和 inode[] 元素的参照计数器。

falloc()

`falloc()` 用来分配 `user.u_ofile[]` 和 `file[]` 的元素（代码清单 10-8 ）。首先，调用 `ualloc()` 取得 `user.u_ofile[]` 的元素，然后寻找 `file[]` 的空闲元素。如果发现空闲元素，则将通过 `ualloc()` 取得的 `user.u_ofile[]` 元素指向该 `file[]` 元素。

代码清单 **10-8　falloc()**（**ken/fio.c)**

```
1 falloc()
2 {
3     register struct file *fp;
4     register i;
5
6     if ((i = ufalloc()) < 0)
7         return(NULL);
8     for (fp = &file[0]; fp < &file[NFILE]; fp++)
9         if (fp->f_count==0) {
10            u.u_ofile[i] = fp;
11            fp->f_count++;
12            fp->f_offset[0] = 0;
13            fp->f_offset[1] = 0;
14            return(fp);
15        }
16    printf("no file\n");
17    u.u_error = ENFILE;
18    return(NULL);
19 }
```

6~7　从 `user.u_ofile[]` 中取得空闲元素。`ufalloc()` 返回所取得的 `user.u_ofile[]` 元素的数组下标。

8~15　寻找 `file[]` 的空闲元素（参照计数器的值为 0）。找到后将刚才取得的 `user.u_ofile[]` 元素的值设定为指向该 `file[]` 元素的指针，然后递增 `file[]` 元素的参照计数器，并将文件中偏移量的初始值设定为 0。

16~18　如果在 `file[]` 中未找到空闲元素，则进行错误处理。

ufalloc()

`ufalloc()` 从 `user.u_ofile[]` 中取得新的元素（代码清单 10-9 ）。该函数在 `user.u_ofile[]` 中寻找空闲元素，找到后将该元素的数组下标赋予用户进程的 `r0` ，并将下标作为返回值返还给调用者。

代码清单 **10-9　ufalloc()**（**ken/fio.c)**

```
ufalloc()
{
    register i;

    for (i=0; i<NOFILE; i++)
        if (u.u_ofile[i] == NULL) {
            u.u_ar0[R0] = i;
            return(i);
        }
    u.u_error = EMFILE;
    return(-1);
}
```

openi()

openi() 在对象为特殊文件时执行打开设备的处理（表 10-8，代码清单 10-10 ）。如果对象为一般文件，则不做任何处理。

表 **10-8　openi()** 的参数

参数	含义
ip	inode[] 元素
rw	是否进行写入处理

代码清单 **10-10　openi()**（**ken/fio.c)**

```
 1 openi(ip, rw)
 2 int *ip;
 3 {
 4     register *rip;
 5     register dev, maj;
 6
 7     rip = ip;
 8     dev = rip->i_addr[0];
 9     maj = rip->i_addr[0].d_major;
10     switch(rip->i_mode&IFMT) {
11
12     case IFCHR:
13         if(maj >= nchrdev)
14             goto bad;
15         (*cdevsw[maj].d_open)(dev, rw);
16         break;
17
18     case IFBLK:
19         if(maj >= nblkdev)
20             goto bad;
21         (*bdevsw[maj].d_open)(dev, rw);
22     }
23     return;
24
25 bad:
26     u.u_error = ENXIO;
27 }
```

8~9　如果是特殊文件，inode.i_addr[0] 会被设置为设备编号。为了使用设备驱动表中相应的设备驱动，此处会获取设备的大编号。

10~23　检查 inode.i_mode 。如果是特殊文件则执行打开设备的处理。IFCHR 表示字符设备，IFBLK 表示块设备。

10.4　文件的读取和写入

系统调用**read**、**write**

read() 是系统调用 read 的处理函数（表 10-9，代码清单 10-11 ），该系统调用用于读取文件。write() 是系统调用 write 的处理函数（表 10-10，代码清单 10-12 ），该系统调用用于将数据写入文件。

文件的读取与写入在数据的流动方向上是相反的（内存←磁盘，及磁盘→内存），但是处理本身基本上是相同的（图 10-5 ）。相同的处理部分由 rdwr() 实现。read() 和 write() 只是将 file 结构体的标志位

FREAD 和 FWRITE 作为参数调用 rdwr()。

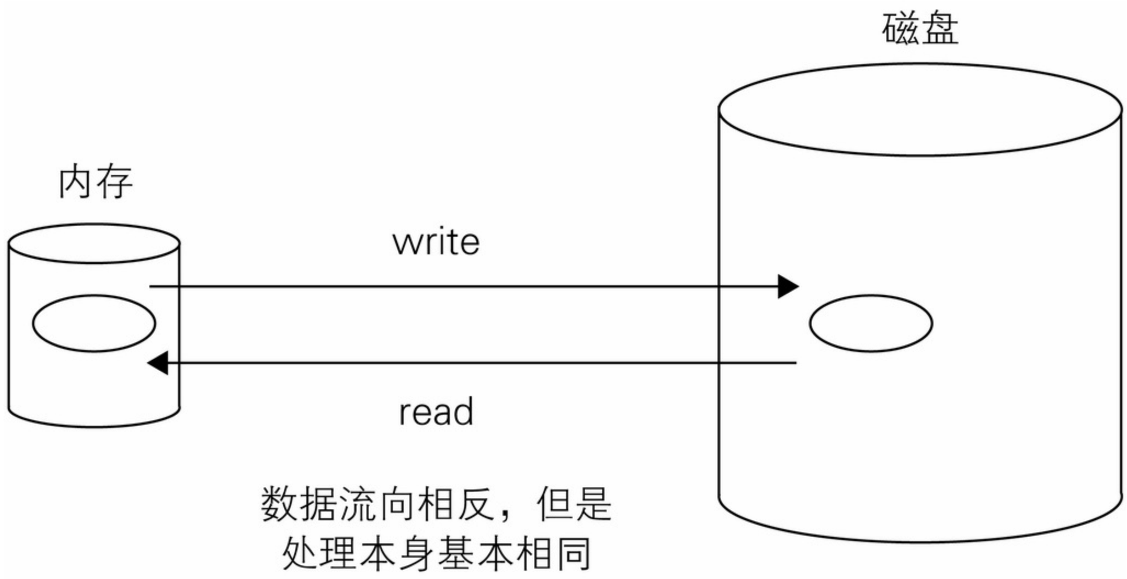


图 **10-5　read** 和 **write**

表 **10-9**　系统调用 **read** 的参数

参数	含义
r0	文件描述符
u.u_arg[0]	存放读取数据的虚拟地址（以字节为单位）
u.u_arg[1]	数据传送量（以字节为单位）

代码清单 **10-11　read()**（**ken/sys2.c**）

```
read()
{
    rdwr(FREAD);
}
```

表 **10-10**　系统调用 **write** 的参数

参数	含义
r0	文件描述符
u.u_arg[0]	存放写入数据的虚拟地址（以字节为单位）

u.u_arg[1]	数据传送量（以字节为单位）

代码清单 **10-12　write()**（**ken/sys2.c**）

```
write()
{
    rdwr(FWRITE);
}
```

rdwr()

rdwr() 用来实现 read() 和 write() 的共用处理（表 10-11，代码清单 10-13）。首先为 u.u_base 、u.u_count 、u.u_offset 和 u.u_segflg 设定适当的值，然后调用 readi() 和 writei() 。

用户进程的 r0 被设定为实际读写的数据的字节数，这个值将作为系统调用的返回值。用户程序可通过返回值来判断读写处理是否正常结束。

表 **10-11　rdwr()** 的参数

参数	含义
mode	设定是读取还是写入

代码清单 **10-13　rdwr()**（**ken/sys2.c**）

```
rdwr(mode)
{
    register *fp, m;

    m = mode;
    fp = getf(u.u_ar0[R0]);
    if(fp == NULL)
        return;
    if((fp->f_flag&m) == 0) {
        u.u_error = EBADF;
        return;
    }
    u.u_base = u.u_arg[0];
    u.u_count = u.u_arg[1];
    u.u_segflg = 0;
    if(fp->f_flag&FPIPE) {
        if(m==FREAD)
            readp(fp); else
            writep(fp);
    } else {
        u.u_offset[1] = fp->f_offset[1];
        u.u_offset[0] = fp->f_offset[0];
        if(m==FREAD)
            readi(fp->f_inode); else
            writei(fp->f_inode);
        dpadd(fp->f_offset, u.u_arg[1]-u.u_count);
    }
    u.u_ar0[R0] = u.u_arg[1]-u.u_count;
}
```

6~8　执行 `getf()`，取得与文件描述符相对应的 `file[]` 元素。

9~12　将`file[]` 元素的标志位和通过参数设定的读写模式相比较，确认是否具有所需的权限。

13~15　为了执行 `readi()` 和 `writei()`，首先设定 `u.u_base`、`u.u_count` 和 `u.u_segflg`。

16~19　如果准备读写的文件为管道时的处理。请参照第 11 章。

20~27　如果准备读写的文件为一般文件时的处理。将 `u.u_offset` 设定为 `file[]` 元素的文件偏移量。到此为止，所需的参数都已设置完毕，可以执行 `readi()` 或 `writei()`。执行结束后，将 `file[]` 元素的文件偏移量加上实际读写的字节数。

28　将用户进程的 `r0` 设定为实际读写的字节数，并将其作为系统调用的返回值。

readi()

`readi()` 用于读取文件（表 10-13，代码清单 10-14 ）。地址和传送量等参数通过 `user` 结构体进行设定（表 10-12 ）。

执行 `bmap()`，将逻辑块编号变换为物理块编号，并以块单位（512 字节）进行读取操作。如果是对分散在 8 个块中长度为 4KB 的数据进行读取，需要读取 8 次。

`readi()` 具有预读取的功能。如果判断正在对某个文件的块进行连续读取时，将通过 `breada()` 进行异步的预读取处理。

表 **10-12　readi()** 的参数

参数	含义
u.u_base	存放读取数据的虚拟地址（以字节为单位）
u.u_offset	文件中的偏移量（以字节为单位）
u.u_count	读取数据量（以字节为单位）
u.u_segflg	1：读取至内核空间，0：读取至用户空间

表 **10-13　readi()** 的参数

参数	含义
aip	inode[] 元素

代码清单 **10-14　readi()**（**ken/rdwri.c)**

```
readi(aip)
struct inode *aip;
{
    int *bp;
    int lbn, bn, on;
    register dn, n;
```

```
7       register struct inode *ip;
8
9       ip = aip;
10      if(u.u_count == 0)
11          return;
12      ip->i_flag =| IACC;
13      if((ip->i_mode&IFMT) == IFCHR) {
14          (*cdevsw[ip->i_addr[0].d_major].d_read)(ip->i_addr[0]);
15          return;
16      }
17
18      do {
19          lbn = bn = lshift(u.u_offset, -9);
20          on = u.u_offset[1] & 0777;
21          n = min(512-on, u.u_count);
22          if((ip->i_mode&IFMT) != IFBLK) {
23              dn = dpcmp(ip->i_size0&0377, ip->i_size1,
24                  u.u_offset[0], u.u_offset[1]);
25              if(dn <= 0)
26                  return;
27              n = min(n, dn);
28              if ((bn = bmap(ip, lbn)) == 0)
29                  return;
30              dn = ip->i_dev;
31          } else {
32              dn = ip->i_addr[0];
33              rablock = bn+1;
34          }
35          if (ip->i_lastr+1 == lbn)
36              bp = breada(dn, bn, rablock);
37          else
38              bp = bread(dn, bn);
39          ip->i_lastr = lbn;
40          iomove(bp, on, n, B_READ);
41          brelse(bp);
42      } while(u.u_error==0 && u.u_count!=0);
43  }
```

10~11 如果读取的数据长度为 0 时，不做任何处理立即返回。

12 设置 `inode[]` 元素的更新标志位。

13~16 如果文件是字符设备的特殊文件，则调用该设备的设备驱动。

19 通过文件偏移量计算得到该数据的逻辑块编号。

20 计算块内的偏移量。

21 计算读取的数据长度。并对其调整使其不会跨越多个块。

22~30 读取对象为一般文件时的处理。对每次读取的长度做最终调整，使其不会超过文件的长度。然后执行 `bmap()`，取得存储区域的物理块编号。将 `dn` 设定为块设备编号。

31~34 如果读取对象是块设备的特殊文件，则将 `dn` 设定为块设备编号，并将 `rablock` 设定为准备读取的块的下一个块的块编号。

35~38 如果读取对象为文件中连续的逻辑块，则执行 `breada()` 读取当前准备读取的块，并对下一个块进行异步的预读取。否则执行 `bread()` 进行一般读取。

39 为了进行预读取的判断，将 `inode.i_lastr` 设定为所读取的逻辑块编号。

40 执行 `iomove()`，将读取的磁盘数据从缓冲区复制到进程的虚拟地址。

41 释放读取数据时使用的缓冲区。

42 如果没有发生错误，且需要的数据尚未读取完毕时，继续做读取处理。

writei()

`writei()` 用于对文件进行写入处理（表 10-15，代码清单 10-15 ）。与 `readi()` 相同，地址和传送量等参数也通过 `user` 结构体进行设定（表 10-14 ）。

执行 `bmap()`，将逻辑块编号变换为物理块编号，并以块单位（512 字节）进行写入操作。如果是对分散在 8 个块中长度为 4KB 的数据进行写入，则需要写入 8 次。

如果需要对整个块写入数据，首先执行 `getblk()` 取得该块的缓冲区。因为接下来要对块进行覆盖处理，所以此时无需从磁盘读取块中的数据。如果只是更新块中的一部分内容，则需要首先执行 `bread()` 从磁盘读取块中当前的内容，然后将需要更新的那一部分写入缓冲区，再写入磁盘。

对磁盘进行写入时，如果遇到块的边界数据，将被立即写入磁盘。否则将进行延迟写入，数据不会马上写入磁盘。

表 **10-14　writei()** 的参数

参数	含义
u.u_base	准备被写入的数据的虚拟地址（以字节为单位）
u.u_offset	文件中的偏移量（以字节为单位）
u.u_count	写入的数据量（以字节为单位）
u.u_segflg	1：从内核空间写入，0：从用户空间写入

表 **10-15　writei()** 的参数

参数	含义
aip	inode[] 元素

代码清单 **10-15　writei()**（**ken/rdwri.c**）

```
writei(aip)
struct inode *aip;
{
    int *bp;
    int n, on;
    register dn, bn;
    register struct inode *ip;

    ip = aip;
    ip->i_flag =| IACC|IUPD;
    if((ip->i_mode&IFMT) == IFCHR) {
        (*cdevsw[ip->i_addr[0].d_major].d_write)(ip->i_addr[0]);
        return;
    }
```

```
15     if (u.u_count == 0)
16         return;
17
18     do {
19         bn = lshift(u.u_offset, -9);
20         on = u.u_offset[1] & 0777;
21         n = min(512-on, u.u_count);
22         if((ip->i_mode&IFMT) != IFBLK) {
23             if ((bn = bmap(ip, bn)) == 0)
24                 return;
25             dn = ip->i_dev;
26         } else
27             dn = ip->i_addr[0];
28         if(n == 512)
29             bp = getblk(dn, bn); else
30             bp = bread(dn, bn);
31         iomove(bp, on, n, B_WRITE);
32         if(u.u_error != 0)
33             brelse(bp); else
34         if ((u.u_offset[1]&0777)==0)
35             bawrite(bp); else
36             bdwrite(bp);
37         if(dpcmp(ip->i_size0&0377, ip->i_size1,
38          u.u_offset[0], u.u_offset[1]) < 0 &&
39          (ip->i_mode&(IFBLK&IFCHR)) == 0) {
40            ip->i_size0 = u.u_offset[0];
41            ip->i_size1 = u.u_offset[1];
42         }
43         ip->i_flag =| IUPD;
44     } while(u.u_error==0 && u.u_count!=0);
45 }
```

10 设置 `inode[]` 元素的参照标志位和更新标志位。

11~14 如果准备写入的文件为字符设备的特殊文件，则调用该设备的设备驱动。

15~16 如果准备写入的数据长度为 0，则不做任何处理。

19~21 与 `readi()` 时的处理相同，计算出逻辑块编号、块偏移量和写入数据长度（对其进行调整使数据不会跨越多个块）。

22~25 如果准备写入的文件为一般文件时，执行 `bmap()` 获取物理块编号，并将 `dn` 设定为设备编号。

26~27 如果准备写入的文件为块设备的特殊文件，则将 `dn` 设定为设备编号。

28~30 如果需要向整个块写入数据，首先执行 `getblk()` 获取该块的缓冲区。如果只是更新块中的部分内容，则首先执行 `bread()` 从磁盘读取块当前的内容。

31 执行 `iomove()` 将准备写入的数据从虚拟地址空间传送至缓冲区。

34~36 遇到块边界时，执行 `bawrite()` 以异步方式将数据立即写入磁盘。否则执行 `bdwrite()` 进行延迟写入。

37~42 如果由于写入使文件长度增大，则增加 `inode.i_size` 的值。`u.u_offset` 的值为此时文件末尾的地址。

43 设置 `inode[]` 元素的更新标志位。

44 如果没有发生错误，且需要的数据尚未写完时，继续写入处理。

iomove()

`iomove()` 用于在虚拟地址空间和块设备缓冲区之间传送数据（表 10-17，代码清单 10-16 ）。与 `readi()` 和 `writei()` 相同，内存地址及传送数据长度等通过 `user` 结构体指定。请注意，`user` 结构体的内容将被更新（表 10-16 ）。

当以字为单位对用户空间进行传送时，使用汇编语言编写的 `copyin()` 和 `copyout()` 。否则使用 `cpass()` 和 `passc()` 以字节为单位进行传送。

表 **10-16　iomove()** 的参数

参数	含义
`u.u_base`	读写对象的虚拟地址（以字节为单位）。处理结束后将递增实际传送的长度值
`u.u_offset`	文件中的偏移量（以字节为单位）。处理结束后将被递增实际传送的长度值
`u.u_count`	读写数据量（以字节为单位）。处理结束后将被递减实际传送的长度值
`u.u_segflg`	1：从内核空间读写，0：从用户空间读写

表 **10-17　iomove()** 的参数

参数-	含义
`bp`	块设备缓冲区
`o`	缓冲区内的偏移量（以字节为单位）
`an`	数据传送长度（以字节为单位）
`flag`	设定是读取还是写入

代码清单 **10-16　iomove()**（**ken/rdwri.c)**

```
iomove(bp, o, an, flag)
struct buf *bp;
{
    register char *cp;
    register int n, t;

    n = an;
    cp = bp->b_addr + o;
    if(u.u_segflg==0 && ((n | cp | u.u_base)&01)==0) {
        if (flag==B_WRITE)
            cp = copyin(u.u_base, cp, n);
        else
            cp = copyout(cp, u.u_base, n);
        if (cp) {
            u.u_error = EFAULT;
            return;
        }
```

```
18         u.u_base =+ n;
19         dpadd(u.u_offset, n);
20         u.u_count =- n;
21         return;
22     }
23     if (flag==B_WRITE) {
24         while(n--) {
25             if ((t = cpass()) < 0)
26                 return;
27             *cp++ = t;
28         }
29     } else
30         while (n--)
31             if(passc(*cp++) < 0)
32                 return;
33 }
```

9~22　从用户空间传送数据。如果传送长度、缓冲区内偏移量、内存地址都以字为单位（偶数），则使用 `copyin()` 和 `copyout()`。

23~32　否则使用 `passc()`（表 10-18，代码清单 10-17）和 `cpass()`（代码清单 10-18）。这两个函数以字节为单位传送数据，如果是在内核空间内部进行传送，只需进行单纯的数据复制即可。如果是在用户空间和内核空间之间传送数据，则需要调用 `subyte()` 和 `fubyte()`。

表 **10-18**　**passc()** 的参数

参数	含义
c	传送的数据（字节）

代码清单 **10-17**　**passc()**（**ken/subrc.c**）

```
passc(c)
char c;
{
    if(u.u_segflg)
        *u.u_base = c; else
        if(subyte(u.u_base, c) < 0) {
            u.u_error = EFAULT;
            return(-1);
        }
    u.u_count--;
    if(++u.u_offset[1] == 0)
        u.u_offset[0]++;
    u.u_base++;
    return(u.u_count == 0? -1: 0);
}
```

代码清单 **10-18**　**cpass()**（**ken/subrc.c**）

```
cpass()
{
    register c;

    if(u.u_count == 0)
        return(-1);
    if(u.u_segflg)
```

```
        c = *u.u_base; else
        if((c=fubyte(u.u_base)) < 0) {
            u.u_error = EFAULT;
            return(-1);
        }
    u.u_count--;
    if(++u.u_offset[1] == 0)
        u.u_offset[0]++;
    u.u_base++;
    return(c&0377);
}
```

getf()

`getf()` 根据用户程序指定的文件描述符，返回对应的 `user.u_ofile[]` 元素（表 10-19，代码清单 10-19 ）。

表 **10-19**　**getf()** 的参数

参数	含义
f	文件描述符

代码清单 **10-19**　**getf()**（**ken/fio.c)**

```
getf(f)
{
    register *fp, rf;

    rf = f;
    if(rf<0 || rf>=NOFILE)
        goto bad;
    fp = u.u_ofile[rf];
    if(fp != NULL)
        return(fp);
bad:
    u.u_error = EBADF;
    return(NULL);
}
```

10.5　指定文件的读写位置

系统调用 **seek**

`seek()` 为系统调用 `seek` 的处理函数（表 10-20，代码清单 10-20 ）。该函数用来调整文件的偏移量，可以以块为单位（512 字节）对偏移量进行增减。

首先根据文件描述符获取 `user.u_ofile[]` 指向的 `file[]` 元素，然后更改文件偏移量的值。

表 **10-20**　系统调用 **seek** 的参数

参数	含义

r0	文件描述符
u.u_arg[0]	偏移量的增减值。单位为字节或块（512 字节）
u.u_arg[1]	模式。指定偏移量增减的起点及单位（字节或块）。0、3：文件的起始位置。1、4：当前偏移量。2、5： 文件的末尾位置。0~2 时以字节为单位，3~5 时以块为单位。0、3 时偏移量增减值的类型为 unsigned

代码清单 **10-20　seek()**（**ken/sys2.c)**

```
 1 seek()
 2 {
 3     int n[2];
 4     register *fp, t;
 5
 6     fp = getf(u.u_ar0[R0]);
 7     if(fp == NULL)
 8         return;
 9     if(fp->f_flag&FPIPE) {
10         u.u_error = ESPIPE;
11         return;
12     }
13     t = u.u_arg[1];
14     if(t > 2) {
15         n[1] = u.u_arg[0]<<9;
16         n[0] = u.u_arg[0]>>7;
17         if(t == 3)
18             n[0] =& 0777;
19     } else {
20         n[1] = u.u_arg[0];
21         n[0] = 0;
22         if(t!=0 && n[1]<0)
23             n[0] = -1;
24     }
25     switch(t) {
26
27     case 1:
28     case 4:
29         n[0] =+ fp->f_offset[0];
30         dpadd(n, fp->f_offset[1]);
31         break;
32
33     default:
34         n[0] =+ fp->f_inode->i_size0&0377;
35         dpadd(n, fp->f_inode->i_size1);
36
37     case 0:
38     case 3:
39         ;
40     }
41     fp->f_offset[1] = n[1];
42     fp->f_offset[0] = n[0];
43 }
```

6~8　取得与文件描述符相对应的 `file[]` 元素。

9~12　无法调整管道的偏移量。

14~18　如果模式大于等于 3，则将偏移量的增减值调整为以块为单位。因为模式为 3 时偏移量的增减值的类型必须为 unsigned，所以只取出数值的部分。

19~24　如果模式小于等于 2，偏移量增减值则以字节为单位。但因为模式为 1、2 时增减值的类型为

signed，所以当增减值为负数时需要将 n 的高位字 n[0] 设置为 -1，使 n 全体都为负值。

25　确定新的文件偏移量。

27~31　以当前的文件偏移量为基准对数值进行增减。

33~35　以文件长度（文件的末尾）为基准对数值进行增减。

37~39　因为是以文件的起始位置为基准对数值进行增减，所以不需要进行特别处理。

41~42　更新文件偏移量。

10.6　关闭文件

系统调用 close

close() 是系统调用 close 的处理函数（表 10-21，代码清单 10-21），用来关闭文件。参数为文件描述符，系统调用 close 将相应的 user.u_ofile[] 元素设置为 NULL，然后执行 closef()，递减 user.u_ofile[] 元素原本指向的 file[] 元素的参照计数器。

表 **10-21**　系统调用 **close** 的参数

参数	含义
r0	文件描述符

代码清单 **10-21**　**close()**（**ken/sys2.c)**

```
close()
{
    register *fp;

    fp = getf(u.u_ar0[R0]);
    if(fp == NULL)
        return;
    u.u_ofile[u.u_ar0[R0]] = NULL;
    closef(fp);
}
```

closef()

closef() 用来递减 file[] 元素的参照计数器的值（表 10-22，代码清单 10-22 ）。当参照计数器的值变为 0 时执行 closei()，递减 file[] 元素指向的 inode[] 元素的参照计数器的值。

表 **10-22**　**closef()** 的参数

参数	含义
fp	file[] 元素的指针

代码清单 **10-22　closef()**（**ken/fio.c)**

```
 1 closef(fp)
 2 int *fp;
 3 {
 4     register *rfp, *ip;
 5
 6     rfp = fp;
 7     if(rfp->f_flag&FPIPE) {
 8         ip = rfp->f_inode;
 9         ip->i_mode =& ~(IREAD|IWRITE);
10         wakeup(ip+1);
11         wakeup(ip+2);
12     }
13     if(rfp->f_count <= 1)
14         closei(rfp->f_inode, rfp->f_flag&FWRITE);
15     rfp->f_count--;
16 }
```

7~12　文件为管道时的处理。请参考第 11 章。

closei()

`closei()` 执行 `iput()` 递减 `inode[]` 元素的参照计数器的值（代码清单 10-23，表 10-23 ）。文件为特殊文件，且当参照计数器的值变为 0 时进行关闭设备的处理。

表 **10-23　closei()** 的参数

参数	含义
ip	inode[] 元素
rw	读写模式

代码清单 **10-23　closei()**（**ken/fio.c)**

```
closei(ip, rw)
int *ip;
{
    register *rip;
    register dev, maj;

    rip = ip;
    dev = rip->i_addr[0];
    maj = rip->i_addr[0].d_major;
    if(rip->i_count <= 1)
    switch(rip->i_mode&IFMT) {

    case IFCHR:
        (*cdevsw[maj].d_close)(dev, rw);
        break;

    case IFBLK:
        (*bdevsw[maj].d_close)(dev, rw);
    }
    iput(rip);
}
```

10.7　目录的生成

系统调用 mknod

mknod() 为系统调用 mknod 的处理函数（表 10-24，代码清单 10-24 ），用于生成目录或特殊文件。该系统调用只有超级用户才能执行，一般用户试图执行时不做任何处理并引发错误。

表 **10-24**　系统调用 **mknod** 的参数

参数	含义
u.u_arg[0]	文件路径名
u.u_arg[1]	模式。被赋予 inode.i_mode
u.u_arg[2]	设备编号。被赋予 inode.i_addr[0]。生成目录时值为 0

