HZ BOOKS
华章科技

本书对操作系统内核的驾驭能力和深刻理解程度达到世界顶尖级水平，
是一本能真正引导你深入理解Linux内核设计思想的经典著作。

华章原创精品

Linux内核设计的艺术

图解Linux操作系统架构设计与实现原理

The Art of Linux Kernel Design

新设计团队 著

机械工业出版社
China Machine Press

The Art of Linux Kernel Design

对于操作系统而言，稳定且可靠地运行是最重要的。现行技术方案是将用户进程与用户进程之间、用户进程与操作系统之间进行分离，操作系统可以管理用户进程，但是用户进程之间不能互相干预，更不能“侵入”操作系统内核。然而这一切是如何实现的呢？用户程序和操作系统都是程序，使用的是同一个CPU、同一套指令集、同一个内存条，为什么会有这么大的区别呢？

绝妙之处在于操作系统的设计指导思想：主奴机制！

操作系统的设计者依托硬件，在主奴机制设计思想的指导下，把操作系统内核与用户进程间的关系设计为“主子”与“奴才”的关系。该指导思想贯穿整个操作系统设计的始终，涉及操作系统的方方面面。以内存管理为例，操作系统内核可以访问所有的内存空间，包括用户进程的内存空间，而用户进程则不能访问其他用户进程的内存空间，更不可能访问操作系统内核的内存空间。更有甚者，用户进程甚至不知道自身的代码以及所访问的内存究竟在物理内存中的什么位置……可以说，操作系统就是围绕着“主奴机制”这个设计指导思想实现的。

本书不仅在主奴机制等宏观的设计指导思想方面有独到的见解，而且对操作系统源代码的实现原理和编程技巧等微观内容的讲解与分析也细致入微。例如：

- 为什么操作系统的内核代码不是加载在0x00000的位置？
- 为什么 jmpi 0, 8中的“8”不能简单地当作阿拉伯数字“8”看待？
- 为什么用C语言写的操作系统却不是从main函数开始执行？
- 为什么调用main函数时用return而不用call？

……

所有这些在同类书中可能都不曾提及的问题在本书中都有精彩的论述。

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

上架指导：计算机／操作系统／Linux
ISBN 978-7-111-34744-6
9 787111 347446
定价：79.00元

华章原创精品

Linux 内核设计的艺术

图解Linux操作系统架构设计与实现原理

The Art of Linux Kernel Design

新设计团队 著

机械工业出版社
China Machine Press

关于Linux内核的书已经不计其数，但这本书却是独树一帜的，它的内容代表着Linux内核研究成果的世界顶尖级水平，它在世界范围内首次提出并阐述了操作系统设计的核心指导思想——主奴机制，这是所有操作系统研究者的一笔宝贵财富。本书可能也代表着同类图书的顶尖水平，是一本真正能引导我们较为容易地、极为透彻地理解Linux内核的经典之作，也可能是当前唯一能从本质上指引我们去设计和开发拥有自主知识产权的操作系统的著作。它的出版也许会成为Linux内核研究领域的一个里程碑事件。

本书的最大特点是它的写作方式和内容组织方式，与同类书完全不同。它在深刻地分析了传统讲解方法的利弊之后，破旧立新，从认知学的角度开创了一种全新的方式。以操作系统的真实运行过程为主线，结合真实的内核源代码、349幅精确的内核运行时序图和具有点睛之妙的文字说明，对操作系统从开机加电到系统完全准备就绪的整个过程进行了系统而完整地分析，深刻地揭示了其间每一个动作的设计意图和实现原理，完美地再现了操作系统设计者的设计思路。阅读本书就如同跟随着操作系统设计者一起去思考，我们会在阅读的过程中发现Linux内核设计的精妙，会发现原来处处都"暗藏玄机"，哪怕是一行很短的代码。

本书在所有细节上都力求完美。为了保证知识的准确性，操作系统运行过程中的每个动作都经过了严格的考证；为了让我们真正理解Linux内核的原理，它突破传统，以Linux的真实运行过程为主线进行讲解；为了做到真正易于理解，创新性地使用了图解的方式，精心绘制了349幅分辨率600dpi的时序图，图中表现的运行时结构和状态与操作系统实际运行时的真实状态完全吻合；为了提高阅读体验，本书采用了双色印刷，以便于我们更清楚地观察每一幅图中的细节。

图书在版编目（CIP）数据

Linux内核设计的艺术：图解Linux操作系统架构设计与实现原理 / 新设计团队著. —北京：机械工业出版社，2011.5

ISBN 978-7-111-34744-6

Ⅰ. L… Ⅱ. 新… Ⅲ. Linux操作系统－图解 Ⅳ. TP316-64

中国版本图书馆CIP数据核字（2011）第088008号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：陈佳媛
北京京师印务有限公司印刷
2011年7月第1版第2次印刷
186mm×240mm • 27.75印张
标准书号：ISBN 978-7-111-34744-6
定价：79.00元