代码清单 **10-24**　**mknod()**（**ken/sys2.c)**

```
 1 mknod()
 2 {
 3     register *ip;
 4     extern uchar;
 5
 6     if(suser()) {
 7         ip = namei(&uchar, 1);
 8         if(ip != NULL) {
 9             u.u_error = EEXIST;
10             goto out;
11         }
12     }
13     if(u.u_error)
14         return;
15     ip = maknode(u.u_arg[1]);
16     if (ip==NULL)
17         return;
18     ip->i_addr[0] = u.u_arg[2];
19
20 out:
21     iput(ip);
22 }
```

6　超级用户时的处理。为了使一般用户也可以生成目录，有的系统会对生成目录的命令 mkdir 设置 SUID 标志位。

7~11　利用 namei() 取得与用户指定的路径名相对应的 inode[] 元素。因为准备生成新的文件或目录，所以如果与路径名相对应的 inode[] 元素已经存在，则进行错误处理。

13~14　如果当前用户不是超级用户，suser() 则会将错误代码赋予 u.u_error 。另外，在 namei() 中出错时也会将错误代码赋予 u.u_error 。

15~17　执行 maknode() 生成新的文件。

18　将 `inode.i_addr[0]` 设置为用户通过参数指定的值。

21　生成文件或执行 `namei()` 时会递增 `inode[]` 元素的参照计数器的值，此处则进行递减。

10.8　文件的链接

链接用于实现所谓的“硬链接”（hard link），在 UNIX V6 中没有符号链接的概念。

通过链接可以对一个文件（inode + 存储区域的块）赋予多个名称。例如，对名为 /var/hoge 的文件设定名为 /var/homu 的链接后，就可以通过 /var/hoge 或 /var/homu 的路径名访问同一个文件。

新旧名称具有相同的作用。即使删除了 /var/hoge，/var/homu 仍旧处于可参照的状态，因此文件本身不会被删除。文件具有的@称的数量称为被链接数**,**由 **`inode.i_nlink`** 管理**,**只要被链接数的值不为 **0,**文件就不会被删除 。

与此相反，如果是符号链接，当删除原本的名称“/var/hoge”时，文件本身也会被删除。符号链接具有主从关系，链接（从）可以被看做仅仅是指向原有的文件（主）。

硬链接存在若干问题。首要问题是无法跨设备设定链接 。目录的对应表只记录了 inode 编号和文件名的信息，只通过 inode 编号无法判断该 inode 属于哪个设备。

除此之外，无法对目录设定链接 也是问题之一。这会使命名空间中出现循环，损坏文件路径的一致性。为了解决上述问题，导入了符号链接的概念。

在 UNIX V6 中，超级用户可以对目录设定链接。出于安全性的考虑，最近的一些操作系统开始限制超级用户设定目录的链接。

系统调用 **link**

`link()` 为系统调用 `link` 的处理函数（表 10-25，代码清单 10-25 ），用于设定链接。如果被链接的文件不存在，或与新赋予的路径名相对应的文件已经存在时将引发错误。此外，被链接数的上限为 127，超过此数值时也会引发错误。

表 **10-25**　系统调用 **link** 的参数

参数	含义
`u.u_arg[0]`	被链接的文件的路径名
`u.u_arg[1]`	新赋予的路径名

代码清单 **10-25**　**link()**（**ken/sys2.c)**

```
link()
{
    register *ip, *xp;
    extern uchar;

    ip = namei(&uchar, 0);
    if(ip == NULL)
        return;
    if(ip->i_nlink >= 127) {
        u.u_error = EMLINK;
        goto out;
    }
```

```
13     if((ip->i_mode&IFMT)==IFDIR && !suser())
14         goto out;
15     ip->i_flag =& ~ILOCK;
16     u.u_dirp = u.u_arg[1];
17     xp = namei(&uchar, 1);
18     if(xp != NULL) {
19         u.u_error = EEXIST;
20         iput(xp);
21     }
22     if(u.u_error)
23         goto out;
24     if(u.u_pdir->i_dev != ip->i_dev) {
25         iput(u.u_pdir);
26         u.u_error = EXDEV;
27         goto out;
28     }
29     wdir(ip);
30     ip->i_nlink++;
31     ip->i_flag =| IUPD;
32
33 out:
34     iput(ip);
35 }
```

6~8 取得与被链接文件的路径名相对应的 `inode[]` 元素。

9~12 如果超过链接数的上限则引发错误。

13~14 如果是目录，只有超级用户才可以设定链接。

15 解锁获取的 `inode[]` 元素。在 `namei()` 调用的 `iget()` 中对该元素加锁。

16~21 尝试取得与用户通过参数指定的第 2 个路径名相对应的 `inode[]` 元素。因为希望用该路径设定链接，所以如果取得了与该路径相对应的 `inode[]` 元素（即已经存在同名的文件）则引发错误。

22~23 当 `namei()` 中发生错误，或是取得了 `inode[]` 元素时，`u.u_error` 将被赋予错误代码。

24~28 在不同的设备之间无法设定链接。在第 17 行执行的 `namei()` 中，第 2 个路径名所指向的文件的父目录被赋予 `u.u_pdir` 。

29 将与第 1 个路径名相对应的 `inode[]` 元素，追加至由第 2 个路径名指向的文件父目录的对应表中。由于在第 17 行执行的 `namei()` 中已将与第 2 个路径名相对应的文件名赋予了 `u.u_dent` ，因此对应表记录中的文件名是与第 2 个路径名相对应的文件名。

30~31 递增与被链接文件相对应的 `inode[]` 元素的被链接数，并设置 `inode[]` 元素的更新标志位。

34 递减由 `namei()` 中的 `iget()` 递增的 `inode[]` 元素的参照计数器的值。

suser()

`suser()` 检查执行进程是否由超级用户执行（代码清单 10-26 ）。如果执行者为超级用户，`user.u_uid` 则为 0。

代码清单 **10-26　suser()（ken/fio.c)**

```
suser()
{

    if(u.u_uid == 0)
```

```
        return(1);
    u.u_error = EPERM;
    return(0);
}
```

10.9　删除文件

系统调用unlink

`unlink()` 为系统调用 `unlink` 的处理函数（表 10-27，代码清单 10-27 ），用来删除文件。目录只有超级用户才能删除。

删除文件是通过将目录对应表相应记录中的 inode 编号设置为 0 得以实现的。删除某个目录下名为“hoge.txt”的文件时的例子如表 10-26 所示。

表 **10-26**　文件删除

inode 编号	文件名
0x10	mado.txt
0x16	homu.txt
0x20-> 0x00	hoge.txt
0x35	fuga.dat

此外，系统调用 `unlink` 将递减 `inode.i_nlink` 。当 `inode.i_nlink` 的值变为 0 时，该 inode 以及所使用的存取区域将被释放。但是，存储区域中保存的数据将维持原状直到被其他数据覆盖。删除文件可看做是消除了访问存储区域的途径。

表 **10-27**　系统调用 **unlink** 的参数

参数	含义
`u.u_arg[0]`	准备删除的文件的路径名

代码清单 **10-27**　**unlink()**（**ken/sys4.c)**

```
unlink()
{
    register *ip, *pp;
    extern uchar;

    pp = namei(&uchar, 2);
    if(pp == NULL)
        return;
    prele(pp);
    ip = iget(pp->i_dev, u.u_dent.u_ino);
    if(ip == NULL)
```

```
12        panic("unlink -- iget");
13    if((ip->i_mode&IFMT)==IFDIR && !suser())
14        goto out;
15    u.u_offset[1] =- DIRSIZ+2;
16    u.u_base = &u.u_dent;
17    u.u_count = DIRSIZ+2;
18    u.u_dent.u_ino = 0;
19    writei(pp);
20    ip->i_nlink--;
21    ip->i_flag =| IUPD;
22
23 out:
24    iput(pp);
25    iput(ip);
26 }
```

6~8　将 namei() 的第 2 个参数设定为 2，获取与准备删除的文件的父目录相对应的 inode[] 元素。

9　该元素已由 namei() 加锁，此处对其解锁。

10~12　取得与路径名相对应的 inode[] 元素。u.u_dent.u_ino 已由 namei() 进行了设定。

13~14　如果不是超级用户则无法删除目录。但是，也存在其他 UNIX 版本，为删除目录的命令 rmdir 设定 SUID 位，使得一般用户也可以删除目录。

15~19　执行 writei()，修改目录对应表。u.u_dent 和 u.u_offset 已由 namei() 进行了设定。因为 u.u_dent.u_ino 的值为 0，对应表中相应记录的 inode 编号也将变为 0。

20~21　递减 inode[] 元素的被链接数，并设置更新标志位。

24~25　递减与父目录和被删除文件相对应的 inode[] 元素的参照计数器的值。

10.10　小结

- 已打开的文件由 user.u_ofile[]、file[]、inode[] 3 个结构体管理。
- 文件打开后，user.u_ofile[] 的数组下标，即文件描述符被返还给用户程序。
- file[] 用来管理文件偏移量等操作文件的数据。
- read 和 write 的处理以块为单位进行。
- 删除文件等同于将目录对应表的相应记录的 inode 编号设置为 0。
- 通过链接可以为文件设定多个名称。

第 **11** 章　管道

11.1　什么是管道

管道是在父进程和子进程之间通信的机制。因为进程拥有各自独立的虚拟地址空间，所以任意的进程是无法直接访问其他进程拥有的数据的。为了实现进程间的通信，于是设计了管道。

管道对文件系统（的一部分）进行了巧妙应用，使得进程间的通信成为可能。管道首先获取根磁盘的 inode，然后利用该 inode 指向的存储区域进行数据交换。这个文件（**inode** 和存储区域）构成了管道的实体 。管道的容量由 `PIPSIZ` 定义，缺省值为 4096 字节（代码清单 11-1 ）。

代码清单 **11-1**　**PIPSIZ**（**ken/pipe.c)**

```
#define    PIPSIZ    4096
```

利用管道进行的通信流程如下所示（图 11-1 ）。

1. 发送方的进程向管道写入数据，直到管道被充满。

2. 切换至接收方的进程，使得从管道读入数据。已经接收的数据从管道中被删除。

3. 数据全部读取后，切换至发送方的进程，返回1. 的处理。

发送方以与管道相对应的 `inode[]` 元素的地址 +1、接收方以该地址 +2 为参数执行 `sleep()` 。

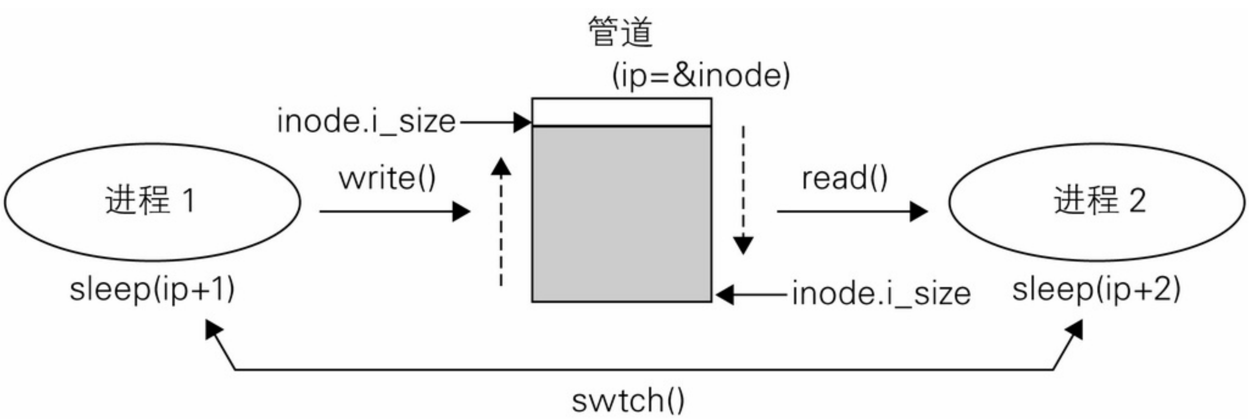


图 **11-1**　管道

使用管道的优点

在进程间传递数据也可通过临时文件来实现（代码清单 11-2 ）。

代码清单 **11-2**　临时文件

```
% ./sender > tmp
% ./receiver < tmp
```

与临时文件相比，使用管道具有下述优点。首先，可使用的块设备的资源是有限的 。管道的容量为固定的 4096 字节，因此即使用来交换更大容量的数据也不会占用更多的块设备区域。而使用临时文件时，会占用与所交换的数据相等容量的块设备区域。

其次，管道所需要的缓冲区的容量小于缓冲区的总容量，利于发挥块设备缓冲区的缓存功能 。而且，当发送方的进程将数据写入管道后，接收方的进程将马上进行读取，块设备的缓存效果会更加明显。

与之相反，当使用临时文件时，如果需要输出大于块设备缓冲区容量的文件，缓冲区所带来的缓存效果将因此受到影响。

但是，当使用管道，且执行进程由发送方切换至接收方时，如果存在执行优先级更高的进程，且该进程也使用了块设备的缓冲区时，则无法期待缓冲区带来的缓存效果。此时，性能虽然并未得到改善，但是由于数据已输出至块设备,因此对通信内容本身不会造成影响 。

管道基于现有的文件系统，实现的成本较低。尽管占用的资源较少，却实现了进程间的高速通信。低投入高产出可视作管道最大的魅力 。

11.2　开始管道通信

系统调用 pipe

系统调用 `pipe` 用来建立管道通信，`pipe()` 为其处理函数（代码清单 11-3 ）。

首先在 `user.u_ofile[]` 和 `file[]` 中分配供 `read` 和 `write` 使用的元素，然后获取根磁盘的 `inode[]` 元素，将供 `read` 和 `write` 使用的 `file[]` 元素指向该 `inode[]` 元素。为 `file[]` 元素设置表示管道的 `FPIPE` 标志位（图 11-2 ）。

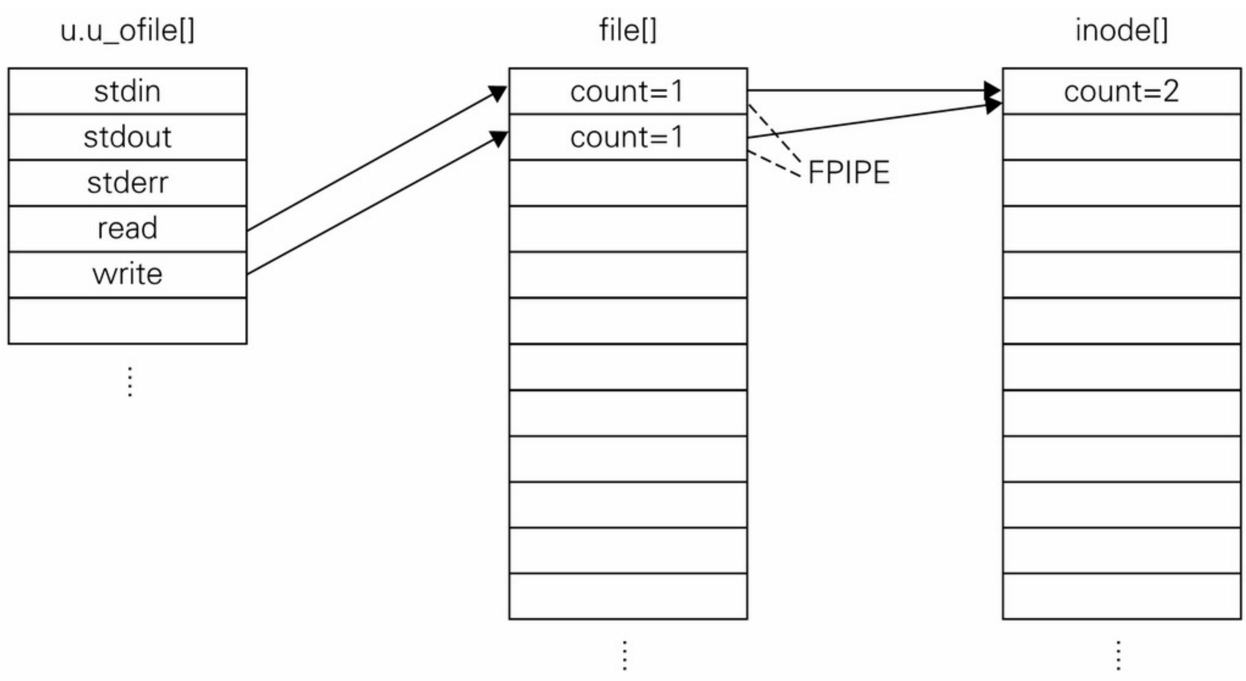


图 11-2　pipe()

`read` 和 `write` 使用的文件描述符返还给用户程序。用户程序对待该文件描述符，可以像对待一般文件一样进行读写，实现管道通信。

代码清单 **11-3　pipe()**（**ken/pipe.c)**

```
 1 pipe()
 2 {
 3     register *ip, *rf, *wf;
 4     int r;
 5
 6     ip = ialloc(rootdev);
 7     if(ip == NULL)
 8         return;
 9     rf = falloc();
10     if(rf == NULL) {
11         iput(ip);
12         return;
13     }
14     r = u.u_ar0[R0];
15     wf = falloc();
16     if(wf == NULL) {
17         rf->f_count = 0;
18         u.u_ofile[r] = NULL;
19         iput(ip);
20         return;
21     }
22     u.u_ar0[R1] = u.u_ar0[R0];
23     u.u_ar0[R0] = r;
24     wf->f_flag = FWRITE|FPIPE;
25     wf->f_inode = ip;
26     rf->f_flag = FREAD|FPIPE;
27     rf->f_inode = ip;
28     ip->i_count = 2;
29     ip->i_flag = IACC|IUPD;
30     ip->i_mode = IALLOC;
31 }
```

6~8　从根磁盘的 inode[] 获取新的元素。

9~14　在 user.u_ofile[] 和 file[] 中分配供 read 使用的元素。falloc() 中将所取得 user.u_ofile[] 元素的数组下标（文件描述符）赋予 u.u_ar0[R0]，此处再将其保存在局部变量 r 中。

15~23　同样，在 user.u_ofile[] 和 file[] 中分配供 write 使用的元素。用户进程的 r1 中保存了供 write 使用的文件描述符，r0 中保存了供 read 使用的文件描述符。这两个值将被返回给用户程序。

24~30　对取得的元素进行初始化。为供 read 使用的 file[] 元素设置 FREAD 、FPIPE 标志位，为供 write 使用的 file[] 元素设置 FWRITE 、FPIPE 标志位。无论是 read 还是 write 使用的 file[] 元素，都使其指向之前取得的根磁盘的 inode[] 元素。将 inode[] 元素的参照计数器的值设定为 2 ，并设置参照标志位和更新标志位，最后将 inode.i_mode 设置为 IALLOC 。

11.3　收发数据

对通过系统调用 pipe 取得的文件描述符，可以像对待一般文件那样执行系统调用 read 和 write ，从而实现数据收发（代码清单 11-4 ）。

代码清单 **11-4　rdwr()** 中的相关代码（**ken/sys2.c)**

```
    u.u_base = u.u_arg[0];
    u.u_count = u.u_arg[1];
    u.u_segflg = 0;
    if(fp->f_flag&FPIPE) {
```

```
17          if(m==FREAD)
18              readp(fp); else
19              writep(fp);
20      } else {
21          u.u_offset[1] = fp->f_offset[1];
22          u.u_offset[0] = fp->f_offset[0];
23          if(m==FREAD)
24              readi(fp->f_inode); else
25              writei(fp->f_inode);
26          dpadd(fp->f_offset, u.u_arg[1]-u.u_count);
27      }
28      u.u_ar0[R0] = u.u_arg[1]-u.u_count;
```

16~19 当设置了 `file[]` 元素的 `FPIPE` 标志位时，在 `rdwr()` 中将执行 `writep()` 和 `readp()` 。

writep()

`writep()` 用来对管道进行写入处理（表 11-1，代码清单 11-5 ）。因为管道的实体为文件，所以采用与对待一般文件相同的方式调用 `writei()` 写入数据。当管道被充满（4096 字节）时进入睡眠状态。如果存在等待管道被写入数据的进程，则将其唤醒。

但是，与一般文件处理不同,文件偏移量（**`file.f_offset`** ）不会发生变化 （图 11-3 ）。

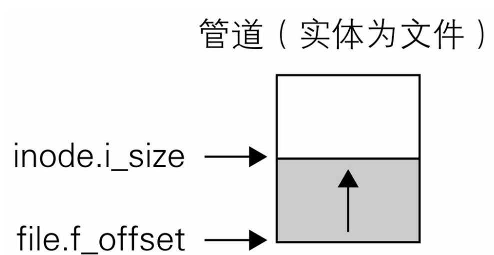


图 **11-3　writep()**

表 **11-1　writep()** 的参数

参数	含义
`fp`	`file[]` 元素

代码清单 **11-5　writep()（ken/pipe.c)**

```
writep(fp)
{
    register *rp, *ip, c;

    rp = fp;
    ip = rp->f_inode;
    c = u.u_count;

loop:
    plock(ip);
    if(c == 0) {
        prele(ip);
        u.u_count = 0;
        return;
    }
```

```
16     if(ip->i_count < 2) {
17         prele(ip);
18         u.u_error = EPIPE;
19         psignal(u.u_procp, SIGPIPE);
20         return;
21     }
22     if(ip->i_size1 == PIPSIZ) {
23         ip->i_mode =| IWRITE;
24         prele(ip);
25         sleep(ip+1, PPIPE);
26         goto loop;
27     }
28     u.u_offset[0] = 0;
29     u.u_offset[1] = ip->i_size1;
30     u.u_count = min(c, PIPSIZ-u.u_off set[1]);
31     c =- u.u_count;
32     writei(ip);
33     prele(ip);
34     if(ip->i_mode&IREAD) {
35         ip->i_mode =& ~IREAD;
36         wakeup(ip+2);
37     }
38     goto loop;
39 }
```

10 加锁 inode[] 元素。

11~15 当数据输出完毕后为 inode[] 元素解锁。将 u.u_count 清 0 并返回。

16~21 如果接收方的管道被关闭，inode[] 元素的参照计数器的值将小于 2。这会造成数据无法被发送，因此进行出错处理。为 inode[] 元素解锁，并向自身发送信号。

22~27 如果管道被充满，则为 inode[] 元素设置 IWRITE 标志位，并为 inode[] 元素解锁，然后进入睡眠状态等待资源 ip+1 。在管道处理中，IWRITE 标志位表示管道已被充满，发送方正在等待数据被使用。在被接收方唤醒之后，发送方将返回 loop ，继续向管道写入数据。

28~31 将 u.u_offset 设定为当前的文件长度，将 u.u_count 设定为小于管道容量（4096 字节）的值。局部变量 c 中保存着准备写入的数据的剩余字节数。

32 执行 writei() 写入数据。

33 为 inode[] 元素解锁。

34~37 如果设置了 inode[] 元素的 IREAD 标志位，则将其清除然后唤醒接收方的进程。

38 返回 loop ，继续向管道写入数据的处理。

readp()

readp() 从管道读取数据（表 11-2，代码清单 11-6 ）。因为管道的实体为文件，所以采用与对待一般文件相同的方式调用 readi() 读取数据。当管道变空时进入睡眠状态。如果存在因管道被充满而等待其他读取数据的进程，则将其唤醒。

文件偏移量将随着从管道中读取的数据增大，当偏移量与文件长度同值时，文件长度与文件指针都将归 0（图 11-4 ）。

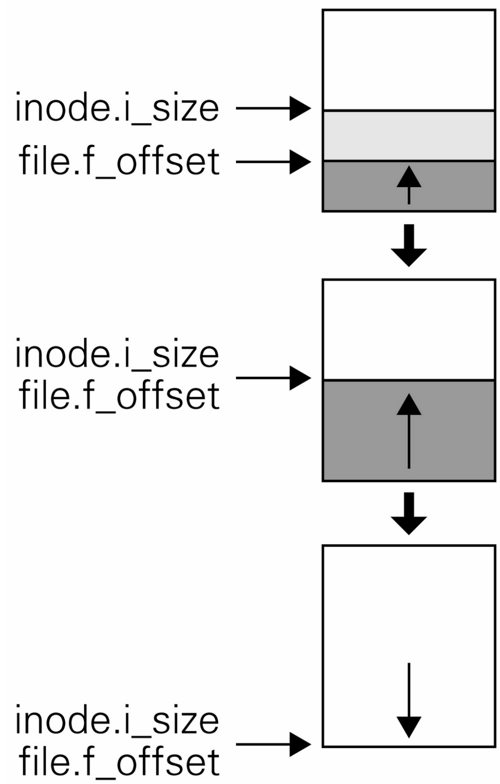


图 **11-4　readp()**

表 **11-2　readp()** 的参数

参数	含义
fp	file[] 元素

代码清单 **11-6　readp()**（**ken/pipe.c)**

```
readp(fp)
int *fp;
{
    register *rp, *ip;

    rp = fp;
    ip = rp->f_inode;

loop:
    plock(ip);
    if(rp->f_offset[1] == ip->i_size1) {
        if(rp->f_offset[1] != 0) {
            rp->f_offset[1] = 0;
            ip->i_size1 = 0;
            if(ip->i_mode&IWRITE) {
                ip->i_mode =& ~IWRITE;
                wakeup(ip+1);
            }
        }
        prele(ip);
        if(ip->i_count < 2)
            return;
```

```
23         ip->i_mode =| IREAD;
24         sleep(ip+2, PPIPE);
25         goto loop;
26     }
27     u.u_offset[0] = 0;
28     u.u_offset[1] = rp->f_offset[1];
29     readi(ip);
30     rp->f_offset[1] = u.u_offset[1];
31     prele(ip);
32 }
```

10 加锁 inode[] 元素。

11 如果已读取了管道中的全部数据，则进入睡眠状态以便发送方传送更多的数据。

12 读取了若干数据后的处理。

13~14 对已读取的数据，将文件偏移量和文件长度清 0。

15~18 如果设置了 IWRITE 标志位，则将其清 0，唤醒发送方的进程。

20 解锁 inode[] 元素。

21~22 当发送方的进程关闭了管道时，inode[] 元素的参照计数器的值将小于 2，此时结束管道处理。

23~25 为 inode[] 元素设置 IREAD 标志位并进入睡眠状态。在管道处理中，IREAD 标志位表示管道已空，数据的接收方正在等待更多的数据。当接收方被发送方的进程唤醒后，将返回 loop 继续读取处理。

27~29 设定偏移量，然后执行 readi() 读取数据。

30 当数据读取完毕后更新文件偏移量。

31 解锁 inode[] 元素。

plock()

plock() 用于对 inode[] 元素进行加锁处理（表 11-3，代码清单 11-7 ）。该函数为 inode[] 元素设置 ILOCK 标志位。如果已设置了该标志位，则为 inode[] 元素设置 IWANT 标志位并进入睡眠状态。

表 **11-3　plock()** 的参数

参数	含义
ip	inode[] 元素

代码清单 **11-7　plock()**（**ken/pipe.c)**

```
plock(ip)
int *ip;
{
    register *rp;

    rp = ip;
```

```
    while(rp->i_flag&ILOCK) {
        rp->i_flag =| IWANT;
        sleep(rp, PPIPE);
    }
    rp->i_flag =| ILOCK;
}
```

prele()