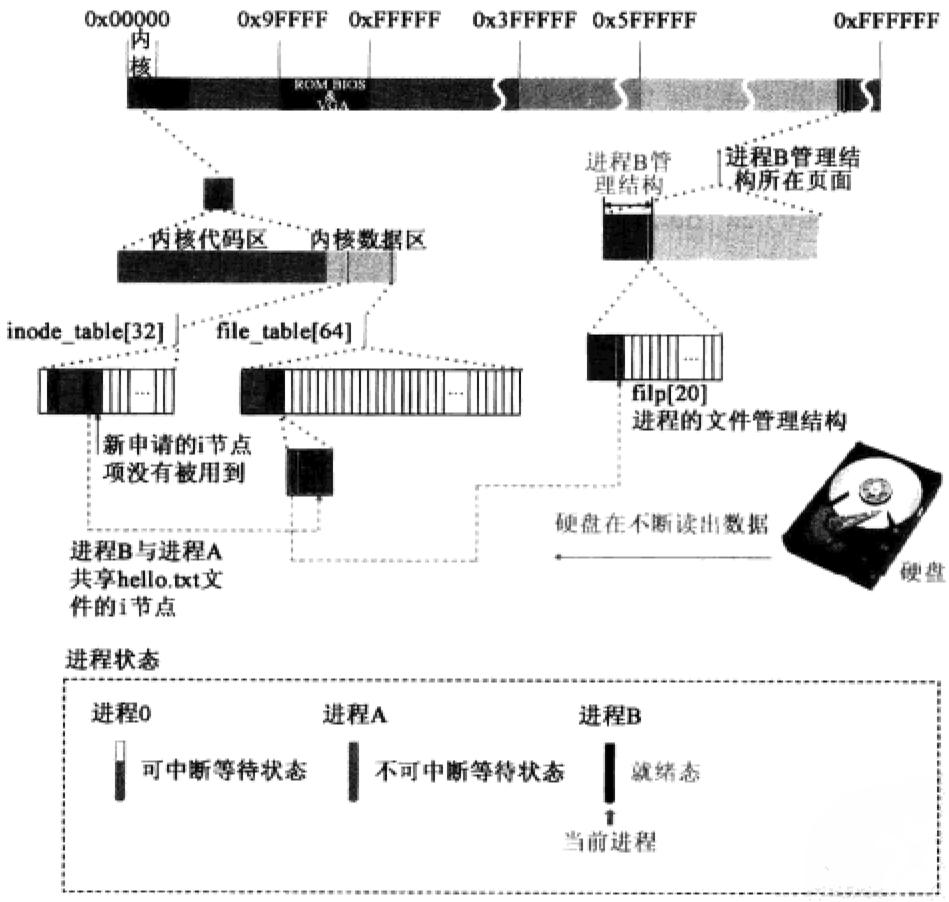
凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

本书导读

一、本书约定

1. 图的约定

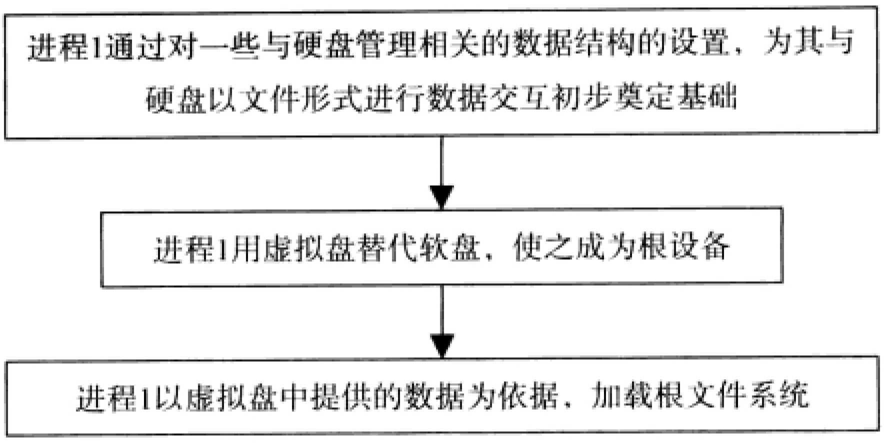
以本书中第 6 章中的图 6-5 为例来说明本书中关于图的约定。



这张图描绘了操作系统运行时的一个场景，最上面的一个横条表示的是内存，其上的 0x00000、0x9FFFF 等都是内存地址值。在内存示意图上，用颜色和灰阶表示了操作系统的内核、缓冲区、虚拟盘区和主内存区。从内存示意图中通过虚线向下引出了更多的横条，它们是在对内存中的局部区域进行放大。这些局部区域就是这个场景中需要关注的内存的细节，它们表现了涉及的代码和数据在内存中的精确位置。虚线箭头表示数据的传递和挂接情况。我们在图中绘制了硬盘的状态，表示此时硬盘正在不断地读出数据。另外，在图的最下方，用三个小竖条表示了三个进程的状态，灰色表示等待状态，黑色表示就绪态，小竖条中的空白表示进程已用的时间片。

对于比较复杂的章节，为了能让读者在阅读之前对该章节的内容有一个总体的了解，我们绘制了程序架构图。以第 2 章 2.3 节加载根文件系统的程序架构图（如下所示）为例，这张图表示

该小节的内容分为三个阶段，框图中写出了每个阶段的主要意思。



2. 源代码的约定

以本书第 6 章 6.1.5 小节的内容所对应的源代码为例来说明本书对源代码的约定。

```
// 代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ......
    while (inode < NR_INODE+inode_table)
    {
        if (inode->i_dev != dev || inode->i_num != nr)
        ......
        inode->i_count++;
        ......
        if (empty)
            iput(empty);
        return inode;
    }
    ......
}
```

第 1 行是该部分源代码在 linux0.11 内核源代码中的路径，通过这个路径，读者可以很容易地找到相关的函数。由于篇幅有限，本书没有贴出全部的源代码，只选取了读者最需要关注的部分。

3. “小贴士”的约定

对于那些可能会影响读者阅读的术语，本书在它们第一次出现时用“小贴士”进行了简单的解释。例如：

小贴士

8259A 中断控制器：8259A 是专门为了对 8085A 和 8086/8088 进行中断控制而设计的芯片，它是可以用程序控制的中断控制器。单个的 8259A 能管理 8 级向量优先级中断。在不增加其他电路的情况下，最多可以级联成 64 级的向量优先级中断系统。

4. “点评”的约定

本书对一些深刻难懂、能反映操作系统精妙设计思想和需要读者特别注意的知识点进行了深度点评，以帮助读者更好地理解本书的内容。例如：

点评

为什么没有最先调用 main 函数？

学过 C 语言的都知道，用 C 语言设计的程序都有一个 main 函数，而且是从 main 函数开始执行的。linux 0.11 的代码是用 C 语言编写的，奇怪的是，为什么在操作系统启动时先执行的是三个由汇编写成的程序，然后才开始执行 main 函数？为什么不是像我们熟知的 C 程序那样，从 main 函数开始执行呢？

……

二、阅读建议

首先，强烈建议读者按照本书的目录顺序进行阅读！此外，还有四点需要注意：

第一，建议读者不要因为某一个地方没有看懂就停下来长时间思考，而应该继续阅读，到后续章节中去寻找答案。比如，第 2 章的内容几乎涉及了整个操作系统的重要理论知识，对于非重点的这些内容更深入细致的描述，我们都安排在了后续的章节中讲解。

第二，注意阅读的层次。对于比较复杂和难懂的章节，我们在开篇处对本章的主要思想和想要讲解的内容进行了宏观的介绍，而且还给出了直观的程序架构图。这部分内容虽然篇幅不大，但是对理解后续的内容非常重要，请读者特别注意。

第三，请一定将文字说明、图和源代码三者结合起来阅读。操作系统的设计思想和实现原理都蕴含在源代码中，需要读者去分析、推断、猜测和验证，不同水平的读者能看到和领悟到的内容会有所不同。文字能很好地对源代码进行辅助说明，而且能尽可能地详细，但是理解起来不够直接。图能很直观地展示一些核心的内容，但是仅凭图能读懂多少信息，这也与读者的水平有很大的关系。说明文字、图和源代码三者各有优势，而且能互相弥补彼此的不足，阅读时如果能有效地结合三者，一定事半功倍。

第四，操作系统毕竟是非常难的，如果有些内容一时没有看懂，这是很正常的。对于不太容易理解的内容，建议读者整章整节地反复阅读。操作系统最难和最有价值的是它的整体，整体看明白了，局部才能清楚。哪怕对整体只有一个朦胧的了解，对理解操作系统都是有意义的。

三、如何获得 Linux 0.11 源代码

Linux 0.11 的源代码是开源的，您可以从赵炯先生维护的网站 http://www.oldlinux.org/Linux.old/Linux-0.11/ 上获取本书所涉及的全部源代码。

四、搭建阅读本书的实践操作环境

我们强烈建议您一边阅读本书，一边对照着 Linux 的源代码进行实践，只有这样才能真正理解 Linux 操作系统。在开始之前，你需要搭建一个 Linux 0.11 系统环境，具体方法见附录 A。

前　言

很早就有一个想法，做中国人自己的、有所突破、有所创新的操作系统、计算机语言及编译平台。

我带领的“新设计团队”（主要由中国科学院研究生院已毕业的学生组成）在实际开发自己的操作系统的过程中，最先遇到的问题就是如何培养学生真正看懂 Linux 操作系统的源代码的能力。开源的 Linux 操作系统的源代码很容易找到，但很快就会发现，培养学生看懂 Linux 操作系统的源代码是一件非常非常困难的事。

操作系统的代码量通常都是非常庞大的，动辄几百万行，即使是浏览一遍也要很长时间。比庞大的代码量更让学习者绝望的是操作系统有着极其错综复杂的关系。看上去，代码的执行序时隐时现，很难抓住脉络。代码之间相互牵扯，相互勾连，几乎无法理出头绪。更谈不上理解代码背后的原理、意图和思想。

对于学生而言，选择从源代码的什么地方开始分析本身就是一个难题。通常，学生有两种选择：一种是从 main 函数，也就是从 C 语言代码的总入口开始，沿着源代码的调用路线一行一行地看下去，学生很快就会发现源代码的调用路线莫名其妙地断了，但直觉和常识告诉他操作系统肯定不会在这个地方停止，一定还在继续运行，但却不知道后续的代码在哪里，这种方法很快就走进了死胡同；另一种则是从某一模块入手，如文件系统，但这样会无形中切断操作系统源码之间复杂的关系，如文件系统与进程管理的关系，文件系统与内存管理的关系，等等。使学生孤立地去理解一个模块，往往只能记住一些名词和简单概念，难以真正理解操作系统的全貌。用学生的话讲，他们理解的操作系统变成了“文科”的操作系统。