`prele()` 用于清除 `inode[]` 元素的 `ILOCK` 标志位并为其解锁（表 11-4，代码清单 11-8 ）。当设置了 `IWANT` 标志位时则将其清除，并唤醒其他正在等待该 `inode[]` 元素的锁被释放的进程。

表 **11-4　prele()** 的参数

参数	含义
ip	inode[] 元素

代码清单 **11-8　prele()**（**ken/pipe.c)**

```
prele(ip)
int *ip;
{
    register *rp;

    rp = ip;
    rp->i_flag =& ~ILOCK;
    if(rp->i_flag&IWANT) {
        rp->i_flag =& ~IWANT;
        wakeup(rp);
    }
}
```

11.4　结束管道通信

closef()

当管道通信结束后，用户程序对通过系统调用 `pipe` 取得的文件描述符，像对待一般文件一样执行系统调用 `close` 来关闭管道。

如果设置了 `file[]` 元素的 `FPIPE` 标志位，则在 `closef()` 中为 `file[]` 元素指向的 `inode[]` 元素清除 `IREAD` 和 `IWRITE` 标志位，并唤醒发送方和接收方的进程。通信的双方此时都处于睡眠状态，等待对方进行读写，如果对方的进程就这样结束处理，睡眠中的进程就无法被唤醒，因此要在此处进行唤醒处理（代码清单 11-9 ）。

此后的处理流程与关闭一般文件相同。

代码清单 **11-9　closef()** 的相关处理（**ken/fio.c)**

```
    if(rfp->f_flag&FPIPE) {
        ip = rfp->f_inode;
        ip->i_mode =& ~(IREAD|IWRITE);
```

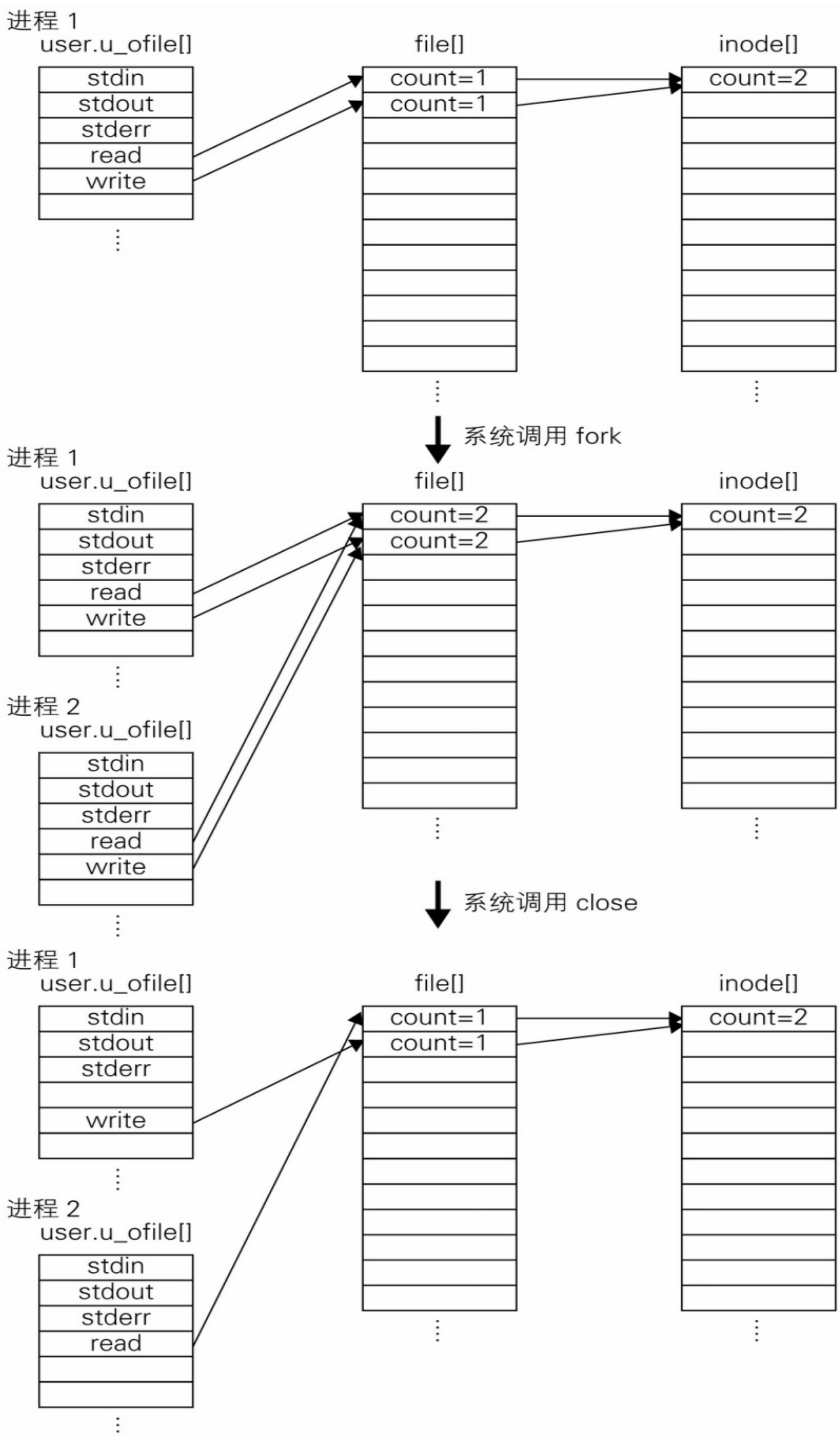
```
        wakeup(ip+1);
        wakeup(ip+2);
    }
    if(rfp->f_count <= 1)
        closei(rfp->f_inode, rfp->f_flag&FWRITE);
    rfp->f_count--;
```

11.5 建立管道通信的流程

建立父子进程间的通信

本节介绍在父子进程间建立管道通信的流程，此处需要用到系统调用 **pipe** 、系统调 用 **fork** 以及系统调用 **close** 。

在系统调用 pipe 执行结束后，再执行系统调用 fork ，父进程的 user.u_ofile[] 的内容将被复制到新生成的子进程中。如果利用系统调用 close 关闭与父进程的 read 使用的文件描述符，和与子进程的 write 使用的文件描述符相对应的 user.u_ofile[] 元素，就形成了一条从父到子传送数据的管道（图 11-5 ）。



进程 1
user.u_ofile[]
stdin
stdout
stderr
read
write
file[]
count=1
count=1
inode[]
count=2
系统调用 fork
进程 1
user.u_ofile[]
stdin
stdout
stderr
read
write
file[]
count=2
count=2
inode[]
count=2
进程 2
user.u_ofile[]
stdin
stdout
stderr
read
write
系统调用 close
进程 1
user.u_ofile[]
stdin
stdout
stderr
write
file[]
count=1
count=1
inode[]
count=2
进程 2
user.u_ofile[]
stdin
stdout
stderr
read

图 **11-5**　**pipe**、**fork**、**close**

再加上系统调用 **dup**，就可以实现在 Shell 中使用的通过“|”连接的管道通信（代码清单 11-10 ）。此例中，ls 向标准输出写入数据，而 cat 从标准输入中取得数据，通过这种简单的方式即可实现数据的交换。

代码清单 **11-10**　管道通信

```
1 % ls | cat
```

进程的 user.u_oflie[0] 、user.u_oflie[1] 和 user.u_oflie[2] 中分别保存了与标准输入（stdin ）、标准输出（stdout ）和标准错误输出（stderr ）相对应的文件描述符。程序在对标准输入、标准输出和标准错误输出进行读写时，需要使用系统调用 read 和 write 。上述的管道通信通过将标准输入和标准输出改为管道得以实现。

在执行系统调用 pipe 和 fork 之后，通过系统调用 close 关闭父进程的标准输出（=user.u_ofile[1] ）和子进程的标准输入（=user.u_ofile[0] ）。此后，执行系统调用 dup 复制与 write 和 read 使用的文件描述符相对应的 user.u_ofile[] 元素。因为系统调用 **dup** 将文件描述符复制到 **user.u_ofile[]** 中排在最前面的空闲元素处，所以 write 和 read 使用的文件描述符将被复制到刚才关闭的 user.u_ofile[] 元素的位置。

最后，关闭由系统调用 pipe 生成的与 write 和 read 使用的文件描述符相对应的 user.u_ofile[] 元素，结束建立管道通信的处理（图 11-6 ）。

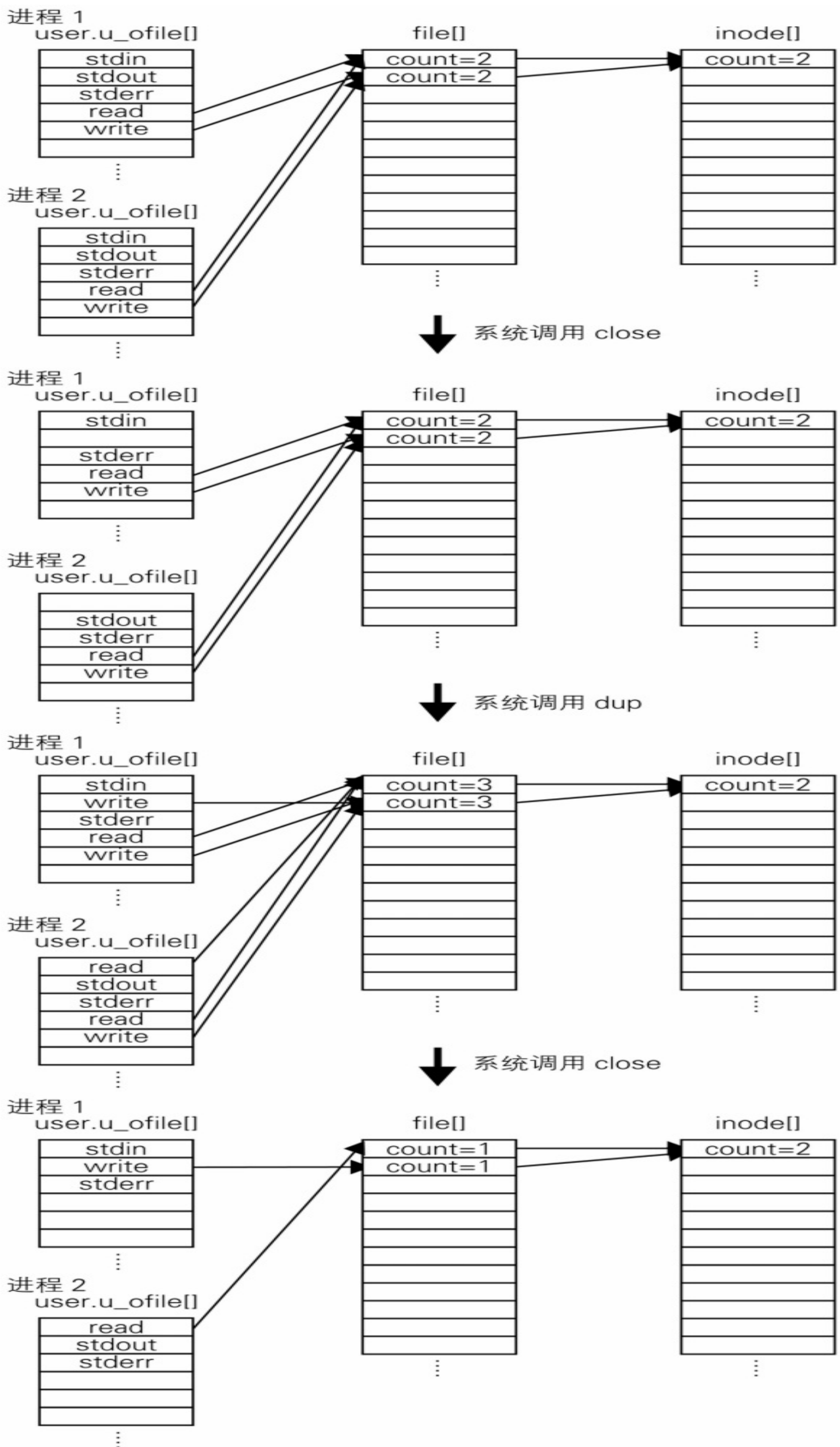

进程 1
user.u_ofile[]
stdin
stdout
stderr
read
write
file[]
count=2
count=2
inode[]
count=2
进程 2
user.u_ofile[]
stdin
stdout
stderr
read
write
系统调用 close
进程 1
user.u_ofile[]
stdin
stderr
read
write
file[]
count=2
count=2
inode[]
count=2
进程 2
user.u_ofile[]
stdout
stderr
read
write
系统调用 dup
进程 1
user.u_ofile[]
stdin
write
stderr
read
write
file[]
count=3
count=3
inode[]
count=2
进程 2
user.u_ofile[]
read
stdout
stderr
read
write
系统调用 close
进程 1
user.u_ofile[]
stdin
write
stderr
file[]
count=1
count=1
inode[]
count=2
进程 2
user.u_ofile[]
read
stdout
stderr

图 **11-6　pipe、fork、close、dup、close**

系统调用 **dup**

`dup()` 为系统调用 `dup` 的处理函数（表 11-5，代码清单 11-11 ）。系统调用 `dup` 将与用户程序指定的文件描述符相对应的 `user.u_ofile[]` 元素复制到 `user.u_ofile[]` 中排在最前面的空闲元素处（图 11-7 ）。

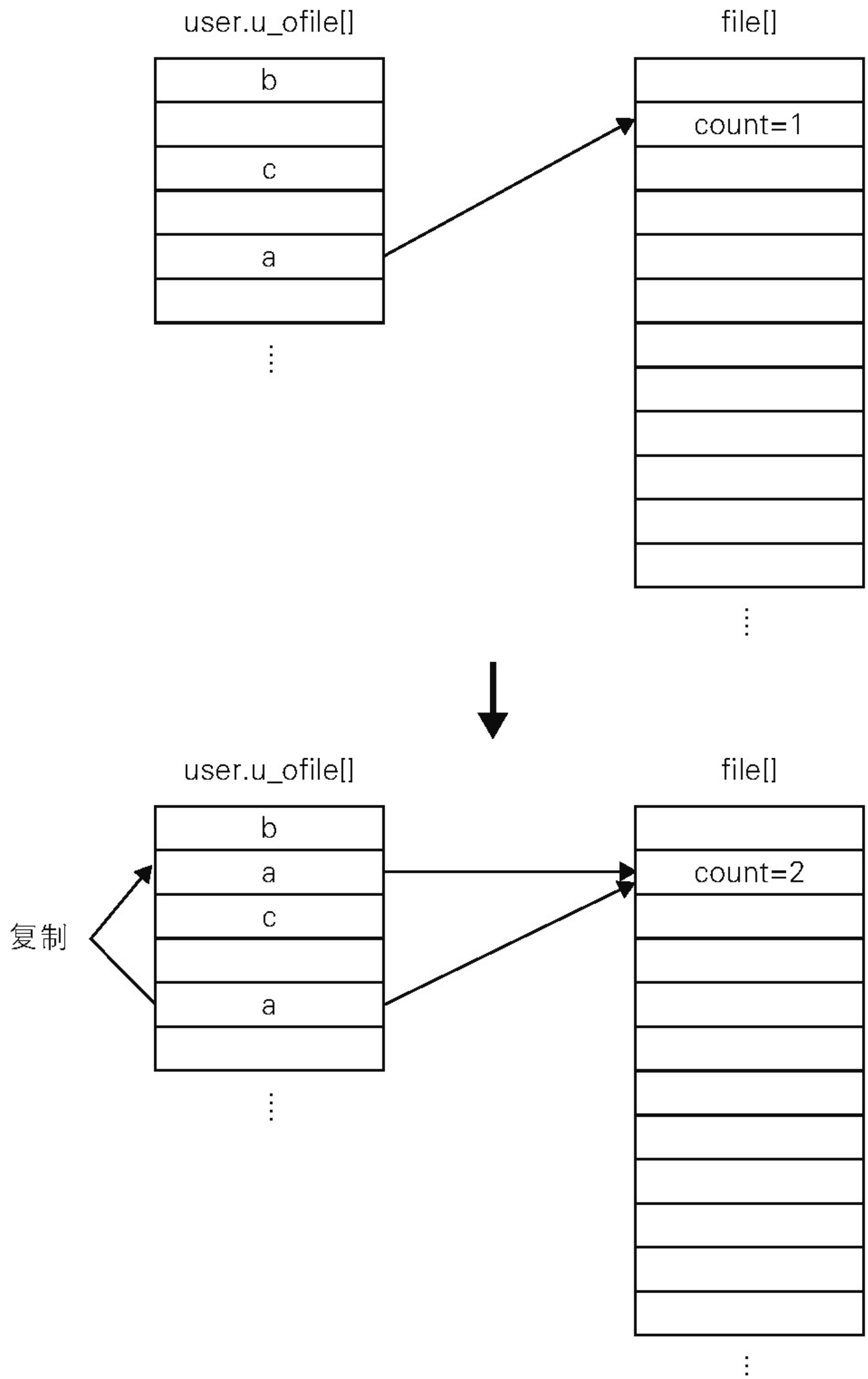


图 **11-7**　系统调用 **dup**

表 **11-5**　系统调用 **dup** 的参数

参数	含义
r0	与准备复制的 u.u_ofile[] 元素相对应的文件描述符

代码清单 **11-11　dup()**（**ken/sys3.c)**

```
 1 dup()
 2 {
 3     register i, *fp;
 4
 5     fp = getf(u.u_ar0[R0]);
 6     if(fp == NULL)
 7         return;
 8     if ((i = ufalloc()) < 0)
 9         return;
10     u.u_ofile[i] = fp;
11     fp->f_count++;
12 }
```

5~7　取得与用户程序通过参数指定的文件描述符相对应的 file[] 元素。

8~9　取得 u.u_ofile[] 中排在最前面的空闲元素。

10~11　将获取的 u.u_ofile[] 元素指向在第5行中获取的 file[] 元素，并递增 file[] 元素的参照计数器的值。

11.6　小结

- 利用管道可以实现父子进程间的通信。
- 发送方进程和接收方进程交替对长度为 4096 字节的管道（文件）进行读写。
- 结合系统调用 pipe 、fork 、close 、dup ，建立管道通信。

第 VI 部分　字符 I/O 系统

字符 I/O 设备以文字为单位处理数据,在行打印机和终端等直接被用户使用的外部设备中经常用到。第 VI 部分主要介绍以下内容。

- 字符设备如何与系统进行数据交换
- 字符设备驱动如何对字符设备进行操作
- 内核如何对终端处理提供支持

通过阅读本部分的内容,读者会了解用户和系统是如何发生关联的。

第 12 章　字符设备

12.1　字符设备驱动

与块设备驱动相同，字符设备也存在相应的设备驱动表。`cdevsw[]` 为字符设备驱动表（代码清单 12-1，表 12-1 ）。

字符设备利用缓冲区处理数据（图 12-1 ）。

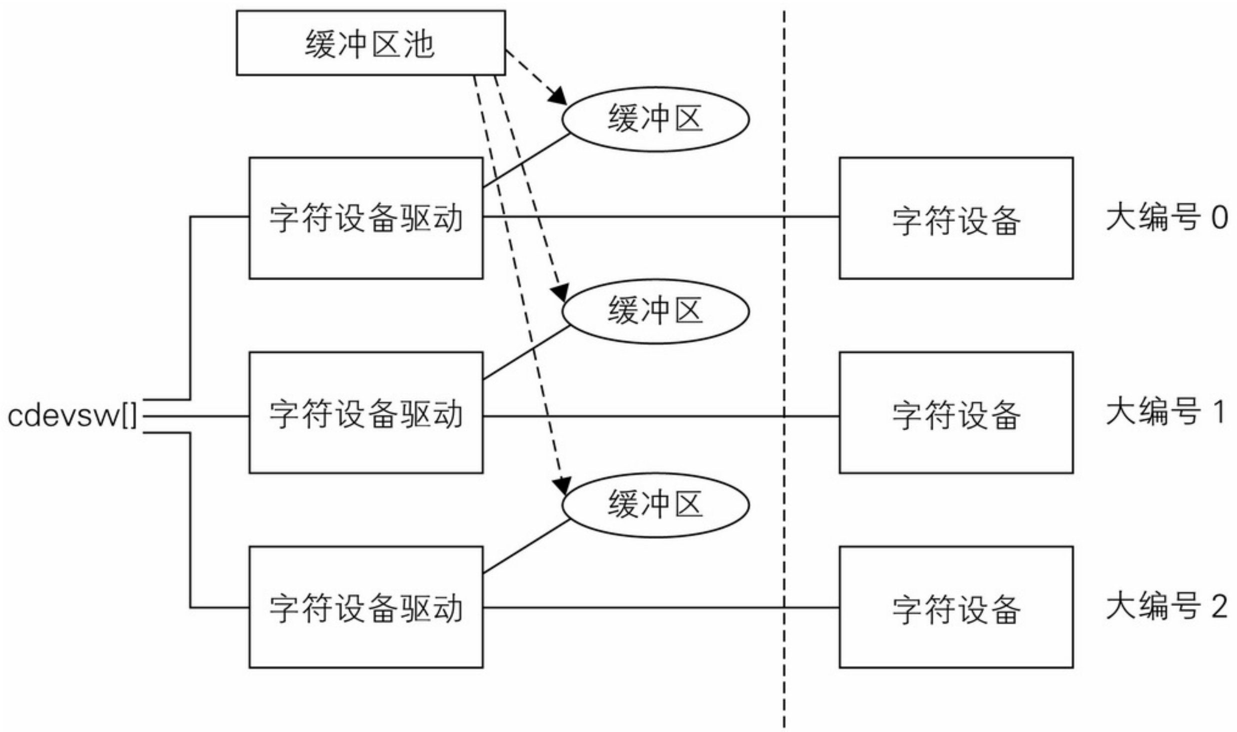


图 **12-1**　字符设备驱动表

代码清单 **12-1**　**cdevsw[]**（**conf.h)**

```
struct    cdevsw
{
    int    (*d_open)();
    int    (*d_close)();
    int    (*d_read)();
    int    (*d_write)();
    int    (*d_sgtty)();
} cdevsw[];
```

表 **12-1**　**cdevsw** 结构体

成员	含义
`(*d_open)()`	指向用于打开设备的函数的指针

(*d_close)()	指向用于关闭设备的函数的指针
(*d_read)()	指向用于读取的函数的指针
(*d_write)()	指向用于写入的函数的指针
(*d_sgtty)()	指向用于设定终端的函数的指针

cdevsw[] 的实体与 bdevsw[] 相同，由 conf.c 设置（代码清单 12-2 ）。系统管理者需要根据系统所使用的字符设备编辑 conf.c ，并对内核进行重新构筑。

代码清单 **12-2**　**cdevsw[]** 的例子（**conf.c)**

```
int    (*cdevsw[])()
{
    &klopen,   &klclose,  &klread,   &klwrite,  &klsgtty,  /* console */
    &pcopen,   &pcclose,  &pcread,   &pcwrite,  &nodev,    /* pc */
    &lpopen,   &lpclose,  &lpread,   &lpwrite,  &nodev,    /* lp */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* dc */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* dh */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* dp */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* dj */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* dn */
    &nulldev,  &nulldev,  &mmread,   &mmwrite,  &nodev,    /* mem */
    &nodev,    &nodev,    &rkread,   &rkwrite,  &nodev,    /* rk */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* rf */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* rp */
    &nodev,    &nodev,    &noddev,   &nodev,    &nodev,    /* tm */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* hs */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* hp */
    &nodev,    &nodev,    &nodev,    &nodev,    &nodev,    /* ht */
    0
};
```

字符设备缓冲区

字符设备驱动利用缓冲区与字符设备进行数据交换。缓冲区的实体为 cblock 结构体的数组 cfree[] （代码清单 12-3，表 12-2 ）。

代码清单 **12-3**　**cblock**（**dmr/tty.c)**

```
struct cblock {
    struct cblock *c_next;
    char info[6];
};
struct cblock cfree[NCLIST];
```

表 **12-2**　**cblock** 结构体

成员	含义
*c_next	指向下一个 cblock 结构体的指针

info[]	数据。长度为 6 字节（文字）

缓冲区由链表进行管理，c_next 为指向下一个 cblock 结构体的指针，info[] 用于容纳数据，每个 cblock 结构体可保存长度为 6 字节（文字）的数据。

每个字符设备驱动都拥有以 clist 结构体（代码清单 12-4，表 12-3 ）为起始元素的缓冲区的链表。未使用的缓冲区由 cfreelist （代码清单 12-5 ）指向的以 cblock 结构体为起始元素的空闲队列进行管理。设备驱动在需要时从空闲队列取得 cblock 结构体，并在使用完毕后返还给空闲队列（图 12-2 ）。

代码清单 **12-4　clist（tty.h)**

```
struct clist
{
    int c_cc;
    int c_cf;
    int c_cl;
};
```