由于操作系统是底层系统程序，对应用程序行之有效的调试和跟踪等手段对操作系统的源代码而言，几乎无效。就算把每一行源代码都看懂了，对源代码已经烂熟于心，知道这一行是一个 for 循环，那一行是一个调用……但仍然不知道整个代码究竟在做什么，以及起什么作用，更不知道设计者的意图究竟是什么。

我们在操作系统的课程上学习过进程管理、内存管理、文件系统等基础知识，但是这些空洞的理论在一个实际的操作系统中是如何实现的却不得而知。他们在源代码中很难看出进程和内存之间有什么关联，内核程序和用户程序有什么区别，为什么要有这些区别。也很难从源代码中看清楚，我们实际经常用到的操作，比如打开文件，操作系统在其中都做了哪些具体的工作？想在与常见的应用程序的编程方法有巨大差异的、晦涩难懂的、浩瀚如海的操作系统底层源代码中找

到这些问题的答案，似乎比登天还难。

对熟悉操作系统源代码的学生而言，他们也知道像分页机制这样知识点，但是未必能够真正理解隐藏在机制背后的深刻意义。

这些都是学生在学习 Linux 操作系统源代码时遇到的实际问题。中国科学院研究生院的学生应该是年轻人中的佼佼者，他们遇到的问题可能其他读者也会遇到。我萌发了一个想法，虽然学生的问题早已解决，但是否可以把他们曾经在学习、研发操作系统的过程中遇到的问题和心得体会得拿出来供广大读者分享？

当时，针对学生的实际问题，我的解决方法是以一个真实的操作系统为例，让学生理解源代码，把操作系统在内存中的运行时状态画出图来。实践证明，这个方法简单有效。

现在我们把这个解决方案体现在这本书中。就是以一个真实的操作系统的实际运行为主线；以图形、图像为核心，突出描述操作系统在实际运行过程中内存的运行时结构；强调站在操作系统设计者的视角，用体系的思想方法，整体把握操作系统的行为、作用、目的和意义。

在全书的讲解过程中，我们不仅详细分析了源代码、分析了操作系统的执行序，我们还特别分析了操作系统都作了哪些“事”，并且把“事”与“事”之间的关系和来龙去脉，这些“事”意味着什么，为什么要做这些“事”，这些“事”背后的设计思想是什么……都做了非常详细且深入的分析。

更重要的是，对于所有重要的阶段，我们几乎都用图解的方式把操作系统在内存中的实际运行状态精确地表示了出来。我们用 600dpi 的分辨率精心绘制了 349 张图，图中表现的运行时结构和状态与操作系统实际运行的真实状态完全吻合。每一条线、每一个色块、每一个位置、每一个地址及数字都经过了我们认真反复地推演和求证，并最终在计算机上进行了核对和验证。看了这些绘制精美的图后，在读者的头脑中就不再是一行行、一段段枯燥的、令人眩晕的源代码，而是立体呈现的一件件清晰的“事”，以及这些“事”在内存中直截了当、清晰鲜活的画面。用这样的方法讲解操作系统是本书的一大特色。理解这些图要比理解源代码和文字容易得多，毫不夸张地说，只要你能理解这些图，你就理解了操作系统的 80%，这时你可以自豪的说，你比大多数用别的方法学过操作系统的人的水平都要高出一大截。

作者和机械工业出版社的编辑作了大量的检索工作，就我们检索的范围而言，这样的创作方法及具有这样特色的操作系统专著在世界范围都是第一次。

我们分三个部分来讲解 Linux 操作系统：第一部分由第 1 章和第 2 章组成，分析了从开机加电到操作系统启动完成并进入怠速状态的整个过程；第二部分由第 3 章、第 4 章、第 5 章、第 6 章、第 7 章组成，讲述了操作系统进入系统怠速后，在执行用户程序的过程中，操作系统和用户进程的实际运行过程和状态；第三部分由第 8 章组成，阐述整个 Linux 操作系统的设计指导思想，本章内容是从微观到宏观的回归。

第一部分，我们详细讲解了开机加电启动 BIOS，通过 BIOS 加载操作系统程序，对主机的初始化，打开保护模式和分页，调用 main 函数，创建进程 0、进程 1、进程 2 以及 shell 进程，并且具备以文件的形式与外设交互。

第二部分，我们设计了几个尽可能简单又有代表性的应用程序，并以这些程序的执行为引导，详细讲解了安装文件系统、文件操作、用户进程与内存管理、多个进程对文件的操作以及进程间通信。

我们将操作系统的原理自然而然地融入到了讲解真实操作系统的实际运行过程中。在读者看来，操作系统原理不再是空对空的、“文科”概念的计算机理论，而是既有完整的、体系的理论，又有真实、具体、实际的代码和案例，理论与实际紧密耦合。

第三部分是全书水平最高的部分，我们尝试从操作系统设计者的视角讲解操作系统的设计指导思想。详细阐述了主奴机制以及实现主奴机制的三项关键技术：保护和分页、特权级、中断，分析了保障主奴机制实现的决定性因素——先机，还详细讲解了缓冲区、共享页面、信号、管道的设计指导思想。希望帮助读者用体系的思想理解、把握、驾驭整个操作系统以及背后的设计思想和设计意图。

在本书中，我们详细讲解了大家在学习操作系统的过程中可能会遇到的每一个难点，如 main 函数中的 pause() 调用，虽然已经找不到后续代码，但该调用结束后，程序仍然执行的原因是：中断一经打开，进程调度就开始了，而此时可以调度的进程只有进程 1，所以后续的代码应该从进程 1 处继续执行……

我们还对一些读者不容易理解和掌握的操作系统特有的底层代码的编程技巧作了详细的讲解。如用模拟 call 的方法，通过 ret 指令“调用”main 函数……

……

总之，我们所做的一切努力就是想真正解决读者遇到的实际问题和难题，给予读者有效的帮助。我们盼望即使是刚刚考入大学的学生也有兴趣和信心把这本书读下去；我们同样希望即使是对操作系统源代码很熟悉的读者，这本书也能给他们一些不同的视角、方法和体系性思考。

这本书选用的操作系统源代码是 Linux 0.11。对为什么选用 Linux 0.11 而不是最新版本，赵炯先生有过非常精彩的论述，我们认为赵先生的论述是非常到位的。

我们不妨看一下 Linux 最新的版本 2.6，代码量大约在千万行这个量级，去掉其中的驱动部分，代码量仍在百万行这个量级。一个人一秒钟看一行，一天看 8 小时，中间不吃、不喝、不休息，也要看上几个月，很难想象如何去理解。

就算我们硬要选用 Linux 2.6，就算我们写上 3000 页（书足足会有十几厘米厚），所有的篇幅都用来印代码，也只能印上不到十分之一的代码。所以，即使是这么不切实际的篇幅，也不可能整体讲解 Linux 2.6。读者会逐渐明白，对于理解和掌握操作系统而言，真正有价值的是整体、是体系，而不是局部。

Linux 0.11 的内核代码虽然只有约两万行，但却是一个实实在在、不折不扣的现代操作系统，因为它具有现代操作系统最重要的特征——支持实时多任务，所以必然支持保护和分页……而且它还是后续版本的真正的始祖，有着内在的、紧密的传承关系。读者更容易看清设计者最初的、最根本的设计意图和设计指导思想。

Linux 0.11 已经问世 20 年了，被世人广为研究和学习。换一个角度看，要想在众人熟悉的

事物和领域讲出新意和特色，对作者来说也是一个强有力的挑战。

这本书能够顺利出版，我们首先要感谢机械工业出版社华章公司的副总经理温莉芳女士以及其他领导，是他们的决心和决策成就了这本书，并且在几乎所有方面给予了强有力的支持。特别令人感动的是他们主动承担了全部的出版风险，同时给予了作者最好的条件，让我们看到一个大出版社的气度和风范。

其次，我们还要感谢的是机械工业出版社华章公司的编辑杨福川。杨先生的鉴赏力和他的事业心以及他对工作认真负责的态度为这本书的出版打开了大门。杨先生对读者的理解以及他的计算机专业素养使得他有能力对这本书给予全方位的指导和帮助，使我们对这本书整体修改了6次，使之更贴近读者，可读性更好。

我们还要感谢我们和杨福川先生共同的朋友张国强先生和杨缙女士。

最后，我们要感谢我们的家人和朋友。是他们坚定的支持才使得我们的整个团队能够拒绝方方面面、形形色色的诱惑，放弃普遍追求的短期利益，在常人难以想象的艰苦条件下，长时间专注于操作系统、计算机语言、编译器、计算机体系结构等基础性学科的研究。认认真真、踏踏实实、不为名利，做了一点实在、深入的工作，有了一点能凑合拿得出手的10年积累，以及一支敢想、敢干、敢打、敢拼、不惧世界顶级强敌的队伍。这些是本书的基础。

杨力祥

中国科学院研究生院

2011年5月

目　录

第7章 IPC问题 ……358

第 1 章

从开机加电到执行 main 函数之前的过程

从开机到 main 函数的执行分三步完成，其目的是实现从启动盘加载操作系统程序，完成执行 main 函数所需要的准备工作。第一步，启动 BIOS，准备实模式下的中断向量表和中断服务程序；第二步，从启动盘加载操作系统程序到内存，加载操作系统程序的工作就是利用第一步中准备的中断服务程序实现的；第三步，为执行 32 位的 main 函数做过渡工作。本章将详细分析这三步在计算机中是如何完成的，以及每一步在内存中都做了些什么。

小贴士

实模式（Real mode）是 Intel 80286 和之后的 80x86 兼容 CPU 的操作模式。实模式的特性是一个 20 位的存储器地址空间 (2^20=1048576，即 1MB 的存储器可被寻址)，可以直接通过软件的方式访问 BIOS 以及周边硬件，没有硬件支持的分页机制和实时多任务的概念。从 80286 开始，所有的 80x86 CPU 的开机状态都是实模式；8086 等早期的 CPU 只有一种操作模式，类似于实模式。