表 **12-3　clist** 结构体

成员	含义
c_cc	缓冲区中的文字总数
c_cf	缓冲区链表的起始位置
c_cl	缓冲区链表的结束位置

代码清单 **12-5　cfreelist（dmr/tty.c)**

```
struct cblock *cfreelist;
```

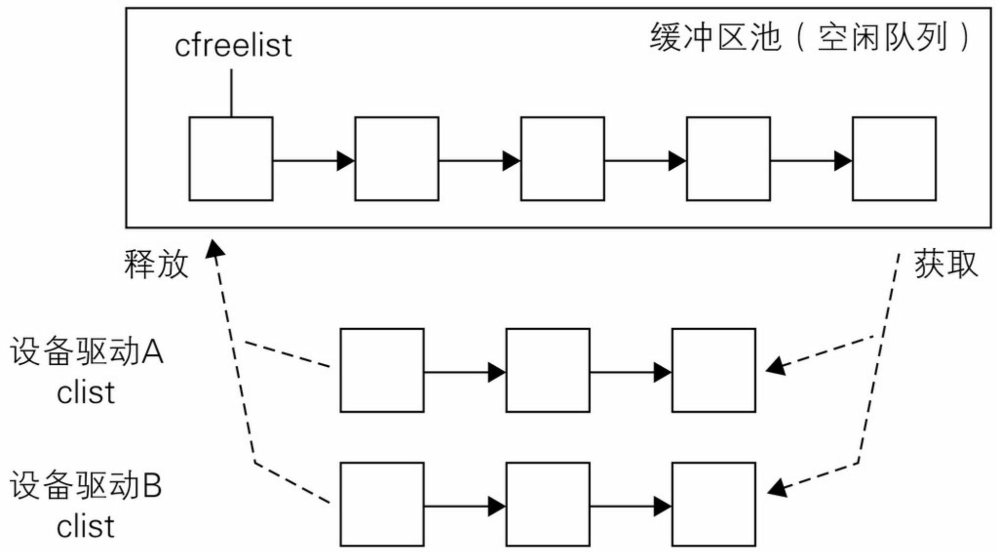


图 **12-2**　缓冲区的分配

对缓冲区的操作

字符设备驱动通过由汇编语言编写的 `getc()` 和 `putc()` 操作缓冲区。

考虑一下某个字符设备驱动将长度为 16 文字（16 字节）的字符串“hellogoodbyehowa”保存在缓冲区时的情况（图 12-3 ）。

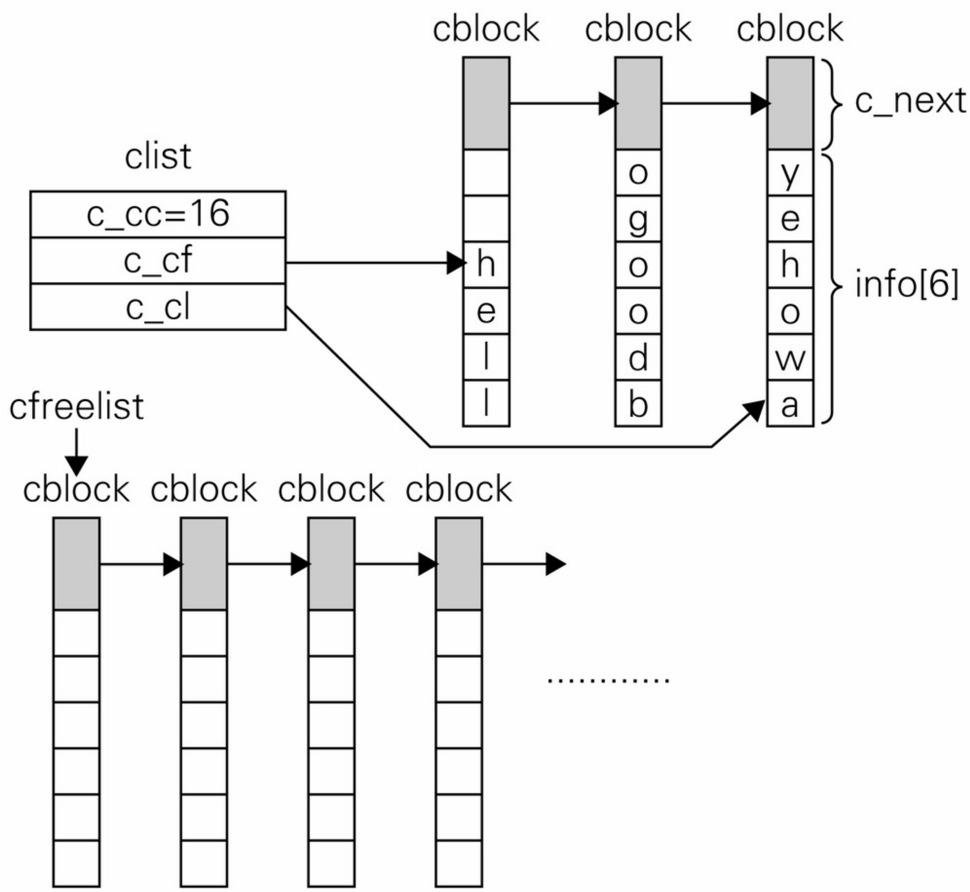


图 **12-3**　缓冲区链表的例子

`getc()` 从缓冲区的起始位置取得一个文字。图 12-4 表示从缓冲区的起始位置取出文字 h 时的情况。

`putc()` 向缓冲区的末尾追加一个文字。图 12-5 表示在图 12-4 的基础上向缓冲区的末尾追加文字 r 时的情况。由于末尾的 `cblock` 结构体的 `info[]` 已被用尽，因此此处需要从空闲队列中取得新的 `cblock` 结构体。

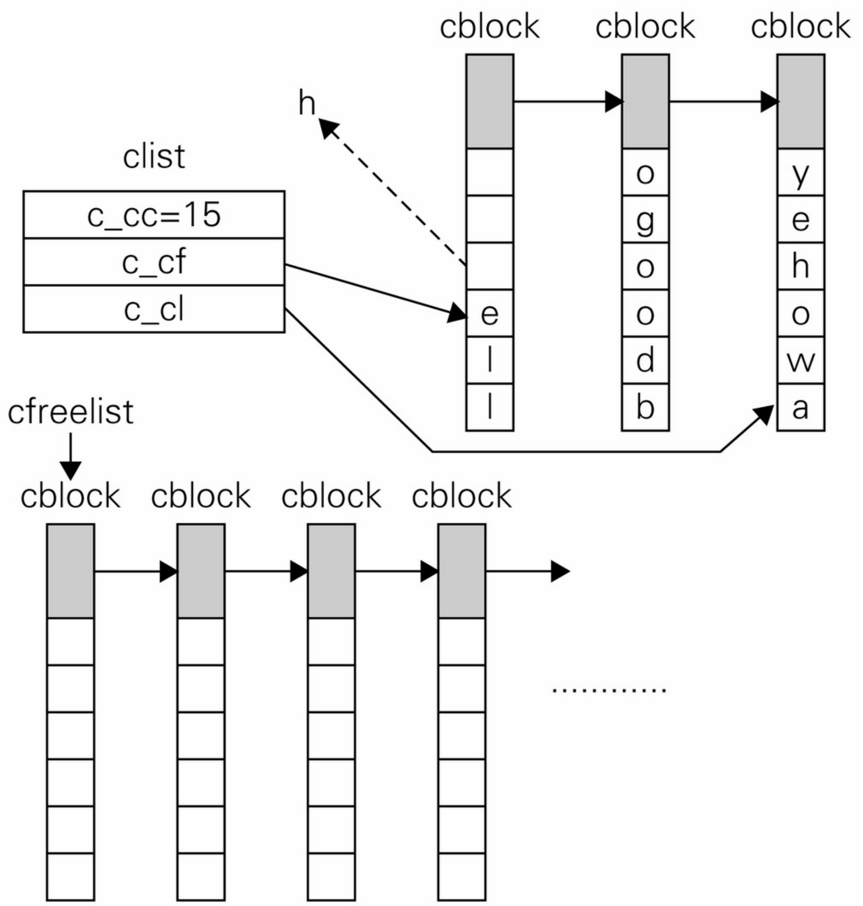

cblock
cblock
cblock
h
clist
c_cc=15
c_cf
c_cl
o
g
o
o
d
b
y
e
h
o
w
a
e
l
l
cfreelist
cblock
cblock
cblock
cblock
....

图 **12-4　getc()**

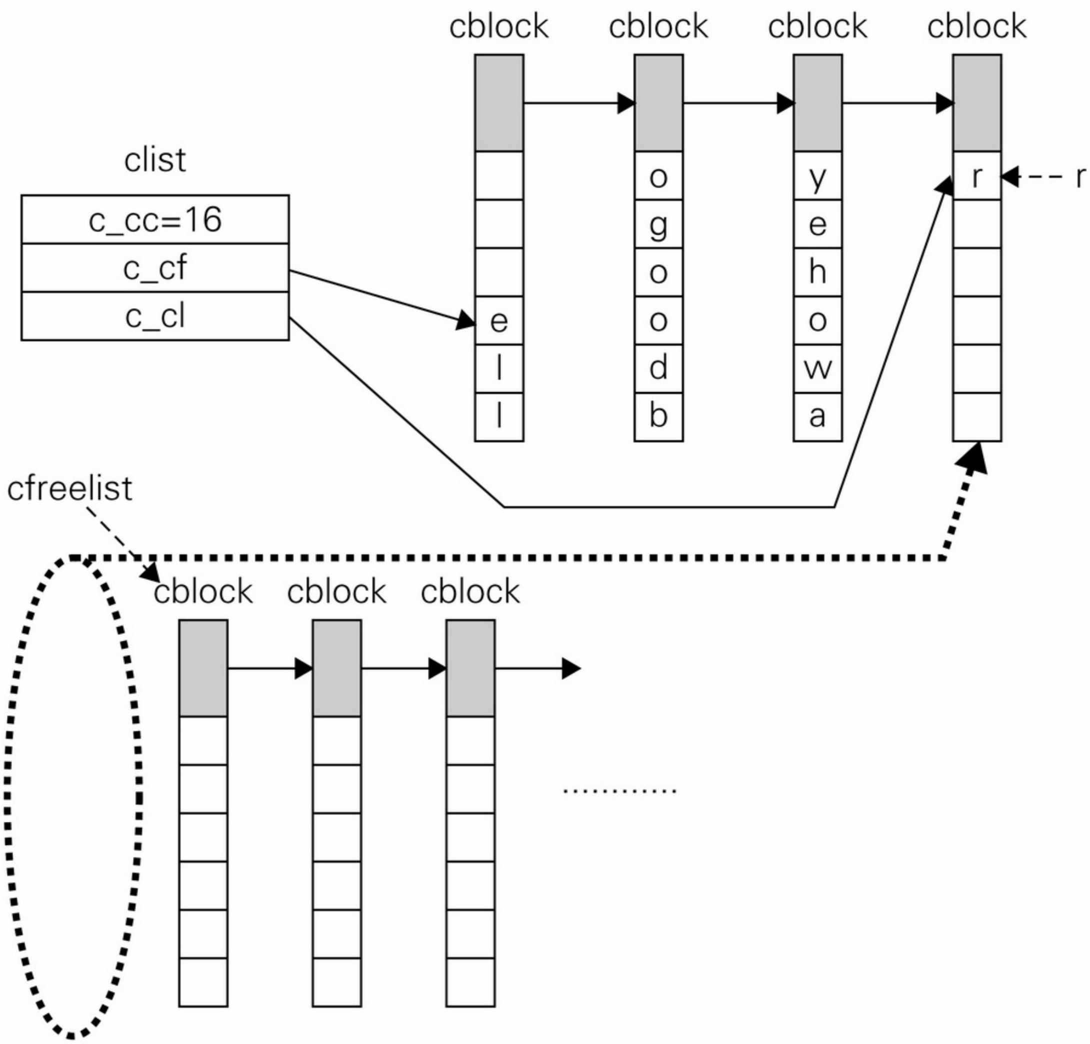


图 **12-5　putc()**

初始化缓冲区池

`cinit()` 是初始化缓冲区池时使用的函数（代码清单 12-6 ）。系统启动时由 `main()` 调用，且仅调用一次。该函数将 `cfree[]` 的元素全部追加至空闲队列，并统计系统中使用的字符设备（的种类）的数量，保存统计结果。

代码清单 **12-6　cinit()**（**dmr/tty.c)**

```
cinit()
{
    register int ccp;
    register struct cblock *cp;
    register struct cdevsw *cdp;

    ccp = cfree;
    for (cp=(ccp+07)&~07; cp <= &cfree[NCLIST-1]; cp++) {
        cp->c_next = cfreelist;
        cfreelist = cp;
    }
    ccp = 0;
    for(cdp = cdevsw; cdp->d_open; cdp++)
```

```
14         ccp++;
15     nchrdev = ccp;
16 }
```

7~11　向空闲队列追加 `cfree[]` 的元素，通过此队列可反向参照 `cfree[]`。由于构成 `cfree[]` 元素的 `cblock` 结构体的长度为 8 字节，因此对 for 语句中的初始化计算公式进行调整，使起始地址为 8 的倍数[1]。

12~15　通过统计在 `cdevsw[]` 中注册的 `cdevsw.d_open` 处理函数的数量，可得到字符设备（驱动）的数量，将统计结果保存在 `nchrdev` 中。

[1] 此处将空闲队列向 8 倍数字节地址对齐，有利于在 `getc` 等函数中读取缓存块中字符时若碰到 8 倍数字节地址，则可以判断此缓存块已经读尽。——审校者注

代码清单 **12-7　nchrdev（conf.h)**

```
int    nchrdev;
```

12.2　LP11 设备驱动

以下以 LP11 的设备驱动为例介绍字符设备驱动。

什么是 LP11

LP11 是由 DEC 公司开发的与 PDP-11 系列相对应的行打印机，用于打印所要求的文字。

LP11 对应 ASCII 码。根据型号的不同使用 96 文字或 64 文字的字符集。64 文字的字符集称为 half ASCII 字符集，只支持大写文字和部分符号。

当输入表 12-4 所示的 ASCII 码时，并不打印文字而是进行相应的处理。

表 **12-4　LP11** 可处理的特殊文字

ASCII 码	含义
012	换行（LF）
014	送纸（FF）
015	回车（CR）。将打印头移动至行的起始位置

LP11拥有两个16位的寄存器。一个是用来处理控制信息的 LP Status 寄存器（表 12-5 ），另一个是用来处理输出文字的 LP Data Buffer 寄存器（表 12-6 ）。当数据存入 LP Data Buffer 寄存器后，会启动 LP11 的打印处理，将 LP Status 寄存器的第 7 比特位设为 0。当打印处理结束后，LP Data Buffer 寄存器的值被清空，LP Status 寄存器的第 7 比特位则会被设定为 1。

表 **12-5　LP Status** 寄存器

15	LP11 的处理过程中发生错误时设置为 1
7	表示 LP11 处于 Ready 状态。当 LP Data Buffer 寄存器中存在数据可传递给 LP11 时设为 0。当 LP11 的处理 结束后设置为 1
6	设置为 1 时表示 LP11 的处理结束，或是发生了错误。此时，通过中断优先级 4 和向量 0200 引发中断

表 **12-6　LP Data Buffer** 寄存器

6-0	打印对象文字的 ASCII 码。当设置了数据之后，开始 LP11 的印刷处理，结束后数据被清 0

LP11设备驱动的功能

系统管理者预先生成名为"/dev/lpn"（n 为小编号）的特殊文件。

LP11 的设备驱动主要进行下述处理（图 12-6 ）。

1. 当程序发出输出数据的请求时，将该数据追加至缓冲区。对换行和制表符等特殊文字进行相应的处理。

2. 将 LP11 无法输出的文字更改为可输出的文字。

3. 将缓冲区的数据传送至 LP11 的寄存器。LP11 的寄存器被写入数据后将引发 LP11 的输出处理。

4. 如果发生了 LP11 的处理终了中断，则继续对 LP11 输出数据。

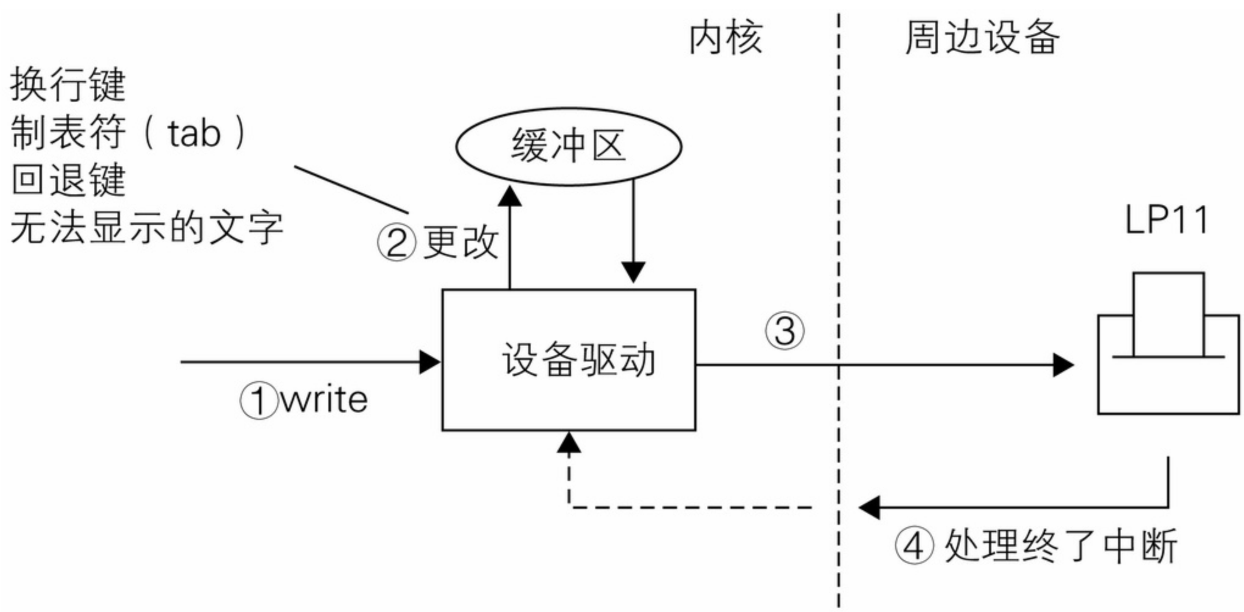


图 **12-6　LP11** 设备驱动

当缩进功能有效时，在行的起始位置首先输出长度为 8 文字的空白。每一行最多输出 80 文字，超过的文字会被忽略。当输出一定行数之后将进行换页处理。

当输入表 12-7 所示的特殊文字时，进行与其相对应的处理。

表 **12-7　ASCII** 码的特殊文字

特殊码	含义
010	删除此前的 1 个文字

011（\t）	输出空白，将打印头移至以 8 文字为单位时的下一个位置
012（\n）	向行打印机输出“FF”换行
014	向行打印机输出“PF”送纸
015（\r）	将打印头移至行的起始位置。当缩进功能有效时输出长度为 8 文字的空白

对使用 half ASCII 字符集的型号而言，首先在设备驱动中将小写文字变为大写文字，然后再对设备进行输出处理。此外，图 12-8 所示的符号，将其变为可使用的符号并叠加表示取消的文字“-”。

表 **12-8**　需要更改的符号

更改前	更改后
{	(-
}	-)
`	'-
|	!-
~	^-

LP11 设备驱动中定义了名为 `lp11` 的结构体（代码清单 12-8，表 12-9）。该结构体位于 LP11 设备驱动使用的缓冲区的头部，也用于管理 LP11 的状态信息和打印头的位置信息。

代码清单 **12-8**　**lp11**（**dmr/lp.c)**

```
struct {
  int cc;
  int cf;
  int cl;
  int flag;
  int mcc;
  int ccc;
  int mlc;
} lp11;

#define    CAP     01
#define    EJECT   02
#define    OPEN    04
#define    IND     010
```

表 **12-9**　**lp11** 结构体

成员	含义
`cc`	缓冲区中的文字总数。用于 `putc()` 和 `getc()` 中

cf	缓冲区链表的头部。用于 putc() 和 getc() 中
cl	缓冲区链表的尾部。用于 putc() 和 getc() 中
flag	状态、控制信息。在打开设备时根据 LP11 的型号和模式进行相应的设定（参见表 12-10）
mcc	当前处理行中打印头的物理位置
ccc	当前处理行中打印头的逻辑位置
mlc	页中的处理行数

表 **12-10　lp11** 中的标志位

标志位	含义
CAP	只处理大写文字（half ASCII 字符集）
EJECT	具有排纸功能
OPEN	设备已打开
IND	使用缩进功能

输出文字后，递增 `lp11.mcc` 和 `lp11.ccc` 的值记录打印头在当前行中的位置。当需要输出空格或因为缩进需要输出空白时，并不是每次都将其输出至打印机，而是只递增 `lp11.ccc` 的值。当需要输出空白以外的文字时，首先输出长度为 `lp11.mcc` 与 `lp11.ccc` 差值的空白，再输出该文字。此处，将 `lp11.mcc` 称为“打印头的物理位置”，将 `lp11.ccc` 称为“打印头的逻辑位置”。

lpopen()

`lpopen()` 用于打开 LP11 的处理（表 12-11，代码清单 12-9 ）。对 LP11 的特殊文件执行系统调用 open 后，在 open 的处理函数调用的 `openi()` 中执行在 `cdevsw[].d_open()` 中注册的 `lpopen()` 。

`lpopen()` 首先检查设备是否已被打开，以及是否处于出错状态。然后设定 `lp11.flag` 并将 LP Status 寄存器的第 6 比特位设为 1。最后让 LP11 进行送纸的操作。

表 **12-11　lpopen()** 的参数

参数	含义
dev	在 lpopen() 中未使用
flag	在 lpopen() 中未使用

代码清单 **12-9　lpopen()**（**dmr/lp.c)**

```
1 lpopen(dev, flag)
2 {
3
4     if(lp11.flag & OPEN || LPADDR->lpsr < 0) {
5         u.u_error = EIO;
6         return;
7     }
8     lp11.flag =| (IND|EJECT|OPEN);
9     LPADDR->lpsr =| IENABLE;
10    lpcanon(FORM);
11 }
```

4~7　如果 LP11 已打开，或是处于出错状态，将错误代码赋予 `u.u_error` 并返回。LP Status 寄存器的第 15 比特位为错误标志位，当此标志位被设置为 1 时，`signed int` 型的变量为负值。

`LPADDR` 表示 LP11 寄存器映射的地址。`lpsr` 为用于操作 LP11 寄存器的结构体的成员变量（代码清单 12-10 ）。

代码清单 **12-10　LPADDR**（**dmr/ken.c)**

```
#define LPADDR 0177514

struct {
  int lpsr;
  int lpbuf;
};
```

8~10　对 LP11 进行初始化处理。首先设定 `lp11.flag` 和 LP Status 寄存器，然后让 LP11 进行送纸操作，再重置打印头的位置。`FORM` 表示 ASCII 码中的送纸字符（FF）（代码清单 12-11）。

代码清单 **12-11　FORM**（**dmr/lp.c)**

```
#define    FORM    014
```

lpwrite()

`lpwrite()` 使 LP11 打印字符串（代码清单 12-12 ）。对 LP11 的特殊文件执行系统调用 `write` 后，在 `write` 的处理函数调用的 `writei()` 中将执行在 `cdevsw[].d_write()` 中注册的 `lpwrite()` 。

执行 `cpass()` ，从用户空间读取 1 字节的数据，调用 `lpcanon()` 使 LP11 输出该数据。重复此过程直至所有数据都被输出。

代码清单 **12-12　lpwrite()**（**dmr/lp.c)**

```
lpwrite()
{
    register int c;

    while ((c=cpass())>=0)
        lpcanon(c);
}
```

lpcanon()

`lpcanon()` 执行与参数指定的长度为 1 字节的文字相对应的处理（表 12-12，代码清单 12-13 ）。如果为一般文字，则执行 `lpoutput()` 使 LP11 打印该文字。如果为特殊文字，则进行与其对应的处理。

表 **12-12　lpcanon()** 的参数

参数	含义
c	准备输出的 ASCII 码

代码清单 **12-13　lpcanon()**（**dmr/lp.c)**

```
lpcanon(c)
{
    register c1, c2;

    c1 = c;
    if(lp11.flag&CAP) {
        if(c1>='a' && c1<='z')
            c1 =+ 'A'-'a'; else
        switch(c1) {

        case '{':
        c2 = '(';
            goto esc;

        case '}':
            c2 = ')';
            goto esc;

        case '`':
            c2 = '\'';
            goto esc;

        case '|':
            c2 = '!';
            goto esc;

        case '~':
            c2 = '^';

        esc:
            lpcanon(c2);
            lp11.ccc--;
            c1 = '-';
        }
    }

    switch(c1) {

    case '\t':
        lp11.ccc = (lp11.ccc+8) & ~7;
        return;

    case FORM:
    case '\n':
        if((lp11.flag&EJECT) == 0 ||
           lp11.mcc!=0 || lp11.mlc!=0) {
            lp11.mcc = 0;
            lp11.mlc++;
            if(lp11.mlc >= EJLINE && lp11.flag&EJECT)
                c1 = FORM;
```

```
51             lpoutput(c1);
52             if(c1 == FORM)
53                 lp11.mlc = 0;
54         }
55
56     case '\r':
57         lp11.ccc = 0;
58         if(lp11.flag&IND)
59             lp11.ccc = 8;
60         return;
61
62     case 010:
63         if(lp11.ccc > 0)
64             lp11.ccc--;
65         return;
66
67     case ' ':
68         lp11.ccc++;
69         return;
70
71     default:
72         if(lp11.ccc < lp11.mcc) {
73             lpoutput('\r');
74             lp11.mcc = 0;
75         }
76         if(lp11.ccc < MAXCOL) {
77             while(lp11.ccc > lp11.mcc) {
78                 lpoutput(' ');
79                 lp11.mcc++;
80             }
81             lpoutput(c1);
82             lp11.mcc++;
83         }
84         lp11.ccc++;
85     }
86 }
```

6~35 使用 half ASCII 码字符集时的处理。如果是小写文字，则将其变为大写文字。如果是“{”等特殊文字，将其变为可输出的文字后，将 `c1` 设定为表示取消的文字“-”，然后继续输出其他文字。

39~41 对制表符（`\t`）的处理。调整 `lp11.ccc` 使其成为 8 的倍数。

43~54 对送纸和换行符（`\n`）的处理。如果打印机没有排纸功能，或当前页上已有打印的文字，则将 `lp11.mcc` 设为 0 并递增 `lp11.mlc` 的值。如果每页可打印的行数较多（代码清单 12-14），且打印机具有排纸功能，则进行送纸，否则的话，则输出原有的送纸或换行符。送纸后，将表示行数的变量 `lp11.mlc` 清 0。

代码清单 **12-14 EJLINE**（**dmr/lp.c**）

```
1 #define    EJLINE    60
```

56~60 对回车符（`\r`）的处理。将 `lp11.ccc` 设为 0。如果设置了缩进标志位，则将 `lp11.ccc` 设定为 8。

62~65 对 Back Space 的处理。如果 `lp11.ccc` 的值大于等于 1，则递减该值。

67~69 对空白的处理。递增 `lp11.ccc` 的值。

71~83 如果是一般文字，则进行输出。当 `lp11.ccc` 小于 `lp11.mcc` 时进行回车处理。如果

lp11.ccc 小于 MAXCOL （每行可输出的最大文字数。代码清单 12-15），那么在输出文字后，递增 lp11.mcc 的值。如果 lp11.ccc 大于 lp11.mcc 时，输出长度为 lp11.ccc 与 lp11.mcc 差值的空白，调整打印头的物理位置。

代码清单 **12-15　MAXCOL**（**dmr/lp.c)**

```
#define    MAXCOL    80
```

84　递增 lp11.ccc 的值。虽然无需打印，但也需调整打印头的逻辑位置。

lpoutput()

lpoutput() 使 LP11 打印 1 个文字（表 12-13，代码清单 12-17 ）。调用 putc() 向缓冲区传送数据后，执行 lpstart() 启动 LP11 的处理。但是，当缓冲区内的数据（等待输出的字符串）较长时（LPHWAT 大于等于 100 ），为了避免出现缓冲区池容量不足的情况，进程进入睡眠状态。随着使用缓冲区内的数据（LPLWAT =50 ），进程通过 lpint() 被唤醒（代码清单 12-16 ）。

代码清单 **12-16　LPLWAT** 与 **LPHWAT**（**dmr/lp.c)**

```
#define    LPLWAT    50
#define    LPHWAT    100
```

表 **12-13　lpoutput()** 的参数

参数	含义
c	准备输出的 ASCII 码

代码清单 **12-17　lpoutout()**（**dmr/lp.c)**

```
1 lpoutput(c)
2 {
3     if (lp11.cc >= LPHWAT)
4         sleep(&lp11, LPPRI);
5     putc(c, &lp11);
6     spl4();
7     lpstart();
8     spl0();
9 }
```

3~4　如果缓冲区内的数据长度大于等于 LPHWAT （100）个文字，则进入睡眠状态直至数据被使用。

5　执行 putc() ，向缓冲区追加 1 字节的数据。

6　将处理器优先级提升至 4，以防止在 LP11 的输出过程中发生由 LP11 引发的中断。

7　执行 lpstart() 启动 LP11 的输出处理。

8 当 LP11 的处理结束后，将处理器优先级重置为 0。

lpstart()

`lpstart()` 用于启动 LP11 的处理（代码清单 12-18 ）。如果 LP11 处于等待输入的状态，且缓冲区内存在数据时，从缓冲区取出长度为 1 字节的数据，将其赋予 LP Data Buffer 寄存器。该寄存器被设定数据后，LP11 将自动进行输出文字的处理。