1.1 启动 BIOS，准备实模式下的中断向量表和中断服务程序

相信大家都知道计算机必须安装一个所谓“操作系统”的软件才能使用，否则计算机将是一个毫无生命力的冰冷的硬家伙。在我们为计算机安装了操作系统后，当你按下计算机电源按钮的那一刻，计算机机箱传来了嗡嗡的声音，这时你感觉到计算机开始启动工作了。然而，在计算机的启动过程中，操作系统底层与计算机硬件之间究竟做了哪些复杂的交互动作？下面我们将根据操作系统实际的启动和运行过程对这个问题进行逐步的剖析和讲解。

计算机的运行是离不开程序的。然而，加电的一瞬间，计算机的内存中，准确地说是 RAM 中，空空如也，什么程序也没有。软盘里虽然有操作系统程序，但 CPU 的逻辑电路被设计为只能运行内存中的程序，它没有能力直接从软盘运行操作系统[⊖]。如果要运行软盘中的操作系统，

⊖ Linus 写 Linux 0.11 是在 1991 年底，那时，很多计算机是从软盘启动的，他为 Linux 0.11 设计的系统启动盘也是软盘。

必须将软盘中的操作系统程序加载到内存（RAM）中。

小贴士

RAM（Random Access Memory）：随机存取存储器。常见的内存条就是一类 RAM，它的特点是加电状态下可任意读、写，断电后信息消失。

问题是在 RAM 中什么程序也没有的时候，谁来完成加载软盘中操作系统的任务呢？

答案是：BIOS

1.1.1 BIOS 的启动原理

在了解 BIOS 是如何将操作系统程序加载到内存中之前，我们先来了解一下 BIOS 程序自身是如何启动的。从我们使用计算机的经验得知：要想执行一个程序，必须在窗口中双击它，或者在命令行界面中输入相应的执行命令。从计算机底层机制上讲，这其实是在一个已经运行起来的操作系统的可视化界面或命令行界面中执行一个程序。但是，在开机加电的一瞬间，内存中什么程序也没有，没有任何程序在运行，不可能有操作系统，更不可能有操作系统的用户界面。我们无法人为地执行 BIOS 程序，那 BIOS 程序又是由谁来执行的呢？

答案是：0xFFFF0！！！

从体系的角度看，不难得出这样的结论：既然软件方法不可能执行 BIOS，那就只能靠硬件方法完成了。

从硬件角度看，Intel 80x86 系列的 CPU 可以分别在 16 位实模式和 32 位保护模式下运行。为了兼容，也为了解决最开始的启动问题，Intel 将所有 80x86 系列的 CPU（包括最新型号的 CPU）的硬件都设计为加电即进入 16 位实模式状态运行。同时，还有一点非常关键，即将 CPU 硬件逻辑设计为加电瞬间强行将 CS 的值置为 0xFFFF，IP 的值置为 0x0000，这样 CS:IP 就指向 0xFFFF0 这个地址位置，如图[⊖] 1-1 所示。从图 1-1 中可以清楚地看到，0xFFFF0 指向了 BIOS 的地址范围。

小贴士

IP/EIP（Instruction Pointer）：指令指针寄存器，存在于 CPU 中，记录将要执行的指令在代码段内的偏移地址，它与 CS 组合即为将要执行的指令的内存地址。实模式为绝对地址，指令指针为 16 位，即 IP；保护模式下为线性地址，指令指针为 32 位，即 EIP。

⊖ 本书中的大部分图都是依照计算机实际运行时的内存真实状态，严格按比例以 600dpi 分辨率精确绘制的，所以有些内存区域因为体积比较小，在图中占的位置也比较小，请大家阅读时仔细辨认。

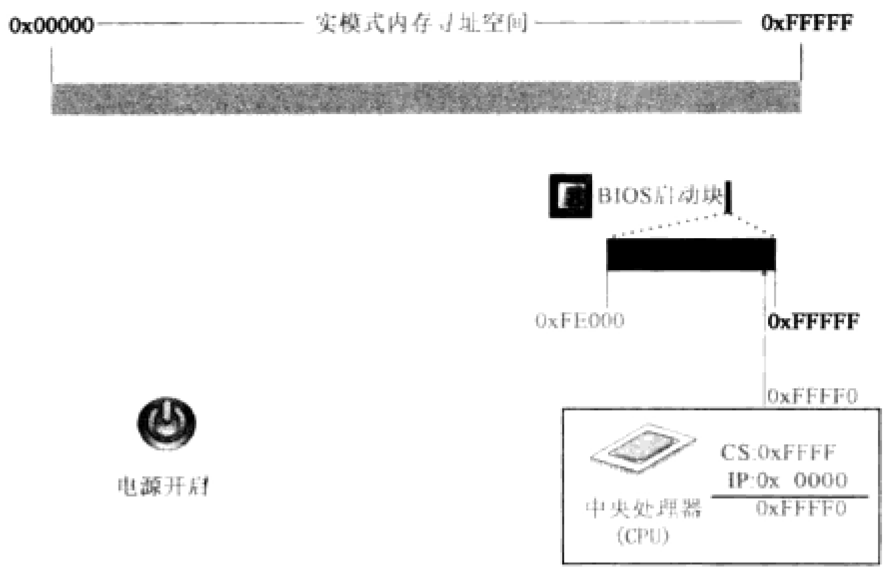


图 1-1 启动时 BIOS 在内存中的状态及初始执行位置

小贴士

CS（Code Segment Register）：代码段寄存器，存在于 CPU 中，指向 CPU 当前执行代码在内存中所在的区域。

注意，这是一个纯硬件完成的动作！如果此时这个位置没有可执行代码，那就什么也不用说了，计算机就此死机。反之，如果这个位置有可执行代码，计算机将从这里的代码开始，沿着后续程序一直执行下去。