代码清单 **12-18　lpstart()**（**dmr/lp.c)**

```
lpstart()
{
    register int c;

    while (LPADDR->lpsr&DONE && (c = getc(&lp11)) >= 0)
        LPADDR->lpbuf = c;
}
```

lpint()

LP11 在结束输出文字的处理后将引发中断，`lpint()` 为该中断的处理函数（代码清单 12-19 ）。

执行 `lpstart()`，如果缓冲区内存在残留数据则继续印刷处理。当缓冲区内的数据数量（等待输出的字符串数）为 0 或 `LPLWAT` （=50 ）时，唤醒因等待数据输出而进入睡眠状态的进程。

代码清单 **12-19　lpint()**（**dmr/lp.c)**

```
lpint()
{
    register int c;

    lpstart();
    if (lp11.cc == LPLWAT || lp11.cc == 0)
        wakeup(&lp11);
}
```

lpclose()

`lpclose()` 用于关闭 LP11（代码清单 12-20 ）。对 LP11 的特殊文件执行系统调用 `close` 后，将在 `close` 的处理函数调用的 `closei()` 中执行在 `cdevsw[].d_close()` 中注册的 `lpclose()` 。`lpclose()` 使 LP11 进行送纸处理，并将 `lp11.flag` 清 0。

表 **12-14　lpclose()** 的参数

参数	含义
`dev`	在 `lpclose()` 中未使用
`flag`	在 `lpclose()` 中未使用

代码清单 **12-20　lpclose()**（**dmr/lp.c)**

```
lpclose(dev, flag)
{
    lpcanon(FORM);
    lp11.flag = 0;
}
```

12.3 小结

- 字符设备驱动表为 cdevsw[] 。
- 字符设备驱动以字节为单位向缓冲区传送数据，然后再通过缓冲区将数据传送给设备。

第 13 章　电传终端

13.1　什么是电传终端

电传终端是用于在相离的两点间进行通信的输入输出装置 。UNIX V6 内核支持电传终端的处理，用户通过电传终端可以以对话的形式操作系统。PDP-11 系列附带了由 Teletype 公司制造的 ASR-33 型电传终端（图 13-1 ）。

图片：Marcin Wichary, GFDL

图 **13-1　ASR-33**[1]

[1] http://ja.wikipedia.org/wiki/ASR-33

电传终端由输入装置和输出装置组成。如果将输入装置看作是现在的键盘，输出装置看作显示器（或打印机）的话，应该更容易理解。

电传终端采用 ASCII 码。输入数据以行为单位进行处理。如果只是输入字符串，那么输入的数据只会保存在缓冲区内，不会被传送到系统（用户空间）。只有当输入换行后，缓冲区内的数据才会传送到系统。

电传终端的接口

在连接 PDP11/40 与电传终端时，需要使用电传终端接口 。该接口与 Unibus 并行通信，与电传终端进行串行通信（图 13-2 ）。

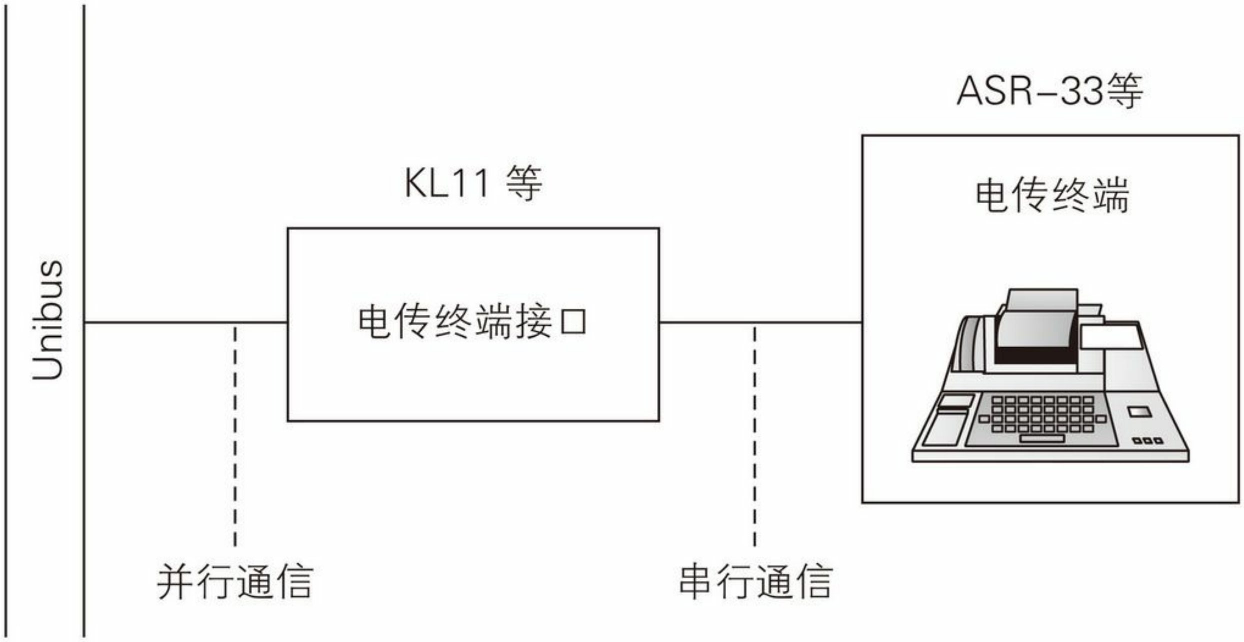


图 **13-2**　电传终端接#

电传终端接口有 KL11、DC11、DH11 等多个种类。设备驱动通过操作接口拥有的寄存器控制终端。根据接口种类的不同，其规格也有所不同，需要分别提供相应的设备驱动。与接口种类的差异无关的共用处理定义在名为 `tty.c` 的文件中。

为了运行系统操作台（console），必须将 1 台电传终端与 KL11 相连。此外，也可让系统连接多个接口与终端，使多名用户可以同时使用系统（图 13-3 ）。

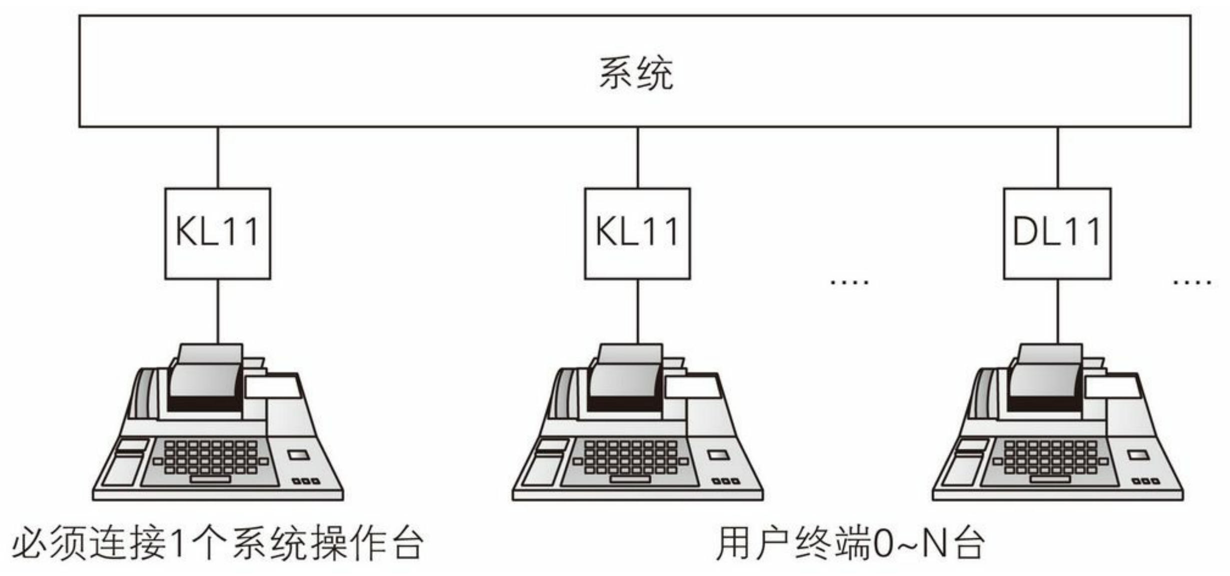


图 **13-3**　多台终端

特殊文件

系统管理员需要生成名为 /dev/ttyX（X 为任意文字）的特殊文件。此外，各个终端的设定数据保存在名为 /etc/ttys 的文件中。

系统启动时，生成与各终端相对应的进程，并等待用户的连接与登录。此时确定了标准输入（`user.u_ofile[0]`）、标准输出（`user.u_ofile[1]`）和标准错误输出（`user.u_ofile[2]`）与各终端的对应关系（严格来讲，标准错误输出在用户登录时才被打开）。

tty 结构体

终端拥有属于自己的 `tty` 结构体（代码清单 13-1，表 13-1）。`tty` 结构体管理终端的设定和控制信息，可以以终端为单位进行设定。

终端的 **`tty`** 结构体与 **`proc.p_ttyp`** 相关联,进程可以以此管理负责控制自己运行的终端。比如，通过此处的关联信息，终端可对自己控制的所有进程发送信号。

代码清单 **13-1**　**tty** 结构体（**tty.h)**

```
struct tty
{
    struct    clist t_rawq;
    struct    clist t_canq;
    struct    clist t_outq;
    int       t_flags;
    int       *t_addr;
    char      t_delct;
    char      t_col;
    char      t_erase;
    char      t_kill;
    char      t_state;
    char      t_char;
    int       t_speeds;
    int       t_dev;
};

/* modes */
#define    HUPCL      01
#define    XTABS      02
#define    LCASE      04
#define    ECHO       010
#define    CRMOD      020
#define    RAW        040
#define    ODDP       0100
#define    EVENP      0200
#define    NLDELAY    001400
#define    TBDELAY    006000
#define    CRDELAY    030000
#define    VTDELAY    040000

/* Internal state bits */
#define    TIMEOUT    01
#define    WOPEN      02
#define    ISOPEN     04
#define    SSTART     010
#define    CARR_ON    020
#define    BUSY       040
#define    ASLEEP     0100
};
```

表 **13-1**　**tty** 结构体

成员	含义
`t_rawq`	队列。用于管理从终端输入的数据

t_canq	队列。用于管理由输入数据经过适当更改后的数据
t_outq	队列。用于管理向终端输出的数据
t_flags	标志变量。将作为系统调用 stty 和 gtty 的操作对象（参照表 13-2）
*t_addr	终端接口寄存器的基地址
t_delct	t_rawq 中分隔符的计数器
t_col	终端打印头的水平方向的位置。用于计算输出延迟处理时间
t_erase	被删除的 1 个文字。将作为系统调用 stty 和 gtty 的操作对象
t_kill	被删除的 1 行文字。将作为系统调用 stty 和 gtty 的操作对象
t_state	状态（参照表 13-3）
t_char	临时区域
t_speeds	终端处理速度。将作为系统调用 stty 和 gtty 的操作对象
t_dev	终端的设备编号

表 **13-2　tty** 的标志位

状态	含义
HUPCL	关闭处理时切断连接
XTABS	XTABS 模式。输出"\t"时，使打印头输出空白直至已输出字符的长度（包括空白）为 8 字节的倍数。并非让终端直接处理制表符，而是在设备驱动端模拟再现制表符
LCASE	使终端只处理大写文字
ECHO	ECHO 模式。输入到终端的文字也同时输出至终端
CRMOD	CR 模式。将输入的 CR（回车：将打印头移至行头）变为 LF（换行）。将输出的 LF 变为 CR 和 LF
RAW	RAW 模式。不进行删除 1 文字、删除 1 行，或通过"\"进行的转义（escape）处理，将输入的字符串直接传递给系统
ODDP	使用奇校验
EVENP	使用偶校验

NLDELAY	对应表示 NL 延迟时间的比特位。通过系统调用 stty 从多个类型中选择
TBDELAY	对应表示 TAB 延迟时间的比特位。通过系统调用 stty 从多个类型中选择
CRDELAY	对应表示 CR 延迟时间的比特位。通过系统调用 stty 从多个类型中选择
VTDELAY	对应表示 VT 延迟时间的比特位。通过系统调用 stty 从多个类型中选择

表 **13-3**　**tty** 状态的标志位

标志位	含义
TIMEOUT	输出延迟处理中
WOPEN	等待打开设备的处理结束
ISOPEN	已打开
SSTART	为 t_addr 设定特殊的启动处理。只被一部分终端接口采用
CARR_ON	已打开
BUSY	输出处理中
ASLEEP	等待使用 t_outq 的数据

`tty` 结构体拥有 **3** 个缓冲区队列（缓冲区链表）。从终端输入的数据由队列 `tty.t_rawq` 管理，对输入数据进行适当变化后的数据由 `tty.t_canq` 管理，而输出数据由队列 `tty.t_outq` 管理（图 13-4 ）。

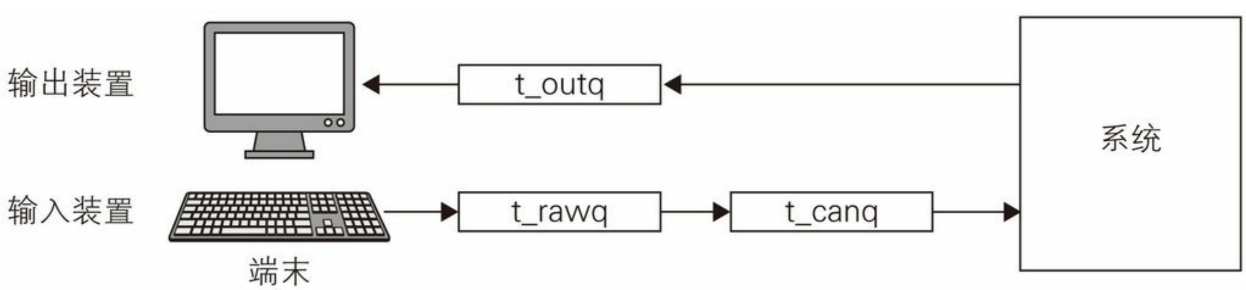


图 **13-4**　**3** 个缓冲区

如前所述，从终端输入的数据以行为单位进行处理。如果输入了换行符，`tty.t_rawq` 中将追加 1 个与其对应的分隔符（=0377 ）。`tty.t_rawq` 中分隔符的数量，即处理单位的数量，由 `tty.t_delct` 管理。

终端对（由缓冲区队列保持的，未向系统传送的）输入的字符串，可以删除其中的 1 个文字或 1 行文字，也可以对每台终端分别设定在输入何种文字时进行上述删除处理。`tty` 结构体的 `tty.t_erase` 与 `tty.t_kill` 对应上述设定，在缺省状态下输入“#”时删除 1 个文字，输入“@”时删除 1 行文字。

用户程序通过向终端的特殊文件执行系统调用 `stty` 和 `gtty` 设定终端。但是，可设定的值仅限于 `tty` 结构体的 `tty.t_erase` 、`tty.t_kill` 、`tty.t_speeds` 和 `tty.t_flags` 。

根据终端的不同，在对特殊文字进行处理时有可能发生额外的延迟。适当设定 `tty.t_flags` 可使终端共用处理进行适当的延迟等待处理。

此外，根据所使用的终端，可使用的 ASCII 代码有可能受到限制。当终端只能处理大写文字时，从终端输入的大写文字将变为小写文字，向终端输出的文字则变成大写文字。因此，终端和系统内部所处理的文字类型分别统一至大写文字和小写文字。另外，当 ASCII 码受到限制时，终端对无法处理的符号，如表 13-4 所示，会将其变为反斜线和可处理符号的组合。

表 **13-4**　发生变化的符号

更改前	更改后
‘	\’
|	\!
~	\^
{	\(
}	\)

对表 13-5 所示的 ASCII 码进行输入输出时，将在系统内或终端内进行特殊的处理。

表 **13-5**　**ASCII** 码特殊文字

ASCII 码	名称	含义·处理
04	EOT（End Of Transmission）	终止传送
010	BS（Back Space）	删除 1 个文字
011	HT（Horizontal Tab）	追加水平方向的 tab
012	NL（New Line）	追加换行
014	NP（New Page）	追加换页
015	CR（Carriage Return）	使打印头回到行头
034	FS（File Separator）	发送 SIGQIT
040	SP（Space）	追加空格
0177	DEL（Delete）	发送 SIGINT

maptab[]

终端采用 ASCII 字符集。若干 ASCII 码用于控制终端。如果希望按照一般文字进行处理，需要在之前加入反斜线“\”进行转义。这些文字（ASCII 码）由 `maptab[]` 管理（代码清单 13-2 ）。

在代码清单 13-2 所示的例子中，（缺省情况下）输入“#”将删除一个文字，但输入“\#”后将传送“#”本身。“#”的 ASCII 码为 043，将 `maptab[043]` 设定为“#”。

此外，本例中在反斜线之后输入小写文字时，该文字将变为相应的大写文字。

代码清单 **13-2　maptab[]**（**dmr/tty.c)**

```
char    maptab[]
{
    000,000,000,000,004,000,000,000,
    000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,
    000,'|',000,'#',000,000,000,'`',
    '{','}',000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,
    '@',000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,000,000,
    000,000,000,000,000,000,'~',000,
    000,'A','B','C','D','E','F','G',
    'H','I','J','K','L','M','N','O',
    'P','Q','R','S','T','U','V','W',
    'X','Y','Z',000,000,000,000,000,
};
```

partab[]

电传终端使用 8 比特的 ASCII 码传送数据，其中 7 比特为 ASCII 码，第 8 比特位为偶校验位。设定校验位时需要使用 `partab[]`（代码清单 13-3，表 13-6 ）。与 `maptab[]` 相同，与某个 ASCII 码 n 所代表的文字相对应的 `partab[]` 元素为 `partab[n]` 。

`partab[]` 不仅表示校验位，也表示各个文字（ASCII 码）将引发何种操作。在输出字符串时，将使用此信息来判断是否需要延迟输出。

代码清单 **13-3　partab[]**（**dmr/partab.c)**

```
char partab[] {
    0001,0201,0201,0001,0201,0001,0001,0201,
    0202,0004,0003,0205,0005,0206,0201,0001,
    0201,0001,0001,0201,0001,0201,0201,0001,
    0001,0201,0201,0001,0201,0001,0001,0201,
    0200,0000,0000,0200,0000,0200,0200,0000,
    0000,0200,0200,0000,0200,0000,0000,0200,
    0000,0200,0200,0000,0200,0000,0000,0200,
    0200,0000,0000,0200,0000,0200,0200,0000,
    0200,0000,0000,0200,0000,0200,0200,0000,
    0000,0200,0200,0000,0200,0000,0000,0200,
    0000,0200,0200,0000,0200,0000,0000,0200,
    0200,0000,0000,0200,0000,0200,0200,0000,
    0000,0200,0200,0000,0200,0000,0000,0200,
    0200,0000,0000,0200,0000,0200,0200,0000,
```

```
    0200,0000,0000,0200,0000,0200,0200,0000,
    0000,0200,0200,0000,0200,0000,0000,0201
};
```

表 **13-6　partab[]**

比特位	含义
7	校验位。位于 7 比特的 ASCII 码之前
6~3	未使用
2~0	所引发的特殊操作的种类（参见 13.6 节的“ttyoutput()”）

KL11/DL11

与终端相关的代码分为两类：终端共用处理和各终端接口专用的设备驱动。虽然终端处理的大部分都集中在终端共用处理，但由于处理的流程涉及上述两类代码，因此下文将对两者进行同时介绍。下面将以 KL11/DL11 的设备驱动为例，介绍终端接口的设备驱动。

KL11/DL11 拥有 4 个 16 比特的寄存器。对终端的输入和向终端的输出分别具有属于自己的控制寄存器和数据寄存器（表 13-7 ）。

另外，KL11/DL11 具有 1 个端口，可以连接 1 台终端。系统也可以连接多个 KL11/DL11 接口。

表 **13-7　KL11/DL11** 的寄存器

寄存器	名称	含义
klrcsr	Receiver Status 寄存器	终端输入控制寄存器（参见表 13-8）
klrbuf	Receiver Buffer 寄存器	终端输入数据寄存器（参见表 13-9）
kltcsr	Transmitter Status 寄存器	终端输出控制寄存器（参见表 13-10）
kltbuf	Transmitter Buffer 寄存器	终端输出数据寄存器（参见表 13-11）

表 **13-8　klrcsr**

比特位	含义
11	终端处理中
7	接收处理（将数据传送至 klrbuf）结束时设置为 1
6	当第 7 比特位设置为 1 时中断优先级 4，向量 060 将引发中断

0	接收许可。此比特变为 1 时将第 7 比特位清 0

表 **13-9　klrbuf**

比特位	含义
7~0	由 7 比特 ASCII 码和头部的偶校验位所构成的 8 比特 ASCII 码

表 **13-10　kltcsr**

比特位	含义
7	终端接口已做好接受请求的准备
6	当第 7 比特位设为 1 时中断优先级 4，向量 064 将引发中断
2	用于维护。本书不做详细说明

表 **13-11　kltbuf**

比特位	含义
7~0	由 7 比特 ASCII 码和头部的偶校验位所构成的 8 比特 ASCII 码

KL11/DL11设备驱动的规格

各终端被赋予一系列的小编号，各接口的寄存器被映射到如代码清单 13-4 和图 13-5 所示的内核空间的高位地址。

代码清单 **13-4　KL/DL11** 的基地址（**dmr/kl.c)**

```
#define    KLADDR    0177560
#define    KLBASE    0176500
#define    DLBASE    0175610
```

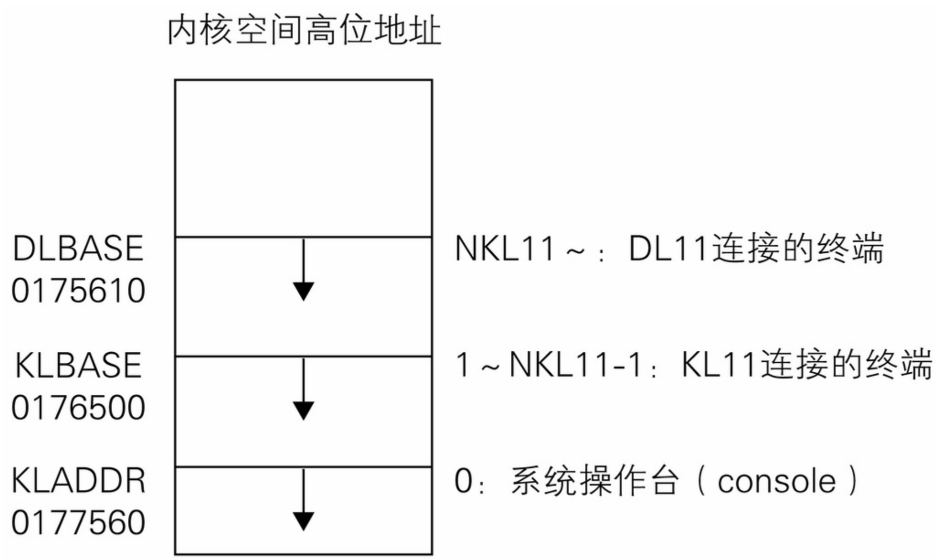


图 **13-5　KL11/DL11** 地址

与 KL11 连接的供系统操作台使用的终端，对应的寄存器的基地址被映射至 `KLADDR` （0177560 ），与 KL11 连接的其他终端相对应的寄存器被映射至 `KLBASE` （0176500 ）之后的地址，与 DL11 连接的终端相对应的寄存器被映射至 `DLBASE` （0175610 ）之后的地址。`NKL11` 与 `NDL11` 分别表示系统中存在的 KL11 与 DL11 接口的数量。系统管理者需要根据系统环境修改 `NKL11` 与 `NDL11` 的值，并重新构筑内核。代码清单 13-5 的示例中只存在 1 个 KL11 接口，这说明只存在一台供系统操作台使用的系统终端。

代码清单 **13-5　NKL11** 和 **NDL11**（**dmr/kl.c)**

```
#define    NKL11    1
#define    NDL11    0
```

KL11/DL11 设备驱动使用 `tty` 结构体的数组 `kl11[]` 管理各终端的信息（代码清单 13-6 ）。设备的小编号对应 `kl11[]` 的数组下标。

代码清单 **13-6　kl11[]**（**dmr/kl.c)**

```
struct    tty kl11[NKL11+NDL11];
```

KL11/DL11设备驱动函数

在代码清单 13-7 的示例中，对于 `cdevsw[]` 的大编号 0，注册了下述 KL11/DL11 设备驱动函数：`klopen()` 、`klclose()` 、`klread()` 、`klwrite()` 、`klsgtty()` 。

除了上述函数，再加上终端处理结束时引发中断的处理函数 `klxint()` ，以及终端输入时引发中断的处理函数 `klrint()` ，即构成了 KL11/DL11 设备驱动的全部函数。

代码清单 **13-7　cdevsw[]**（**conf.c)**

```
int    (*cdevsw[])()
{
```

```
    &klopen,   &klclose,  &klread,   &klwrite,  &klsgtty,  /* console */
```

13.2　终端的开启和关闭

klopen()

`klopen()` 是用于打开 KL11 终端的设备驱动（表 13-12，代码清单 13-8 ）。对终端的特殊文件执行系统调用 `open`，将调用注册于在 `cdevsw[].d_open` 的 `klopen()`。

根据小编号从 `kl11[]` 取得相应的 `tty` 结构体，将执行进程与终端相关联。

计算该终端的地址，对 `tty` 结构体和寄存器进行初始化设定。

表 **13-12　klopen()** 的参数

参数	含义
`dev`	设备编号
`flag`	`klopen()` 中未使用

代码清单 **13-8　klopen()**（**dmr/kl.c**）

```
1 klopen(dev, flag)
2 {
3     register char *addr;
4     register struct tty *tp;
5
6     if(dev.d_minor >= NKL11+NDL11) {
7         u.u_error = ENXIO;
8         return;
9     }
10    tp = &kl11[dev.d_minor];
11    if (u.u_procp->p_ttyp == 0) {
12        u.u_procp->p_ttyp = tp;
13        tp->t_dev = dev;
14    }
15    addr = KLADDR + 8*dev.d_minor;
16    if(dev.d_minor)
17        addr =+ KLBASE-KLADDR-8;
18    if(dev.d_minor >= NKL11)
19        addr =+ DLBASE-KLBASE-8*NKL11+8;
20    tp->t_addr = addr;
21    if ((tp->t_state&ISOPEN) == 0) {
22        tp->t_state = ISOPEN|CARR_ON;
23        tp->t_flags = XTABS|LCASE|ECHO|CRMOD;
24        tp->t_erase = CERASE;
25        tp->t_kill = CKILL;
26    }
27    addr->klrcsr =| IENABLE|DSRDY|RDRENB;
28    addr->kltcsr =| IENABLE;
29 }
```

6~9　检查小编号是否合适。

10　根据小编号，从 `kl11[]` 取得相应的 `tty` 结构体。

11~14　如果试图打开的终端进程尚未与终端相关联，则将刚才取得的 `tty` 结构体分配与该进程，并设定 `tty` 结构体的设备编号。

15~20　计算终端接口寄存器的基地址。将计算式进行简化后得到下述结果。

- 小编号 **0**：`tp->t_addr = KLADDR`
- 小编号 **1~NKL11-1**：`tp->t_addr = KLBASE + 8 ×( 小编号 -1)`
- 小编号 **NKL11~**：`tp->t_addr = DLBASE + 8 ×( 小编号 -NKL11)`

21~26　检查终端状态，如果尚未打开，则对 `tty` 结构体进行初始化处理。

27~28　对终端接口的寄存器进行初始化。`klrcsr` 与 `kltcsr` 是为了操作寄存器而定义的结构体的成员（代码清单 13-9）。`IENABLE` 、`DSRDY` 、`RDRENB` 是为了设置寄存器而定义的值（代码清单 13-10，代码清单 13-11）。

代码清单 **13-9　klregs（dmr/kl.c)**

```
struct klregs {
    int klrcsr;
    int klrbuf;
    int kltcsr;
    int kltbuf;
}
```

代码清单 **13-10　IENABLE（tty.h)**

```
#define    IENABLE    0100
```

代码清单 **13-11　DSRDY fl RDRENB（dmr/kl.c)**

```
#define    DSRDY     02
#define    RDRENB    01
```

klclose()

`klclose()` 用于对 KL11 进行关闭处理（表 13-13，代码清单 13-12 ）。对终端的特殊文件执行系统调用 `close`，将调用注册于 `cdevsw[].d_close` 的 `klclose()`。

根据小编号从 `kl11[]` 取得相应的 `tty` 结构体，刷新 `tty` 结构体持有的队列。并将 `tty` 结构体的 `tty.t_state` 清 0。

表 **13-13　klclose()** 的参数

参数	含义

dev	设备编号

代码清单 **13-12　klclose()（dmr/kl.c)**

```
klclose(dev)
{
    register struct tty *tp;

    tp = &kl11[dev.d_minor];
    wflushtty(tp);
    tp->t_state = 0;
}
```

wflushtty()

`wflushtty()` 用于清空队列（表 13-14，代码清单 13-13 ）。如果 `tty.t_outq` 中残留了数据，则进入睡眠状态直到数据被使用。

将清空队列的处理在 `flushtty()` 中进行。

表 **13-14　wflushtty()** 的参数

参数	含义
atp	tty 结构体

代码清单 **13-13　wflushtty()（dmr/tty.c)**

```
 1 wflushtty(atp)
 2 struct tty *atp;
 3 {
 4     register struct tty *tp;
 5
 6     tp = atp;
 7     spl5();
 8     while (tp->t_outq.c_cc) {
 9         tp->t_state =| ASLEEP;
10         sleep(&tp->t_outq, TTOPRI);
11     }
12     flushtty(tp);
13     spl0();
14 }
```

8~11　如果 `tty.t_outq` 中残留了数据，则将 `tty` 结构体设置为 `ASLEEP` 状态，然后使进程进入睡眠状态。当 `tty.t_outq` 变为空或置为 `TTLOWAT` 状态（后述）时进程被唤醒。

flushtty()

`flushtty()` 用于清空 `tty` 结构体持有的 3 个队列（表 13-15，代码清单 13-14 ）。

表 **13-15　flushtty()** 的参数

参数	含义
atp	tty 结构体

代码清单 **13-14　flushtty()**（**dmr/tty.c)**

```
 1 flushtty(atp)
 2 struct tty *atp;
 3 {
 4     register struct tty *tp;
 5     register int sps;
 6
 7     tp = atp;
 8     while (getc(&tp->t_canq) >= 0);
 9     while (getc(&tp->t_outq) >= 0);
10     wakeup(&tp->t_rawq);
11     wakeup(&tp->t_outq);
12     sps = PS->integ;
13     spl5();
14     while (getc(&tp->t_rawq) >= 0);
15     tp->t_delct = 0;
16     PS->integ = sps;
17 }
```

8~9　将 tty.t_canq 和 tty.t_outq 清空。

10~11　唤醒等待 tty.t_rawq 和 tty.t_outq 被清空的进程。

12~13　保存 PSW，并将处理器的优先级设为 5。

14　将 tty.t_rawq 清空。

15　由于 tty.t_rawq 已被清空，因此将分隔符计数器也清 0。

16　恢复保存的 PSW。

13.3　终端的设定

gtty()

gtty() 为系统调用 gtty 的处理函数（表 13-17，代码清单 13-15 ）。该函数将终端的相关信息（tty 结构体的数据）读取至用户空间，所取得的数据与保存区域的对应关系如表 13-16 所示。

大部分处理在 sgtty() 中进行。该函数为 gtty() 与 stty() 的共用函数。

表 **13-16**　系统调用 **gtty** 所读取的数据

成员	含义
int[0]	tty.t_speed
int[1]	tty.t_erase 、tty.t_kill

int[2]	tty.t_flag

表 **13-17** 系统调用 **gtty** 的参数

参数	含义
r0	用于打开终端特殊文件的文件描述符
u.u_arg[0]	用于存放所读取的终端信息的用户空间地址

代码清单 **13-15** **gtty()**（**dmr/tty.c)**

```
 1 gtty()
 2 {
 3     int v[3];
 4     register *up, *vp;
 5
 6     vp = v;
 7     sgtty(vp);
 8     if (u.u_error)
 9         return;
10     up = u.u_arg[0];
11     suword(up, *vp++);
12     suword(++up, *vp++);
13     suword(++up, *vp++);
14 }
```

7　调用 sgtty()，设定其参数为指向用于存放 tty 结构体数据的 int[3] 的指针。

8~9　如果在执行 sgtty() 中发生错误则返回。

10~13　将 tty 结构体的信息读取至用户空间。

stty()

stty() 为系统调用 stty 的处理函数（表 13-19，代码清单 13-16 ）。该函数将用户空间的数据设定到终端（tty 结构体）。设定的数据如表 13-18 所示。

大部分处理在 sgtty() 中进行。该函数为 gtty() 与 stty() 的共用函数。

表 **13-18** 系统调用 **stty** 所设定的数据

成员	含义
int[0]	tty.t_speed
int[1]	tty.t_erase 、tty.t_kill
int[2]	tty.t_flag

表 **13-19**　系统调用 **stty** 的参数

参数	含义
r0	用于打开终端特殊文件的文件描述符
u.u_arg[0]	指向用户空间中的数据（int[3]）的指针，该数据将被写入 tty 结构体中

代码清单 **13-16**　**stty()**（**dmr/tty.c)**

```
 1 stty()
 2 {
 3     register int *up;
 4
 5     up = u.u_arg[0];
 6     u.u_arg[0] = fuword(up);
 7     u.u_arg[1] = fuword(++up);
 8     u.u_arg[2] = fuword(++up);
 9     sgtty(0);
10 }
```

5~8　u.u_arg[] 中容纳用户程序指定的值。u.u_arg[0] 对应 tty.t_speed，u.u_arg[1] 对应 tty.t_erase 和 tty.t_kill，u.u_arg[2] 对应 tty.t_mode。

9　调用 sgtty()。通过将参数指定为 0，表示对 tty 结构体进行写入处理。

sgtty()

sgtty() 用于执行 stty() 和 gtty() 的共用处理（表 13-20，代码清单 13-17）。首先取得与已打开的特殊文件相对应的 inode[] 元素，并确认其为字符设备。然后将 inode.i_addr[0] 设定为设备编号。最后使用大编号执行在 cdevsw[].d_sgtty 中注册的 klsgtty()。

表 **13-20**　**sgtty()** 的参数

参数	含义
v	赋予 cdevsw[].d_sgtty（klsgtty()）的参数

代码清单 **13-17**　**sgtty()**（**dmr/tty.c)**

```
sgtty(v)
int *v;
{
    register struct file *fp;
    register struct inode *ip;

    if ((fp = getf(u.u_ar0[R0])) == NULL)
        return;
    ip = fp->f_inode;
    if ((ip->i_mode&IFMT) != IFCHR) {
        u.u_error = ENOTTY;
        return;
    }
```

```
14     (*cdevsw[ip->i_addr[0].d_major].d_sgtty)(ip->i_addr[0], v);
15 }
```

7~8 由于用户进程的`r0`保存了终端的文件描述符，因此执行`getf()`取得相对应的`file[]`元素。

9 获取由`file[]`元素所指向的`inode[]`元素。

10~13 如果`inode[]`元素并非字符型的特殊文件则出错。

14 调用用于设定终端的设备驱动。

klsgtty()

`klsgtty()`为用于设定及读取`tty`结构体的 KL11 设备驱动（表 13-21，代码清单 13-18 ）。根据小编号从`kl11[]`获取相应的`tty`结构体，并执行`ttystty()`。

表 **13-21** **klsgtty()** 的参数

参数	含义
dev	设备编号
v	赋予`ttystty()`的参数

代码清单 **13-18** **klsgtty()**（**dmr/kl.c)**

```
klsgtty(dev, v)
int *v;
{
    register struct tty *tp;

    tp = &kl11[dev.d_minor];
    ttystty(tp, v);
}
```

ttystty()

`ttystty()`是实际上对`tty`结构体进行写入或读取处理的函数（表 13-22，代码清单 13-19 ）。该函数由设备驱动调用。参数`av`为 0 时进行写入，其他值时进行读取。

表 **13-22** **ttystty()** 的参数

参数	含义
atp	指向`tty`结构体的指针
av	0：将`u.u_arg[]`中的数据赋予`tty`结构体。处理结束时返回 0。 0 以外：将 tty 结构体的数据读取至由`av`所示的地址。处理结束时返回 1

代码清单 **13-19　ttystty()**（**dmr/tty.c)**

```
 1 ttystty(atp, av)
 2 int *atp, *av;
 3 {
 4     register  *tp, *v;
 5 
 6     tp = atp;
 7     if(v = av) {
 8         *v++ = tp->t_speeds;
 9         v->lobyte = tp->t_erase;
10         v->hibyte = tp->t_kill;
11         v[1] = tp->t_flags;
12         return(1);
13     }
14     wflushtty(tp);
15     v = u.u_arg;
16     tp->t_speeds = *v++;
17     tp->t_erase = v->lobyte;
18     tp->t_kill = v->hibyte;
19     tp->t_flags = v[1];
20     return(0);
21 }
```

7~13　读取 `tty` 结构体的处理。`lobyte` 和 `hibyte` 是为了访问 2 字节数据的低位字节和高位字节而定义的结构体的成员变量（代码清单 13-20）。

代码清单 **13-20　lobyte** 和 **hibyte**（**param.h)**

```
struct
{
    char    lobyte;
    char    hibyte;
};
```

14~20　对 `tty` 结构体进行写入的处理。首先执行 `wflushtty()` 清空队列。

13.4　从终端输入文字

从终端输入文字时的处理如下所示（图 13-6 ）。

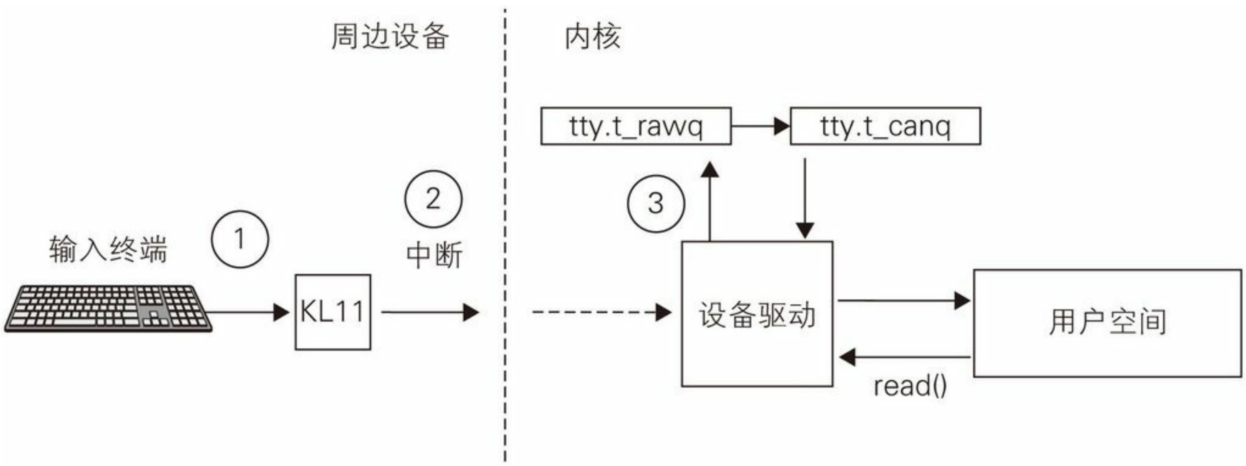


图 **13-6**　从终端输入文字

1. 从终端输入数据后，数据将被赋予 KL11/DL11的输入寄存器

2. KL11/DL11的寄存器设定了数据后将引发中断

3. 中断处理函数从 KL11/DL11的输入寄存器读取数据，对其进行适当的变换，并对特殊文字及分隔符进行处理，然后将数据追加至`tty.t _ rawq`

在 3. 的处理中，为了避免在向 `tty.t_rawq` 追加数据时缓冲区发生溢出的问题，当 `tty.t_rawq` 中的数据达到 `TTYHOG` （代码清单 13-21 ）件时，队列将被清空。

代码清单 **13-21　TTYHOG**（**tty.h**）

```
#define    TTYHOG    256
```

klrint()

`klrint()` 为中断处理函数，用于处理从终端输入数据时发生的中断（表 13-23，代码清单 13-22 ）。首先根据小编号从 `kl11[]` 取得相应的 `tty` 结构体，同时从 KL11 的输入寄存器中获取输入的数据。

设置输入控制寄存器，使其处于可接收状态，然后执行 `ttyinput()` 将输入的数据追加至 `tty.t_rawq` 。

表 **13-23　klrint()** 的参数

参数	含义
dev	设备编号

代码清单 **13-22　klrint()**（**dmr/kl.c**）

```
klrint(dev)
{
    register int c, *addr;
    register struct tty *tp;

    tp = &kl11[dev.d_minor];
```

```
 7     addr = tp->t_addr;
 8     c = addr->klrbuf;
 9     addr->klrcsr =| RDRENB;
10     if ((c&0177)==0)
11         addr->kltbuf = c;
12     ttyinput(c, tp);
13 }
```

10~11 如果接收的数据为空，则将输出数据寄存器设置为接收结果。这是为了保险起见而进行的处理。

ttyinput()

因终端输入引发中断的处理函数将调用 `ttyinput()`（表 13-24，代码清单 13-23 ）。该函数将数据追加至 `tty.t_rawq` 。

此处将进行下面这些特殊处理。

- CR 模式时，将“\r”变为“\n”
- 在 RAW 以外的模式时，如果输入了 FS 和 DEL，则将该终端控制的所有进程发送 `SIGQIT`（FS 时）或 `SIGINT`（DEL 时）信号
- 如果 `tty.t_rawq` 保存的数据件数大于等于 `TTYHOG` 时，将清空全部 3 个队列
- 如果终端只能处理大写文字时，将输入的大写文字变为小写文字
- RAW 模式，或输入了换行符或 EOT 时，在输入文字后追加分隔符（0377），并将其追加至 `tty.t_rawq`，同时递增分隔符计数器的值
- ECHO 模式时，将输入的文字输出至终端

表 **13-24**　**ttyinput()** 的参数

参数	含义
`ac`	输入文字（8 比特 ASCII 码）
`atp`	`tty` 结构体

代码清单 **13-23**　**ttyinput()**（**dmr/tty.c)**

```
ttyinput(ac, atp)
struct tty *atp;
{
    register int t_flags, c;
    register struct tty *tp;

    tp = atp;
    c = ac;
    t_flags = tp->t_flags;
    if ((c =& 0177) == '\r' && t_flags&CRMOD)
        c = '\n';
    if ((t_flags&RAW)==0 && (c==CQUIT || c==CINTR)) {
        signal(tp, c==CINTR? SIGINT:SIGQIT);
```

```
14          flushtty(tp);
15          return;
16      }
17      if (tp->t_rawq.c_cc>=TTYHOG) {
18          flushtty(tp);
19          return;
20      }
21      if (t_flags&LCASE && c>='A' && c<='Z')
22          c =+ 'a'-'A';
23      putc(c, &tp->t_rawq);
24      if (t_flags&RAW || c=='\n' || c==004) {
25          wakeup(&tp->t_rawq);
26          if (putc(0377, &tp->t_rawq)==0)
27              tp->t_delct++;
28      }
29      if (t_flags&ECHO) {
30          ttyoutput(c, tp);
31          ttstart(tp);
32      }
33  }
```

10~11　CR 模式时的处理。

12~16　发送信号的处理。

17~20　如果 `tty.t_rawq` 中保存的数据过多，则将 3 个队列全部清空。

21~22　如果终端只能处理大写文字，则将输入的大写文字变为小写文字。

23　使用 `putc()` 将输入的文字追加至 `tty.t_rawq` 。

24~28　对分隔符的处理。

29~32　ECHO 模式时的处理。

13.5　读取输入的数据

将输入的数据读取至用户空间的流程如下所示（图 13-7 ）。

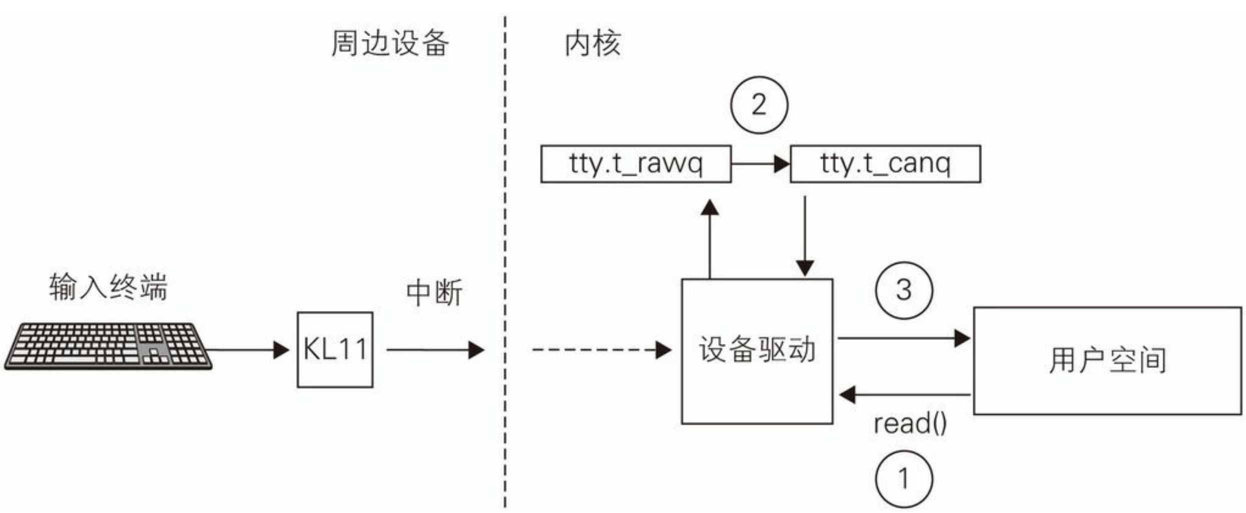


图 **13-7**　读取输入的数据

1. 用户程序对终端执行系统调用read

2. 从tty.t _ rawq 读取数据，将其追加至tty.t _ canq ，同时对下列特殊文字进行处理：删除1个文字（#），删除1行文字（@），转义处理（/）。将输入数据从 tty.t _ rawq 转存到 tty.t _ canq 直至遇到分隔符

3. 将 tty.t _ canq 数据读取至用户空间

klread()

klread() 是将从终端输入的数据读取至用户空间的设备驱动函数（表 13-25，代码清单 13-24 ）。对终端的特殊文件执行系统调用 read 后，将执行在 cdevsw[].d_read 中注册的 klread() 。

根据小编号从 kl11[] 取得相应的 tty 结构体，并执行 ttread() 。

表 **13-25　klread()** 的参数

参数	含义
dev	设备编号

代码清单 **13-24　klread()**（**dmr/kl.c)**

```
klread(dev)
{
    ttread(&kl11[dev.d_minor]);
}
```

ttread()

ttread() 在 read 时由设备驱动调用（表 13-26，代码清单 13-25 ）。首先检查是否已打开终端，然后执行 canon() 将数据从 tty.t_rawq 传送至 tty.t_canq ，再从 tty.t_canq 向用户空间传送数据。

表 **13-26　ttread()** 的参数

参数	含义
atp	指向 tty 结构体的指针

代码清单 **13-25　ttread()**（**dmr/tty.c)**

```
ttread(atp)
struct tty *atp;
{
    register struct tty *tp;

    tp = atp;
    if ((tp->t_state&CARR_ON)==0)
    return;
    if (tp->t_canq.c_cc || canon(tp))
        while (tp->t_canq.c_cc && passc(getc(&tp->t_canq))>=0);
```

```
11 }
```

7~8 如果未打开终端则不作任何处理直接返回。

9~10 如果 `tty.t_canq` 中残留了数据，或是 `canon()` 的结果不为 `0`，则使用 `passc()` 向用户空间传送 `tty.t_canq` 中的数据。

canon()

`canon()` 用于从 `tty.t_rawq` 向 `tty.t_canq` 传送数据（表 13-27，代码清单 13-28 ）。如果向 `tty.t_canq` 成功追加了数据则返回 1。

处理时使用 `canonb[]` 作为工作区域（代码清单 13-26，代码清单 13-27 ）。从 `tty.t_rawq` 的起点开始将数据转存至 `canonb[]` 直至遇到分隔符（0377 ），然后在进行删除 1 个文字、删除 1 行文字和转义处理的同时，再将数据追加至 `tty.t_canq` （图 13-8 ）。

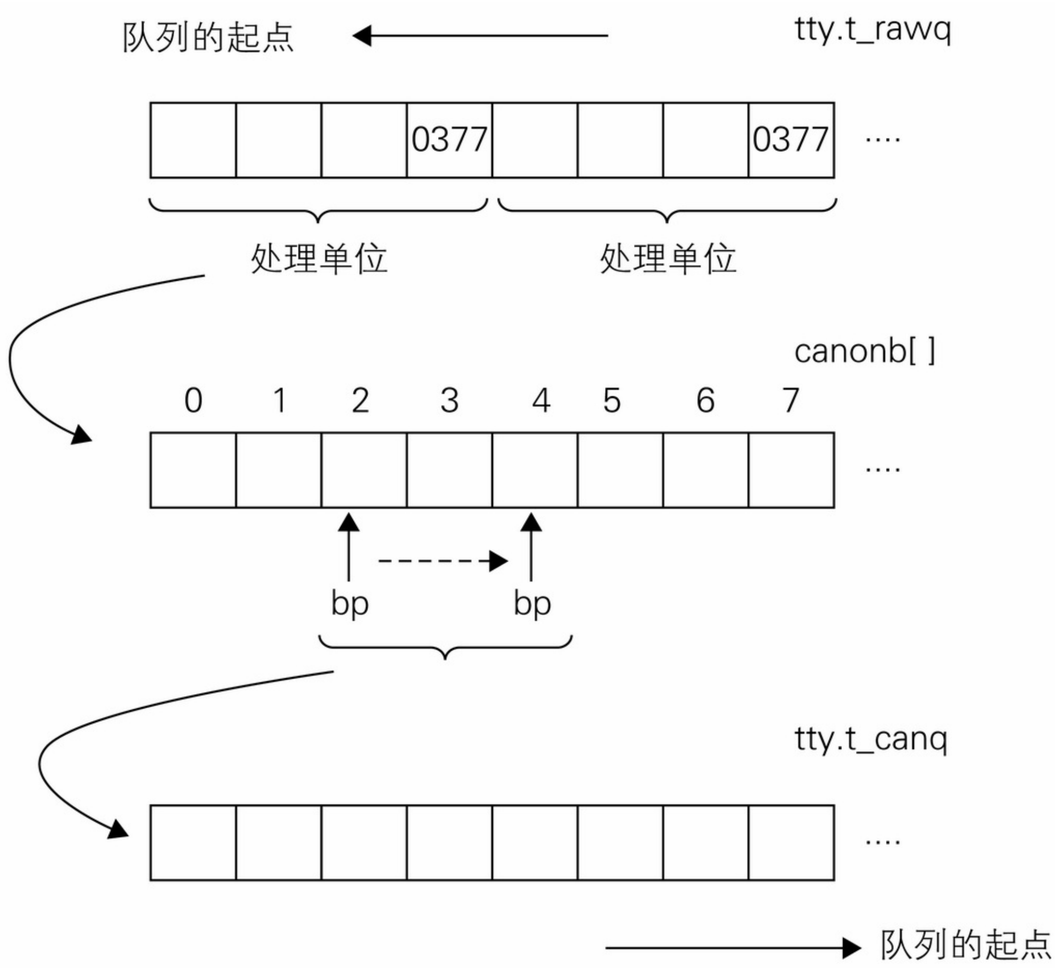


图 **13-8　canon()**

`bp` 是 `canonb[]` 中的工作指针，当 `bp` 之前的文字为“\”时，利用 `maptab[]` 进行转义处理。但是如果 `bp` 之前的倒数第 2 个文字也为“\”时，则认为“\”是已经转义过的文字。

当从 `tty.t_rawq` 获取的文字为用于删除 1 个文字的特殊文字时，将 `bp` 向前移动 1 个文字的位置。当为用于删除 1 行文字的特殊文字时，将 `bp` 移动至作为处理起点的 `&canonb[2]`（图 13-9 ）。