BIOS 程序的入口地址恰恰就是 0xFFFF0！也就是说，BIOS 程序的第一条指令就设计在这个位置上。

1.1.2 BIOS 在内存中加载中断向量表和中断服务程序

BIOS 程序的代码量并不大，却非常精深，需要对计算机的整个硬件体系结构非常熟悉才能将其看明白。要想把 BIOS 是如何运行的讲清楚，也得写很厚一本书，这显然超出了本书的主题和范围。我们的主题是操作系统，所以只会将与启动操作系统有直接关系的内容讲解一下。

BIOS 程序被固化在计算机主机板上的一块很小的 ROM 芯片里。通常，不同的主机板所用的 BIOS 也有所不同，就启动部分而言，各种类型的 BIOS 的基本原理大致相似。为了便于大家理解，我们选用的 BIOS 程序只有 8KB，所占地址段为 0xFE000 ～ 0xFFFFF，如图 1-1 所示。现在 CS:IP 已经指向了 0xFFFF0 这个位置，这意味着 BIOS 开始启动了。随着 BIOS 程序的执行，屏幕上会显示显卡的信息、内存的信息……说明 BIOS 程序在检测显卡、内存……这期间，有一项对启动（boot）操作系统至关重要的工作，那就是 BIOS 在内存中建立中断向量表和中断服务程序。

小贴士

ROM（Read Only Memory）：只读存储器，现在通常用闪存芯片做 ROM。虽然闪存芯片在特定的条件下是可写的，但在谈到主机板上存储 BIOS 的闪存芯片时，业内人士把它看作 ROM。

ROM 有一个特性，就是断电之后仍能保存信息，这一点与硬盘类似。

BIOS 程序在内存最开始的位置（即：0x00000）用 1KB 的内存空间（0x00000 ～ 0x003FF）构建中断向量表，并在紧挨着它的位置用 256 字节的内存空间构建 BIOS 数据区（0x00400 ～ 0x004FF），在大约 56KB 以后的位置（0x0E2CE）加载了 8KB 左右的与中断向量表相应的若干中断服务程序，图 1-2 中精确地标注了这些位置。

小贴士

一个容易计算的方法：0x00100 是 256 个字节，0x00400 就是 4×256 字节 = 1024 字节，即 1KB。因为是从 0x00000 开始计算，所以 1KB 的高地址端不是 0x00400，而是 0x00400−1，也就是 0x003FF。

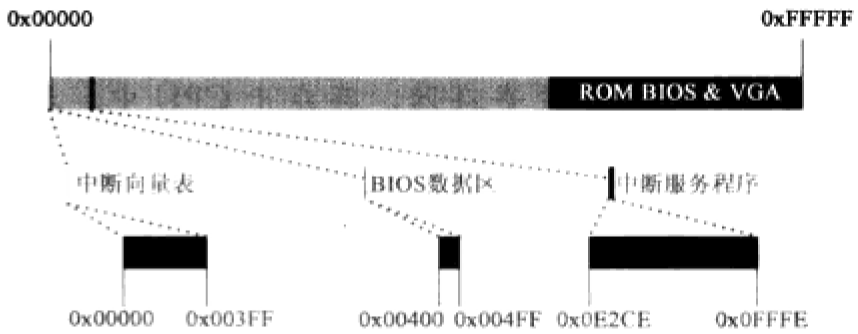


图 1-2　BIOS 在内存中加载中断向量表和中断服务程序

中断向量表中有 256 个中断向量，每个中断向量占 4 个字节，其中两个字节是 CS 的值，两个字节是 IP 的值，每个中断向量都指向一个具体的中断服务程序。

下面将详细讲解后续程序是如何利用这些中断服务程序把系统内核程序从软盘加载至内存的。

小贴士

中断 INT（INTerrupt）：顾名思义中断就是中途打断一件正在进行中的事。其最初的意思是：外在的事件打断正在执行的程序，转而执行处理这个事件的特定程序，处理结束后，回到被打断的程序继续执行。现在，可以先将中断理解为一种技术手段，在这一点上与 C 语言的函数调用有些类似。

中断对操作系统来说意义重大，后面我们还会深入讨论。

1.2　加载操作系统内核程序并为保护模式做准备

从现在开始就要执行真正的 boot 操作了，即把软盘中的操作系统程序加载至内存。对于 Linux 0.11 操作系统而言，计算机将分三批逐次加载操作系统的代码。第一批由 BIOS 中断 int 0x19 把第一扇区 bootsect 的内容加载到内存；第二批和第三批在 bootsect 的指挥下，分别把其后