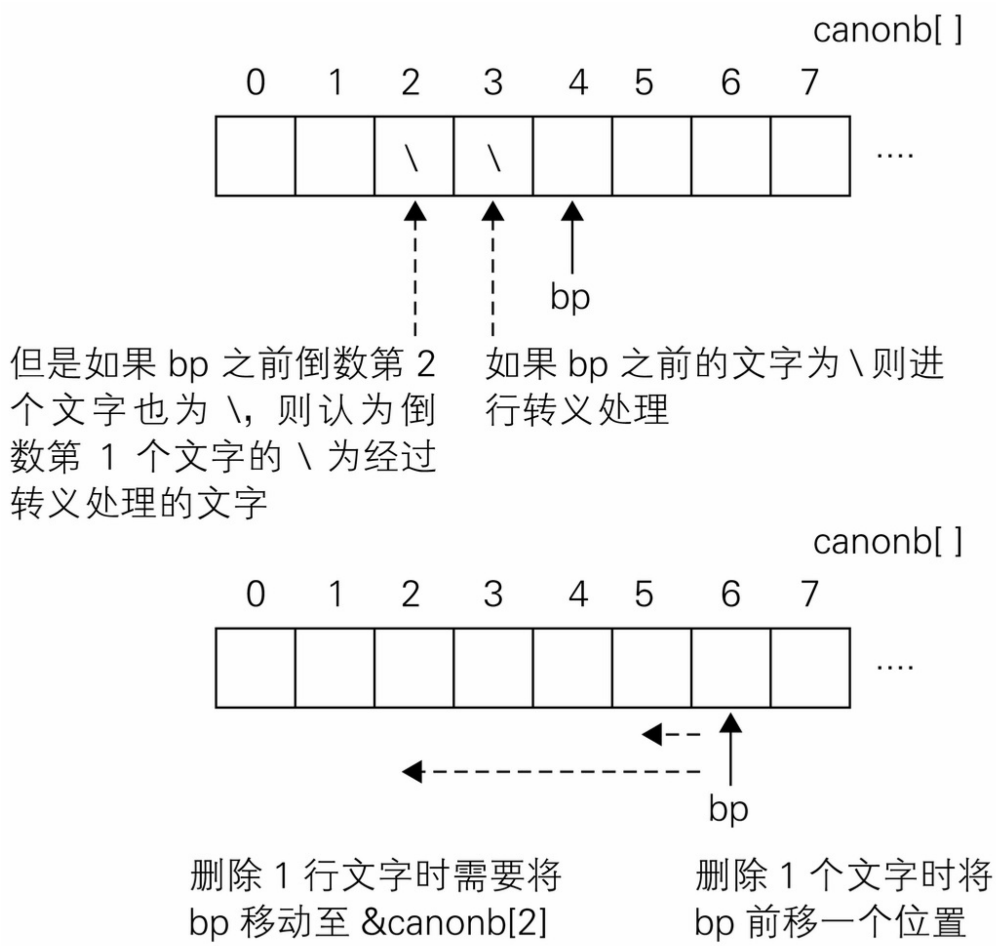


图 **13-9　canonb[]** 的处理

代码清单 **13-26　canonb**（**system.h)**

```
char    canonb[CANBSIZ];
```

代码清单 **13-27　CANBSIZ**（**param.h)**

```
#define    CANBSIZ    256
```

表 **13-27　canon()** 的参数

参数	含义
atp	指向 `tty` 结构体的指针

代码清单 **13-28　canon()**（**dmr/tty.c)**

```
1 canon(atp)
2 struct tty *atp;
3 {
4     register char *bp;
5     char *bp1;
6     register struct tty *tp;
7     register int c;
8
9     tp = atp;
10    spl5();
11    while (tp->t_delct==0) {
12        if ((tp->t_state&CARR_ON)==0)
13            return(0);
14        sleep(&tp->t_rawq, TTIPRI);
15    }
16    spl0();
17 loop:
18    bp = &canonb[2];
19    while ((c=getc(&tp->t_rawq)) >= 0) {
20        if (c==0377) {
21            tp->t_delct--;
22            break;
23        }
24        if ((tp->t_flags&RAW)==0) {
25            if (bp[-1]!='\\') {
26                if (c==tp->t_erase) {
27                    if (bp > &canonb[2])
28                        bp--;
29                    continue;
30                }
31                if (c==tp->t_kill)
32                    goto loop;
33                if (c==CEOT)
34                    continue;
35            } else
36            if (maptab[c] && (maptab[c]==c || (tp->t_flags&LCASE))) {
37                if (bp[-2] != '\\')
38                    c = maptab[c];
39                bp--;
40            }
41        }
42        *bp++ = c;
43        if (bp>=canonb+CANBSIZ)
44            break;
45    }
46    bp1 = bp;
47    bp = &canonb[2];
48    c = &tp->t_canq;
49    while (bp<bp1)
50        putc(*bp++, c);
51    return(1);
52 }
```

10~16　如果 tty.t_rawq 中的数据不包含分隔符，且终端处于未打开状态，则返回 0。否则进入睡眠状态，直到分隔符被追加至 tty.t_rawq 。

18　将 bp 设定为处理起点 &canonb[2] 。

19　以文字为单位从 tty.t_rawq 中获取数据，并将其追加至 canonb[] ，直至遇到分隔符。

20~23　遇到分隔符时，递减分隔符计数器并退出循环。

24~41　如果不为 RAW 模式，且 bp 之前的文字为“\”时，根据 maptab[] 进行转义处理。但是如果 bp 之前倒数第 2 个文字也为“\”，则认为 bp 之前的“\”为已经经过转义的文字。

bp 之前的文字不为“\”时，进行下述处理。

- 删除 1 个文字时将 bp 前移 1 位，但不会移动至小于 &canonb[2] 的位置
- 删除 1 行文字时返回 loop 处（基本等同于将 bp 前移至 &canonb[2] ）
- 如果为 EOT 时继续处理下一个文字

42　将文字追加至 canonb[] 。

43~44　如果 canonb[] 已无空间，则退出循环。

46~50　将 &canonb[2] 至 bp 所对应范围的数据追加至 tty.t_canq 。

51　如果 canon() 的处理正常结束则返回 1。

13.6　向终端输出数据

向终端输出数据的流程如下所示（图 13-10 ）。

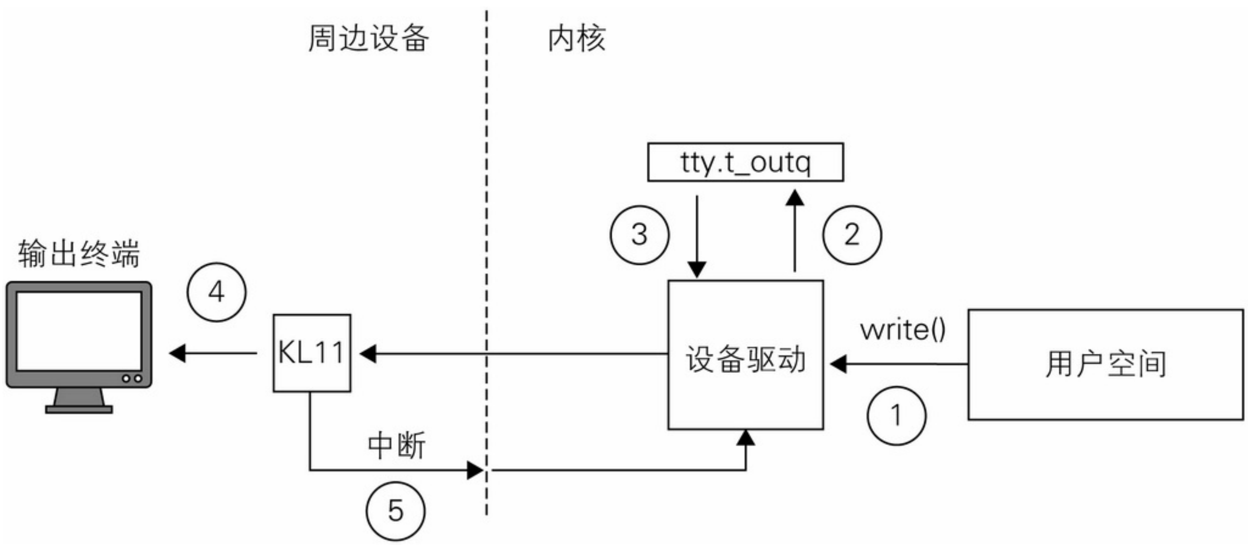


图 **13-10**　向终端输出数据

1. 用户程序对终端执行系统调用 write

2. 用户空间的数据被追加至tty.t _ outq

3. 将位于 tty.t _ outq 起始位置的数据赋予 KL11寄存器，开始向终端输出数据

4. 当向终端输出数据的处理结束后，KL11将引发中断。如果tty.t _ outq 中残留了数据，中断处理函数将继续进行输出处理

当上述 2 中向 tty.t_outq 追加数据时，会根据终端的设置对输出文字进行变换（例如，当终端只能处理大写文字时，将小写文字变为大写文字）。

在处理一部分特殊文字时会花费较长时间（例如会发生移动打印头的处理）。因此，设备驱动在此类处理结束前将延迟输出处理。在 2 中向 tty.t_outq 追加数据时，如果数据为特殊文字，则将额外向 tty.t_outq 追加 1 字节长的数据（如表 13-28 所示）。 在 3 的处理中将 tty.t_outq 的数据赋予 KL11 寄存器时，如果从 tty.t_outq 中取得的 1 字节长的数据的最高比特位为 1，则执行 timeout() 并在指定 tick 发生后再继续进行输出处理。在此延迟处理过程中，为 tty 结构体设置 TIMEOUT 标志位。

表 **13-28**　延迟指示数据

比特位	含义
7	1。执行延迟处理
6~0	延迟时间（tick）

为了避免 tty.t_outq 中保存太多数据，定义了 TTHIWAT （High water mark） 和 TTLOWAT （Low water mark）（代码清单 13-29 ）。在 2 中向 tty.t_outq 追加数据时，如果队列中的数据件数（文字数）超过了 TTHIWAT ，则在向 tty.t_outq 追加数据前，首先执行上述 3 的处理以向终端输出文字，然后进入睡眠状态。当 tty.t_outq 中的数据件数（文字数）通过向终端输出数据变为 TTLOWAT 时，处于睡眠状态的进程被唤醒，并将继续向 tty.t_outq 追加数据。

代码清单 **13-29**　**TTHIWAT** 和 **TTLOWAT**（**tty.h)**

```
#define    TTHIWAT    50
#define    TTLOWAT    30
```

klwrite()

klwrite() 是用于向终端进行输出的设备驱动函数（表 13-29，代码清单 13-30 ）。在对终端的特殊文件执行系统调用 write 后，将会调用在 cdevsw[].d_write 中注册的 klwrite() 。

根据小编号从 kl11[] 获取相应的 tty 结构体，并执行 ttwrite() 。

表 **13-29**　**klwrite()** 的参数

参数	含义
dev	设备编号

代码清单 **13-30**　**klwrite()**（**dmr/kl.c)**

```
klwrite(dev)
{
    ttwrite(&kl11[dev.d_minor]);
}
```

ttwrite()

`ttwrite()` 为终端共用处理函数，由设备驱动在向终端进行输出时调用（表 13-30，代码清单 13-31 ）。

首先执行 `ttyoutput()` ，将用户程序指定的数据全部追加至 `tty.t_outq` ，然后执行 `ttstart()` 开始向终端进行输出。

但是，如果 `tty.t_outq` 中保存了过多的数据，需要先将其输出至终端以腾空 `tty.t_outq` 。

表 **13-30　ttwrite()** 的参数

参数	含义
atp	tty 结构体

代码清单 **13-31　ttwrite()（dmr/tty.c)**

```
 1 ttwrite(atp)
 2 struct tty *atp;
 3 {
 4     register struct tty *tp;
 5     register int c;
 6
 7     tp = atp;
 8     if ((tp->t_state&CARR_ON)==0)
 9         return;
10     while ((c=cpass())>=0) {
11         spl5();
12         while (tp->t_outq.c_cc > TTHIWAT) {
13             ttstart(tp);
14             tp->t_state =| ASLEEP;
15             sleep(&tp->t_outq, TTOPRI);
16         }
17         spl0();
18         ttyoutput(c, tp);
19     }
20     ttstart(tp);
21 }
```

8~9　如果终端处于未打开的状态则不做任何处理立即返回。

10　对用户程序传送的数据进行逐字处理。

12~16　如果 `tty.t_outq` 中保存了超过 `TTHIWAT` 的数据，执行 `ttstart()` 促使系统进行输出处理，同时设置 `ASLEEP` 标志位进入睡眠状态。当 `tty.t_outq` 中的数据减少至 `TTLOWAT` 时，终端输出结束并引发中断，进程将被该中断的处理函数唤醒。

18　执行 `ttyoutput()` ，将数据逐字追加至 `tty.t_outq` 。

20　执行 `ttstart()` 开始向终端进行输出。

ttyoutput()

`ttyoutput()` 用于向 `tty.t_outq` 追加 1 个文字（表 13-31，代码清单 13-32 ）。该函数执行下述特殊处理。

- 如果不处于 RAW 模式，且当前文字为 EOT 时，不进行追加处理。

- 如果处于 XTABS 模式，且当前文字为“\t”时，输出空白使打印头移动至以 8 个文字为间距的下一个位置。

- 如果终端只能处理大写文字，则将小写文字变为大写文字，并将终端无法处理的符号变为反斜线和终端能够处理的符号的组合。

- CR 模式时将“\n”变为“\r\n”。

此外，当特殊处理导致处理时间变长时，向 `tty.t_outq` 追加额外的 1 字节数据，表示进行输出延迟处理。特殊处理通过相应的 `partab[]` 元素的低位比特进行判断。

表 **13-31　ttyoutput()** 的参数

参数	含义
`ac`	输出文字
`tp`	`tty` 结构体

代码清单 **13-32　ttyoutput()**（**dmr/tty.c)**

```
ttyoutput(ac, tp)
struct tty *tp;
{
    register int c;
    register struct tty *rtp;
    register char *colp;
    int ctype;

    rtp = tp;
    c = ac&0177;
    if (c==004 && (rtp->t_flags&RAW)==0)
        return;
    if (c=='\t' && rtp->t_flags&XTABS) {
        do
            ttyoutput(' ', rtp);
        while (rtp->t_col&07);
        return;
    }
    if (rtp->t_flags&LCASE) {
        colp = "({)}!|^~'`";
        while(*colp++)
            if(c == *colp++) {
                ttyoutput('\\', rtp);
                c = colp[-2];
                break;
            }

        if ('a'<=c && c<='z')
            c =+ 'A' - 'a';
    }
    if (c=='\n' && rtp->t_flags&CRMOD)
        ttyoutput('\r', rtp);
    if (putc(c, &rtp->t_outq))
        return;
    colp = &rtp->t_col;
    ctype = partab[c];
    c = 0;
    switch (ctype&077) {

    /* 一般处理 */
    case 0:
```

```
41          (*colp)++;
42
43      /* 不做显示 */
44      case 1:
45          break;
46
47      /* 删除之前的1个文字 */
48      case 2:
49          if (*colp)
50              (*colp)--;
51          break;
52
53      /* 换行 */
54      case 3:
55          ctype = (rtp->t_flags >> 8) & 03;
56          if(ctype == 1) {
57              if (*colp)
58                  c = max((*colp>>4) + 3, 6);
59          } else
60          if(ctype == 2) {
61              c = 6;
62          }
63          *colp = 0;
64          break;
65
66      /* 水平制表符 */
67      case 4:
68          ctype = (rtp->t_flags >> 10) & 03;
69          if(ctype == 1) {
70              c = 1 - (*colp | ~07);
71              if(c < 5)
72                  c = 0;
73          }
74          *colp =| 07;
75          (*colp)++;
76          break;
77
78      /* 垂直方向的处理 */
79      case 5:
80          if(rtp->t_flags & VTDELAY)
81              c = 0177;
82          break;
83
84      /* CR */
85      case 6:
86          ctype = (rtp->t_flags >> 12) & 03;
87          if(ctype == 1) {
88              c = 5;
89          } else
90          if(ctype == 2) {
91              c = 10;
92          }
93          *colp = 0;
94      }
95      if(c)
96          putc(c|0200, &rtp->t_outq);
97  }
```

11~12 对 EOT 的处理。

13~18 XTABS 模式时的处理。

19~29 当终端只能处理大写文字时的处理。

30~31 CR 模式时的处理。将“\r”首先追加至 `tty.t_outq` 。

32~33 向 `tty.t_outq` 追加 1 个文字。

35 从此处开始为输出延迟判断处理。如果需要的话调整代表打印头水平位置的`tty.t_col`，向 `tty.t_outq` 追加额外的 1 字节数据，表示进行输出延迟处理（参照表 13-32）。根据 `tty.t_flags` 的值不同，延迟时间也有可能发生变化。具体请参见 UPM(2) 中关于系统调用 `stty` 的说明。

表 **13-32** 延迟时间和 **tty.t_col** 的位置

partab[2-0]	含义	**tty.t_col**	延迟时间
0	一般文字	+1	0
1	未输出的文字	±0	0
2	删除之前的 1 个文字	-1	0
3	换行	0	换行类型为 1 时，根据当前打印头在 X 方向的位置，选择在 X/16+3 和 6 中较大的值。换行类型为 2 时设定延迟时间为 6，此外为 0
4	水平制表符	8 的倍数	水平制表符类型为 1 时，根据当前打印头的位置与 8 的倍数之间的差值，所得到的值也有所不同。如果差值大于等于 5 则为该差值，否则为 0
5	垂直方向的处理	±0	如果垂直方向的处理会导致延迟，则将延迟时间设定为 0177（延迟时间最大值），否则为 0
6	CR	0	CR 类型为 1 时将延迟时间设定为 5，CR 类型为 2 时设定为 10，此外 为 0

ttstart()

`ttstart()` 是向终端进行输出的函数（表 13-33，代码清单 13-33 ）。

如果输出的 1 字节数据的最高位比特为 1，则进行输出延迟处理。

表 **13-33** **ttstart()** 的参数

参数	含义
`atp`	`tty` 结构体

代码清单 **13-33** **ttstart()**（**dmr/tty.c)**

```
ttstart(atp)
struct tty *atp;
{
    register int *addr, c;
    register struct tty *tp;
    struct { int (*func)(); };

    tp = atp;
```

```
9      addr = tp->t_addr;
10     if (tp->t_state&SSTART) {
11         (*addr.func)(tp);
12         return;
13     }
14     if ((addr->tttcsr&DONE)==0 || tp->t_state&TIMEOUT)
15         return;
16     if ((c=getc(&tp->t_outq)) >= 0) {
17         if (c<=0177)
18             addr->tttbuf = c | (partab[c]&0200);
19         else {
20             timeout(ttrstrt, tp, c&0177);
21             tp->t_state =| TIMEOUT;
22         }
23     }
24 }
```

10~13　与 KL11/DL11 无关的处理。在一部分终端接口中使用。

14~15　如果终端处于发送处理或发送延迟处理中则返回。`tttcsr` 是为了操作终端接口的发送状态寄存器而定义的结构体的成员变量（代码清单 13-34）。

代码清单 **13-34**　用于操作接#寄存器的结构体（**dmr/tty.c)**

```
struct {
    int ttrcsr;
    int ttrbuf;
    int tttcsr;
    int tttbuf;
};
```

16~22　如果 `tty.t_outq` 中存在数据，首先从队列的头部取得 1 个文字。如果所取得的 1 字节数据包含 7 比特信息则判断其为（7 比特的）ASCII 码，将偶校验位追加至最高位比特，并将该数据赋予终端接口的发送数据寄存器。当设置了发送数据寄存器后，终端输出处理将被启动。

最高位比特为 1 时表示需要进行延迟处理。在执行 `timeout()` 并经过由所取得数据的低位 7 比特所示的 tick 数后，执行 `ttrstrt()` 。`tty` 结构体内表示设置了正在进行延迟处理的 `TIMEOUT` 标志位。

ttrstrt()

`ttrstrt()` 由 `ttstart()` 注册至 `callout[]` ，在所设定的时刻由 `clock()` 进行调用（表 13-34，代码清单 13-35 ）。当解除了 `TIMEOUT` 状态后，该函数将执行 `ttstart()` 继续进行输出处理。

表 **13-34**　**ttrstrt()** 的参数

参数	含义
atp	tty 结构体

代码清单 **13-35**　**ttrstrt()**（**dmr/tty.c)**

```
ttrstrt(atp)
{
    register struct tty *tp;
```

```

    tp = atp;
    tp->t_state =& ~TIMEOUT;
    ttstart(tp);
}
```

klxint()

`klxint()` 为终端输出处理结束后引发中断的处理函数（表 13-35，代码清单 13-36 ）。根据小编号从 `kl11[]` 获取相应的 `tty` 结构体，并执行 `ttstart()` 继续终端的输出处理。

表 **13-35　klxint()** 的参数

参数	含义
dev	设备编号

代码清单 **13-36　klxint()**（**dmr/kl.c)**

```
1 klxint(dev)
2 {
3     register struct tty *tp;
4
5     tp = &kl11[dev.d_minor];
6     ttstart(tp);
7     if (tp->t_outq.c_cc == 0 || tp->t_outq.c_cc == TTLOWAT)
8         wakeup(&tp->t_outq);
9 }
```

6　执行 `ttstart()`，如果 `tty.t_outq` 中残留了数据，则继续向终端输出文字。

7~8　如果 `tty.t_outq` 中不再有数据，或者已使用了一部分时，唤醒正在等待 `tty.t_outq` 中的数据被消耗的进程。

13.7　小结

- 内核提供了对电传终端处理的支持。
- 系统用户通过电传终端与系统进行交流。
- 最少存在 1 台供系统操作台使用的终端。
- 多位用户可以使用多台终端同时操作系统。
- 终端具有 3 个缓冲区队列，在系统与终端间适当变换数据并传送。

第 **VII** 部分　启动系统

本书的最后将介绍系统是如何启动的,第 VII 部分会对下述内容进行说明。

- 在内核未运行,且无法进行文件操作的情况下,如何将内核程序读取至内存
- 虚拟地K空间如何从无效状态迁移至有效状态

通过阅读本部分的内容,读者将会了解系统是如何进入可执行状态的。

第 14 章　启动系统

14.1　启动的流程

UNIX V6 的启动流程如下所示（图 14-1 ）。

1. 保存在 ROM 中的引导装入程序，将位于根磁盘块编号 0 处的引导程序 读取至内存地址 0 的位置并执行。

2. 引导程序从根磁盘的文件系统将`/unix` 、`/rkunix` 等内核程序的主程序 读取至内存地址 0 的位置并执行。

3. 内核对系统进行初始化。

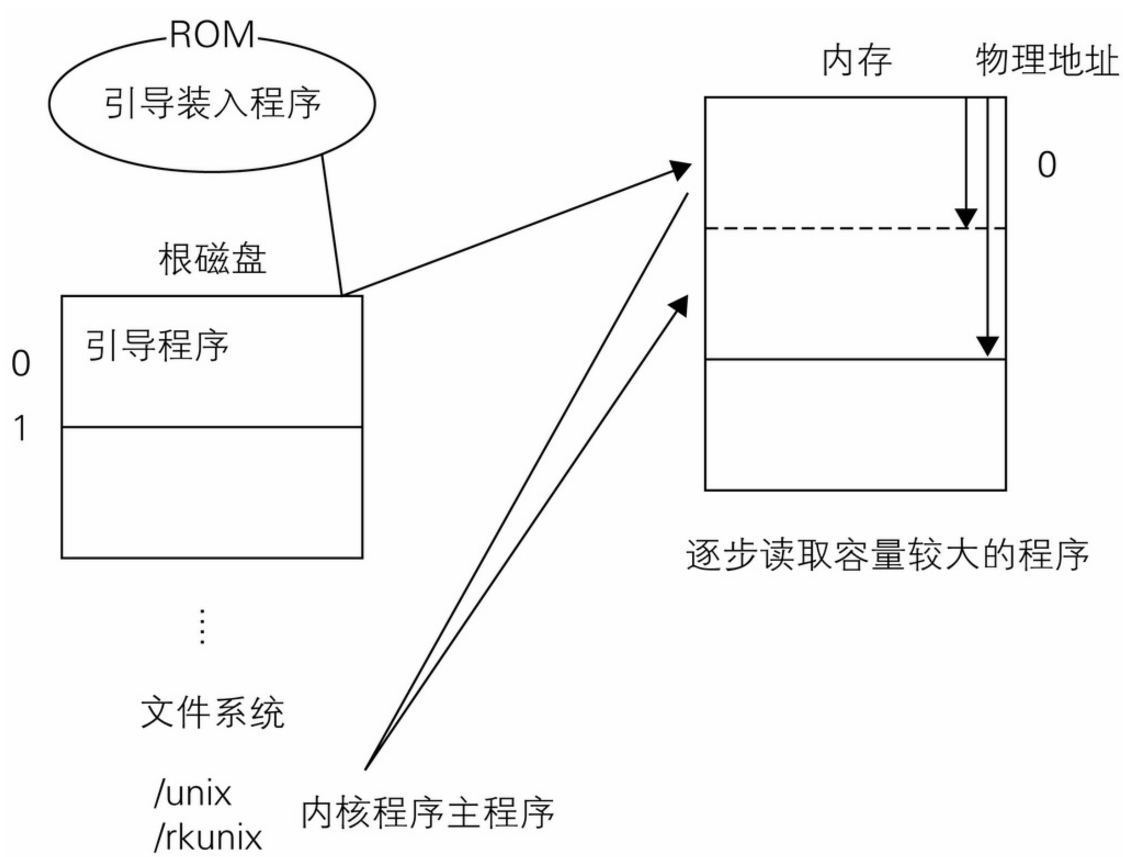


图 **14-1**　启动的流程

为了启动系统需要将内核程序读取至内存，但是内核程序约有 40KB，这对（当时）容量很小的 ROM 而言是无法接受的。因此，需要首先执行可以容纳在 ROM 中的简单程序，再读取较大较复杂的引导程序……如此逐步读取容量较大的程序。

系统管理者在执行 `/etc/mkfs` 构筑文件系统时，将引导程序放置于块设备编号为 0 的块中。

代码清单 14-1 给出的是位于内核主程序低位地址处的代码。第 1 行的 `. = 0^.` 表示当程序被读取至内存时，此处为地址 0 的位置。首先执行的指令是第 2 行的 `br 1f` 。`br` 为分支（跳转）指令，`1f` 表示跳转至前方第 1 个标签的位置。跳转至标签后执行 `jmp start` ，将控制权转交给 `start` 。

代码清单 **14-1**　启动（**low.s)**

```
. = 0^.
    br 1f
    4

略

. = 40^.
.globl    start, dump
1:        jmp    start
          jmp    dump
```

start

`start`（代码清单 14-2 ）进行下述处理。

- 对内核 APR 进行初始化
- 激活 MMU 使之有效
- 执行 `main()`

执行 `start` 时尚未激活 MMU，处于需要直接处理物理地址的状态。内核可以使用 16 比特的数据（地址），但物理地址为 18 比特，因此只能访问物理地址的低位 16 比特的区域（0~0177777 ）。此外，PSW 的当前模式为内核模式（00 ）。

`start` 首先进行内核空间的设定 。对内核 APR 进行如表 14-1，图 14-2 所示的设定。

- APR0~5 被赋予物理地址 0~0137777 的区域。此区域的物理地址 = 虚拟地址，内核程序被置于该区域的起始位置。
- APR6 被赋予位于内核程序后方长度为 1KB 的区域。此区域被分配给 `proc[0]` 的 PPDA（`user` 结构体和内核栈）。APR6 表示执行进程的 PPDA，当执行进程发生切换时，PAR6 的值也将发生切换。
- APR7 被赋予物理地址的最高位地址 0760000~0777777。此区域被分配给 PDP-11/40 及周边设备的寄存器。内核通过操作此区域的内存，对 PDP-11/40 及周边设备进行操作。
- 栈指针被设定为指向 APR6 对应区域的末尾。

内核 APR 中，除了 PAR6 以外，在系统运行时，值是不会发生变化的。

表 **14-1**　内核 **APR** 的初始设定

APR	**PAR**	**PDR**	虚拟地址	物理地址	备注
0	0	077406	0~017777	0~017777	RW，128 个块（以 64 字节为单位）
1	0200	077406	020000~037777	020000~037777	RW，128 个块
2	0400	077406	040000~057777	040000~057777	RW，128 个块
3	0600	077406	060000~077777	060000~077777	RW，128 个块

4	01000	077406	0100000~0117777	0100000~0117777	RW，128 个块
5	01200	077406	0120000~0137777	0120000~0137777	RW，128 个块
6	与内核主程序的长度相关	007406	0140000~0141777	与内核主程序的长度相关	RW，16 个块，供执行程序的 PPDA 使用
7	07600	077406	0160000~0177777	0760000~0777777	RW，128 个块，供 I/O 使用

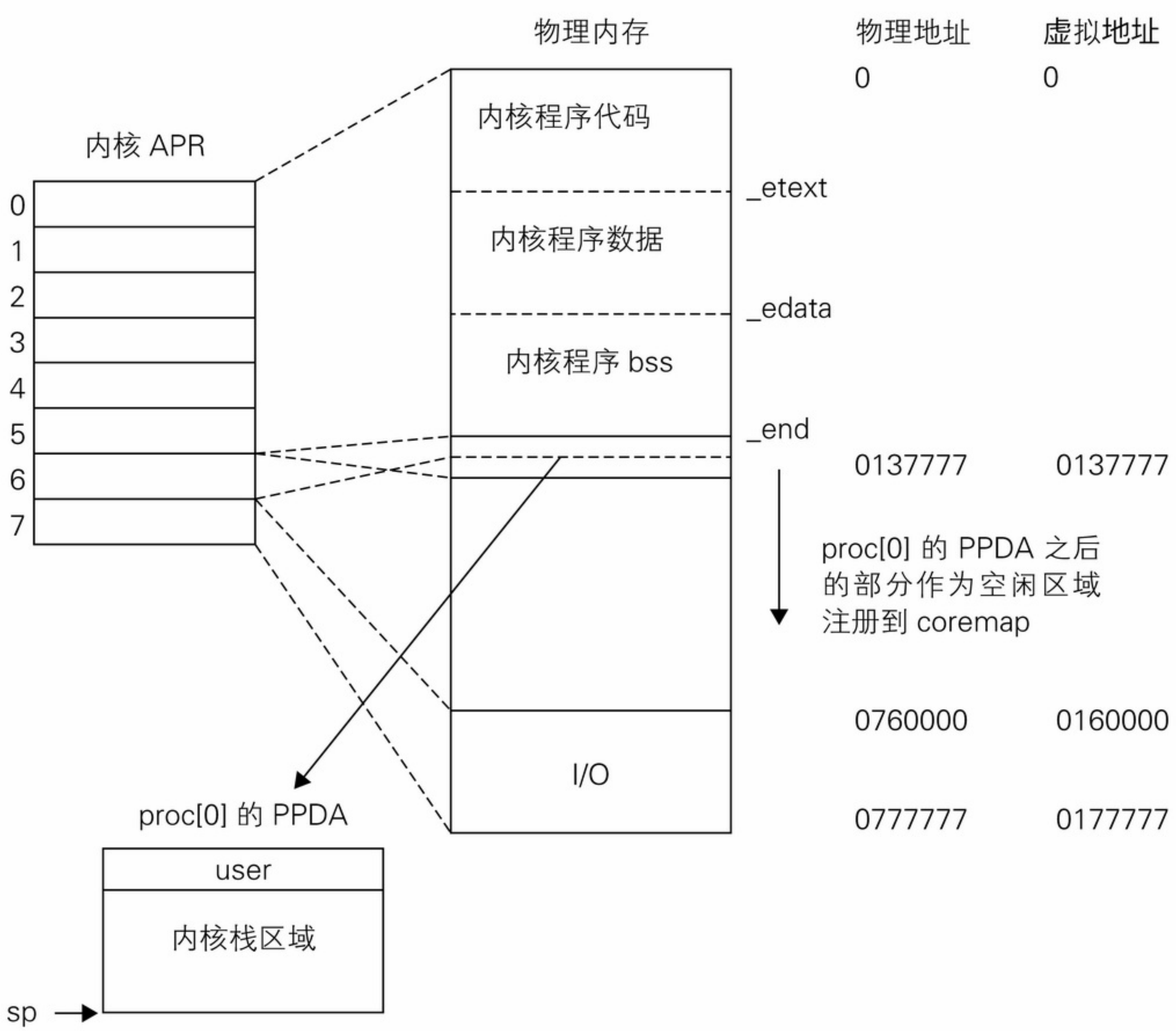


图 **14-2**　内核内存空间

然后，设置 SR0，激活 MMU 使之有效。从此时开始，可将虚拟地址变为物理地址，进程的虚拟地址空间处于有效状态。

将内核程序的 bss 区域和 `proc[0]` 的 PPDA 清 0，并执行 `main()`。执行 `main()` 时，将前一模式设定为用户模式。

代码清单 **14-2　start**（**m40.s**）

```
1 .globl    start, _end, _edata, _main
2 start:
3     bit $1,SSR0
4     bne start
5     reset
6
7 / 内核APR0-5的初始化
8     mov    $KISA0,r0
9     mov    $KISD0,r1
10    mov    $200,r4
11    clr    r2
12    mov    $6,r3
13 1:
14    mov    r2,(r0)+
15    mov    $77406,(r1)+
16    add    r4,r2
17    sob    r3,1b
18
19 / 内核APR6的初始化
20    mov    $_end+63.,r2
21    ash    $-6,r2
22    bic    $!1777,r2
23    mov    r2,(r0)+
24    mov    $usize-1\<8|6,(r1)+
25
26 / 内核APR7的初始化
27    mov    $IO,(r0)+
28    mov    $77406,(r1)+
29
30 / 初始化栈指针，并激活MMU使之有效
31    mov    $_u+[usize*64.],sp
32    inc    SSR0
33
34 / 清除BSS区域
35    mov    $_edata,r0
36 1:
37    clr    (r0)+
38    cmp    r0,$_end
39    blo    1b
40
41 / 将proc[0]的user和内核栈区域清0
42    mov    $_u,r0
43 1:
44    clr    (r0)+
45    cmp    r0,$_u+[usize*64.]
46    blo    1b
47
48 / 调用main()
49    mov    $30000,PS
50    jsr    pc,_main
51    mov    $170000,-(sp)
52    clr    -(sp)
53    rtt
```

3~4　如果 MMU 已处于有效状态，则进入循环使处理不继续进行。SSR0 表示 SR0 寄存器的地址（代码清单 14-3）。SR0 的最低比特位为 1 时表示 MMU 有效。

代码清单 **14-3　SSR0**（**m40.s**）

```
SSR0    = 177572
```

8~9　KISA0 、KISD0 分别表示内核 APR0 的 PAR、PDR 的地址（代码清单 14-4）。

代码清单 **14-4　KISA0** 和 **KISD0**（**m40.s)**

```
1 KISA0    = 172340
2 KISD0    = 172300
```

20~22　_end 表示内核程序 bss 区域的地址下限。_etext 、_edata 、_end 分别指向程序的代码区域、数据区域、bss 区域的地址下限，在程序编译时，由链接器将其保存到符号表中。

对赋予 PAR6 的值进行计算。因为 APR 以 64 字节为单位管理物理内存的，所以将 _end 以 64 字节为单位向上取整，并右移 6 比特，保留低位的 10 比特并将其他比特位清 0。

24　设定 PDR6 的值。

27　IO 为赋予 PAR7 的值（代码清单 14-5）。

代码清单 **14-5　IO**（**m40.s)**

```
1 IO    = 7600
```

31　设定栈指针的值。

32　将 SR0 的最低比特位置 1 使 MMU 有效。此时开始可以将虚拟地址变换为物理地址。

35~39　将内核程序的 bss 区域（从 _edata 到 _end ）清 0。

42~46　将 proc[0] 的 user 结构体、内核栈区域清 0。

49~50　将前一模式设置为用户模式并执行 main() 。

main()

main() 进行下列处理（代码清单 14-11 ）。

- 初始化内存
- 初始化交换空间
- 初始化时钟装置
- 生成 proc[0]
- 初始化 I/O 资源
- 生成 proc[1]

将物理内存的 proc[0] 的 PPDA 之后的部分清 0，并作为空闲领域追加至 coremap 。将实际空闲区域长度和参数 MAXMEM （代码清单 4-7 ）中较小的值设为表示空闲区域长度（以 64 字节为单位）的 maxmem （代码清单 14-6 ）。

代码清单 **14-6　maxmem**（**system.h)**

```
int    maxmem;
```

代码清单 **14-7　MAXMEM**（**param.h)**

```
#define    MAXMEM    (64*32)
```

通过 `mfree()` 分配交换空间中的空闲区域，并将其赋予 `swapmap`。将以 `swplo` 块为起点、长度为 `nswap` 的块作为交换空间注册到 `swapmap`（代码清单 14-8）。

代码清单 **14-8　swplo** 和 **nswap**（**conf.c)**

```
int    swplo    4000;
int    nswap    872;
```

时钟装置可在电源频率时钟装置或可编程时钟装置中选择一种，两者都可在使用时选择电源频率时钟装置。

`proc[0]` 通过手动（并非 `newproc()`）创建。`proc[0]` 为系统使用的内核进程，用于执行 `sched()` 和 `swtch()`。当前目录被设定为根磁盘的根目录。

执行 `cinit()`、`binit()`、`iinit()`，分别对字符设备缓冲区、块设备缓冲区、`inode[]` 进行初始化。

执行 `newproc()` 生成新的进程（`proc[1]`）。`proc[1]` 为执行"/etc/init"的用户进程，将执行容纳在 `icode[]` 中的指令集（代码清单 14-9）。`icode[]` 的内容如果用 C 语言编写的话将如代码清单 14-10 所示。

代码清单 **14-9　icode[]**（**ken/main.c)**

```
int    icode[]
{
    0104413,    /* sys exec; init; initp */
    0000014,
    0000010,
    0000777,    /* br . */
    0000014,    /* initp: init; 0 */
    0000000,
    0062457,    /* init: </etc/init\0> */
    0061564,
    0064457,
    0064556,
    0000164,
};
```

代码清单 **14-10　icode[]** 的 **C** 语言版

```
char *init = "/etc/init";
execl(init, init, 0);
while(1);
```

代码清单 **14-11　main()（ken/main.c）**

```
main()
{
    extern schar;
    register i, *p;

    updlock = 0;
    /* 初始化内存和交换空间 */
    i = *ka6 + USIZE;
    UISD->r[0] = 077406;
    for(;;) {
        UISA->r[0] = i;
        if(fuibyte(0) < 0)
            break;
        clearseg(i);
        maxmem++;
        mfree(coremap, 1, i);
        i++;
    }
    if(cputype == 70)
    for(i=0; i<62; i=+2) {
        UBMAP->r[i] = i<<12;
        UBMAP->r[i+1] = 0;
    }
    printf("mem = %l\n", maxmem*5/16);
    maxmem = min(maxmem, MAXMEM);
    mfree(swapmap, nswap, swplo);

    /* 初始化时钟装置 */
    UISA->r[7] = ka6[1];
    UISD->r[7] = 077406;
    lks = CLOCK1;
    if(fuiword(lks) == -1) {
        lks = CLOCK2;
        if(fuiword(lks) == -1)
            panic("no clock");
    }

    /* 创建进程0 */
    proc[0].p_addr = *ka6;
    proc[0].p_size = USIZE;
    proc[0].p_stat = SRUN;
    proc[0].p_flag =| SLOAD|SSYS;
    u.u_procp = &proc[0];
    *lks = 0115;

    /* 初始化资源 */
    cinit();
    binit();
    iinit();
    rootdir = iget(rootdev, ROOTINO);
    rootdir->i_flag =& ~ILOCK;
    u.u_cdir = iget(rootdev, ROOTINO);
    u.u_cdir->i_flag =& ~ILOCK;

    /* 创建进程1 */
    if(newproc()) {
        expand(USIZE+1);
        estabur(0, 1, 0, 0);
        copyout(icode, 0, sizeof icode);
        return;
    }
    sched();
}
```

8~18　将 i 设定为 proc[0] 的 PPDA 区域后方的地址。ka6 已在 m40.s 中被设定为内核 APR6 的地址（KISA6 ），通过 *ka6 可以取得 APR6 的值（代码清单 14-12）。

代码清单 **14-12　ka6**（**conf/m40.s)**

```
_ka6:    KISA6
```

UISD 和 UISA 指向用户 APR0 的 PAR，PDP 的地址（代码清单 14-13 ）。

代码清单 **14-13　UISD** 和 **UISA**（**seg.h)**

```
#define    UISD    0177600
#define    UISA    0177640
```

fuibyte() 用来复制位于前一模式虚拟地址空间中长度为 1 字节的数据。由于前一模式为用户模式，因此 fuibyte(0) 从分配给用户 APR0 的页的起始位置开始复制数据。

循环递增用户 PAR0 的值，并执行 fuibyte(0) ，如果成功，将相应地址追加至 coremap 。如果出错（-1。参照地址无应答等情况），即可判断已到达物理地址的终点。

29~30　将用户 APR7 设定为内核 APR7 的值，使 fuiword() 可以访问 I/O 区域。

31~36　CLOCK1 和 CLOCK2 分别表示电源频率时钟装置和可编程时钟装置的的寄存器地址（代码清单 14-14）。对 CLOCK1 执行 fuiword() ，如果失败则选择 CLOCK2 。如果对 CLOCK2 执行 fuiword() 也失败的话，则执行 panic() 终止处理。

代码清单 **14-14　CLOCK**（**ken/main.c)**

```
#define    CLOCK1    0177546
#define    CLOCK2    0172540
```

30~43　设定 proc[0] 。将 proc.p_addr 设定为内核 PAR6 的值，并为 proc.p_flag 设置 SLOAD 和 SSYS 标志位。SSYS 表示系统进程，不作为交换处理的对象。

56　生成 proc[1] 。

62　proc[0] 执行 sched() 。由于不是交换处理的对象，因此 proc[0] 进入睡眠状态并将控制权转交给 proc[1] 。

57~60　proc[1] 将 icode[] 复制到位于地址 0 处的数据区域的起始位置。执行 return 后返回至 start 调用 main() 的位置（代码清单 14-15），如表 14-2 所示，将值压入栈后执行 rtt 指令。rtt 指令从栈顶将 0 读取至 pc ，将 0170000 读取至 PSW。在当前模式和前一模式为用户模式的状态下，从用户空间的地址 0 处开始执行指令。因为 icode[] 的内容已被复制到地址 0 处，所以将启动执行 /etc/init 的处理。

代码清单 **14-15　start** 中的相关部分（**conf/m40.s)**

```
mov    $170000,-(sp)
clr    -(sp)
```

```
53     rtt
```

表 **14-2**　**start** 中执行 **rtt** 时的栈状态

sp	值
→	0 → pc
	0170000 → PSW

/etc/init

`/etc/init` 为用户程序，在此只做简要介绍（图 14-3），详细内容请参照 UPM（8）。

1. 对已注册的终端创建进程。

2. 进程进入待机状态等待拨号连接终端和用户登录。用户登录后启动 shell 程序。

3. `/etc/init` 在这之后进入无限循环状态，对失去父进程的进程进行持续处理。

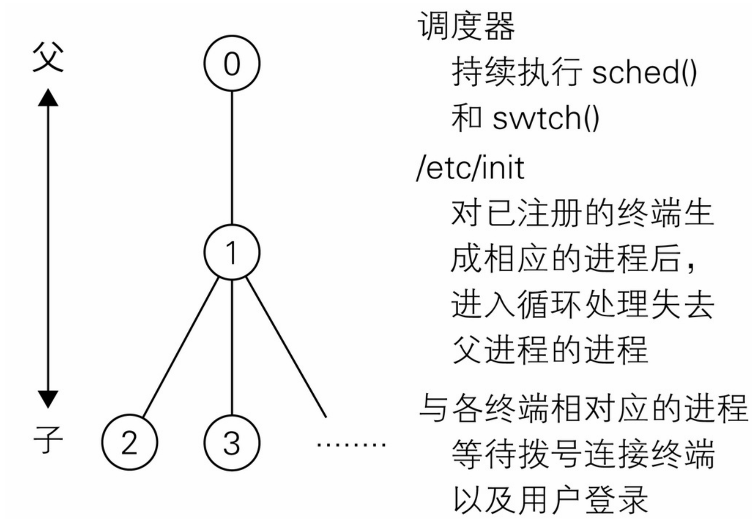


图 **14-3**　执行 **init** 后进程的状态

14.2　小结

- UNIX V6 启动时，按照引导装入程序→引导程序→内核主程序的顺序，逐步执行容量较大的程序。

- 启动时的处理包括：使 MMU 有效，设定内核 APR，初始化时钟装置，初始化内存和交换空间，初始化 I/O 资源，生成 `proc[0]` 和 `proc[1]` 等。

- `proc[0]` 为系统进程，作为调度器执行。

- `proc[1]` 用于执行 `/etc/init` 。

附录　参考资料等

A.1　参考文献、网站

参考书籍、网站等

- [1] The Unix Tree（Minnie's Home Page）
  http://minnie.tuhs.org/cgi-bin/utree.pl
  可在此阅览 UNIX V6 的代码。

- [2] Unix V6 Manuals
  http://man.cat-v.org/unix-6th/
  可在此阅览 UNIX V6 的手册。

- [3] *Lions' Commentary on Unix 6th Edition* / John Lions 著 / Annabooks/Rtc Books 出版 / ISBN 978-1573980135
  由 John Lions 执笔，介绍 UNIX V6 的书籍[1]。

[1] 中文版：《莱昂氏 UNIX 源代码分析》，机械工业出版社 2000 年7月出版，尤晋元译。——译者注

- [4] *Commentary on the Sixth Edition UNIX Operating System*
  http://www.lemis.com/grog/Documentation/Lions/
  可以免费获取在 USENET alt.folklore.computers newsgroup 公开的此版本的 Lions 书。

- [5] *Modern Operating Systems* / Andrew S. Tanenbaum 著 / Prentice Hall 出版 / ISBN 978-0136006633
  http://www.pearsonhighered.com/educator/product/Modern-Operating-Systems/9780136006633.page
  经常作为大学课程的教科书，是一本很经典的书[2]。

[2] 中文版：《现代操作系统》，机械工业出版社 2009 年7月出版，陈向群、马洪兵译。——译者注

- [6] *Design of the UNIX Operating System* / Maurice J. Bach 著 / Prentice Hall 出版 / ISBN 978-0132017992
  使用了很多的插图和模拟语言，以初期的 UNIX 为中心介绍 UNIX 的书籍[3]。

[3] 中译本名为《UNIX 操作系统设计》，机械工业出版社2000 年4 月出版，陈葆珏译。——译者注

- [7] The Computer History Simulation Project
  http://simh.trailing-edge.com/
  simh 模拟器的主页。该模拟器用于模拟包含 PDP-11/40 在内的较古老的处理器。

- [8] Computer History Wiki 上关于安装 simh 的网页
  http://gunkies.org/wiki/Installing_Unix_v6_(PDP-11)_on_SIMH
  对使用 simh 启动 UNIX V6 的方法进行了说明。

- [9] WikiPedia PDP-11
  http://en.wikipedia.org/wiki/PDP-11
  WikiPedia 上 PDP-11 页面的相关内容。对理解框架及编译器很有帮助。

论文、手册等

- [10] The Unix Time-Sharing System / Dennis M. Ritchie，Ken Thompson 著
  介绍了处于萌芽时期 UNIX 的整体概要。

- [11] UNIX Implementation / Ken Thompson 著
  从实现的角度对处于萌芽期的 UNIX 进行介绍。

- [12] SETTING UP UNIX - Sixth Edition
  介绍 UNIX V6 的环境构筑和启动方法。对希望了解用户空间（userland）系统程序的读者会有帮助。

- [13] The UNIX I/O System / Dennis M. Ritchie 著
  介绍了处于萌芽期的 UNIX 的 I/O 处理。

- [14] PDP11/40 processor handbook / DEC 公司
  PDP-11/40 的使用手册。在调查处理器的详细数据、指令集时是必备的资料。

- [15] PDP Peripherals Handbook / DEC 公司
  PDP 周边设备的使用手册。

- [16] C Reference Manual / Dennis M. Ritchie 著
  UNIX V6 使用的被称为 pre K&R C 的 C 语言使用手册。

- [17] UNIX Assembler Reference Manual / Dennis M. Ritchie 著
  UNIX V6 使用的编译器的参考手册。

Lions'Commentary on UNIX 读书会会员的网站

在笔者也曾加入过的 Lions’Commentary on UNIX 读书会会员制作的网站之中，选择一些刊载了与 UNIX V6 相关话题的网站在此进行介绍。由于每个话题都很有深度，因此当对 UNIX V6 内核的理解陷入困境，或是希望进一步了解相关信息时，这些网站一定会有很大的帮助。

- [18] Lions’Commentary on UNIX 读书会 https://sites.google.com/site/lionscommentaryonunixreading/
  Lion 书读书会的主页。刊载了 UNIX V6、V7 相关网站的链接，以及读书会举办的封闭式活动的记录。

- [19] A boring diary
  http://d.hatena.ne.jp/takahirox/
  笔者的博客。可视为本书的基础。刊载了本书中未涉及的细节、参加 Lions 书读书会的笔记、利用 UNIX V6 模拟器进行实验的记录等内容，并公开了笔者自己开发的 UNIX V6 模拟器。

- [20] 2238 Club
  http://www.tom-yam.or.jp/2238/
  浜田直树先生的网站。刊载了多种有助于阅读 UNIX V6 代码的资料，以及 UNIX V6、PDP-11 的手册和相关资料。对本书未介绍的，UNIX V6 中最难理解的 backup() 也进行了说明。

- [21] 山本英雄先生的 slideshare
  http://www.slideshare.net/magoroku15/
  山本英雄先生的 slideshare。公开了很多 UNIX V6 的说明资料。入门篇易于初学者起步，容易理解，与本书配套使用一定会加深理解。还介绍了使用 simh 确认 UNIX V6 内核处理的方法，以及在 FPGA 上执行 UNIX V6 的资料。

- [22] Plan9 日记
  http://d.hatena.ne.jp/oraccha/
  高野了成先生的博客。介绍了包括 UNIX V6、PDP-11 的引导程序在内的内容。笔者在对 UNIX V6 关联资料进行调查的时候总是最终查看这个博客。

- [23] 七志的开发日记（旧）
  http://d.hatena.ne.jp/n7shi/
  七志先生的博客。他开发了 UNIX V6 的 a.out 解释器。虽然七志先生已经开设了新的博客，但在此还是要介绍一下刊载了 UNIX V6 相关内容的旧博客。

- [24] 丰岛语录
  http://toyoshim.blogspot.jp/

丰岛隆志先生的博客。在 Lions 书读书会的笔记中包含了很敏锐的观点。此外，还介绍了重新构筑 UNIX V6 内核并加入自制的系统调用的方法。

- [25] 电脑羊（Android Dream）
http://xiangcai.at.webry.info/
在山本英雄先生制作的介绍 UNIX V7 的 Ustream[4] 中经常以学生身份登场的大野徹先生的博客。他将 Ustream 的内容经过整理并上传至博客。由于 UNIX V6 与 V7 并没有很大的差别，因此对阅读 V6 的代码一定会有很大帮助。

> 4 有名的视频分享网站。——译者注

A.2　pre K&R C

本节针对熟悉现代 C 语言的读者，介绍 UNIX V6 使用的 pre K&R C 中较难理解的部分。

运算代入

运算代入符不是 `ep=`，而是 `=ep`（代码清单 A-1）。

代码清单 **A-1**　加法、减法代入

```
a =+ 1;
a =- 1;
a =* 1;
a =/ 1;
a =% 1;
a =>> 1;
a =<< 1;
a =& 1;
a =^ 1;
a =| 1;
```

无名结构体

pre K&R C 允许使用无名结构体，而且此结构体的成员可以对任意变量进行使用。使用无名结构体可对映射到内存周别设备的寄存器，或对 2 字节数据的高位和低位字节进行操作。

代码清单 A-2 演示了对 PSW 的读取操作。`PS` 为 PSW 映射到（内核地址空间）的内存地址。由于 pre K&R C 不支持型变换（cast），因此无法直接使用 `PS` 读取 PSW 的值。

此处使用具有成员变量 `int integ` 的无名结构体，通过 `PS->integ`，用 int 型从 `PS` 表示的地址中（强制）读取长度为 1 个字的数据。

代码清单 **A-2**　操作 **PSW**

```
/* param.h */
#define PS    0177776

struct
{
    int    integ;
};

/* ken/slp.c */
/* in sleep() */
s = PS->integ;
```

下面再给出一个例子。在代码清单 A-3 中，通过 RKADDR->rkcs 可以访问 RK 磁盘寄存器中映射至地址 0177404 处的的 rkcs 寄存器。

代码清单 **A-3**　对 **rkcs** 的操作

```
/* dmr/rk.c */
#define    RKADDR    0177400

struct {
    int rkds;
    int rker;
    int rkcs;
    int rkwc;
    int rkba;
    int rkda;
};

/* in rkstart() */
RKADDR->rkcs = RESET|GO;
```

有限的变量类型

pre K&R C 中没有 unsigned 型，希望使用 unsigned 型时可以用指针代替。此外，也没有 long 型，最大只能处理长度为 1 个字（16 比特）的数据。因此在处理更大的数据时，需要使用数组或多个变量。

例如，file 结构体成员变量的文件偏移量 f_offset，使用 char*（16 字节）的数组表现 32 字节的数据。f_offset[0] 对应高位 16 比特，f_offset[1] 对应低位 16 比特。通过这种方式进行运算时，高位 16 比特与低位 16 比特的数据必须分开计算，显得十分繁琐。因此 UNIX V6 定义了专用的函数，dpadd() 用于加法，dpcmp() 用于减法（比较）。在将实际读写量追加至文件偏移量的处理中也使用了这些函数（代码清单 A-4 ）。

代码清单 **A-4**　使用数组表现 **32** 比特的数据

```
/* file.h */
struct    file
{
    char    f_flag;
    char    f_count;
    int     f_inode;
    char    *f_offset[2];
} file[NFILE];

/* ken/sys2.c */
/* in rdwr() */
dpadd(fp->f_offset, u.u_arg[1]-u.u_count);
```

后记

非常感谢大家能完整地阅读本书。在将内核代码阅读一遍之后，如果您能够感受到在“前言”中提到的那些效果，笔者会十分高兴。如果感觉对有些内容的理解不够深入，建议复习一遍，或者参考相关的设计书及论文。因为已经对整体内容有了大致的了解，相信再次阅读会更容易理解。

对那些已经相当了解 UNIX V6 内核的读者来说，今后的学习之路也有很多选择，例如下述方法。

- 改写 UNIX V6 的内核源代码，进行各种实验
- 阅读并理解本书未做说明的内核源代码和设备驱动源代码
- 跟踪用户空间（userland）的系统程序
- 理解周边设备及 PDP-11/40 的详细设计
- 逐步将学习的范围扩大至 UNIX V7、BSD 等较新的操作系统
- 开发 PDP-11 的模拟器，尝试在其中运行 UNIX V6
- 在理解内核处理的基础上，对开发中的软件进行微调

阅读完内核源代码之后，相信在大家的脑海中都会萌发出一些新的想法。衷心希望大家能够追随这些想法，继续技术之路的探索，活跃在计算机的世界当中。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks

3、 25岁前一定要读的25本书

4、 有生之年，你一定要看的25部外国纯文学名著

5、 有生之年，你一定要看的20部中国现当代名著

6、 美国亚马逊编辑推荐的一生必读书单100本

7、 30个领域30本不容错过的入门书

8、 这20本书，是各领域的巅峰之作

9、 这7本书，教你如何高效读书

10、 80万书虫力荐的“给五星都不够”的30本书

关注“幸福的味道”微信公众号，即可查看对应书单和得到电子书

也可以在我的网站（周读）www.ireadweek.com 自行下载

更多书单，请关注微信公众号：一种思路