的四个扇区和随后的 240 个扇区的内容加载至内存。

1.2.1 加载第一部分代码——引导程序（bootsect）

按照我们使用计算机的经验，如果在开机的时候马上按 Del 键，屏幕上会显示一个 BIOS 画面，可以在里面设置启动设备。现在我们基本上都是将硬盘设置为启动盘，Linux 0.11 是 1991 年设计的操作系统，那时常用的启动设备是软驱以及其中的软盘。站在体系结构的角度看，从软盘启动和从硬盘启动的基本原理和机制是类似的。

经过执行一系列 BIOS 代码之后，计算机完成了自检等操作（这些与我们讲的启动操作系统没有直接的关系，读者不必关心），由于我们把软盘设置为启动设备，计算机硬件体系结构的设计与 BIOS 联手操作，会让 CPU 接收到一个 int 0x19 中断，CPU 接收到这个中断后，会立即在中断向量表中找到 int 0x19 中断向量。我们在图 1-3 的左下方可以看到 int 0x19 中断向量在内存中所在的准确位置，这个位置几乎紧挨着内存的 0x00000 位置。

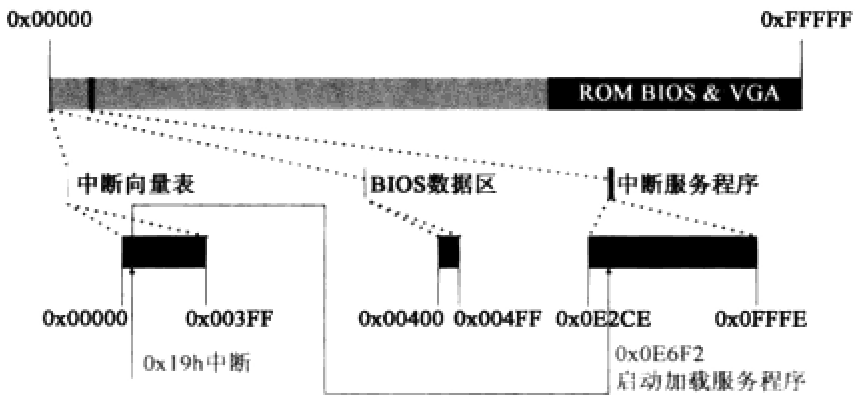


图 1-3 响应 int 0x19 中断

接下来，中断向量把 CPU 指向 0x0E6F2，这个位置就是 int 0x19 相对应的中断服务程序的入口地址，即图 1-3 中所示“启动加载服务程序”的入口地址。这个中断服务程序的作用就是把软盘的第一个扇区中的程序（512B）加载到内存中的指定位置。这个中断服务程序的功能是 BIOS 事先设计好的，代码是固定的，与 Linux 操作系统无关，无论 Linux 0.11 的内核是如何设计的，这段 BIOS 程序所要做到就是“找到软盘”并“加载第一扇区”。其余的它什么都不知道，也不必知道。

小贴士

中断向量表（Interrupt Vector Table）：实模式中断机制的重要组成部分，表中记录所有中断号对应的中断服务程序的内存地址。

中断服务程序（Interrupt Services）：通过中断向量表的索引对中断进行响应服务，是一些具有特定功能的程序。

按照这个简单、“生硬”的规则，int 0x19 中断向量所指向的中断服务程序（即启动加载服务程序）将软驱 0 号磁头对应盘面的 0 磁道 1 扇区的内容拷贝至内存 0x07C00 处。我们可以在

图 1-4 的左边看到第一扇区加载的具体位置。

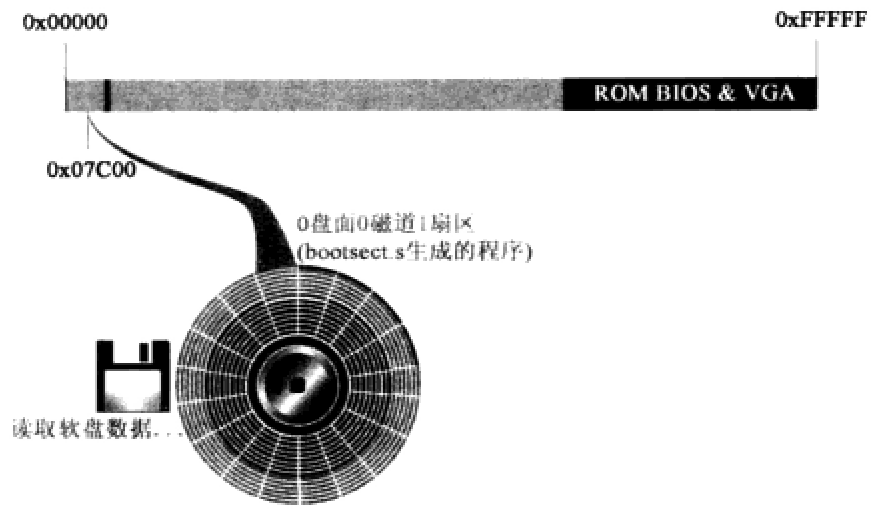


图 1-4　把软盘第一个扇区中的程序加载到内存中的指定位置

这个扇区里的内容就是 Linux 0.11 操作系统的引导程序，也就是我们将要讲解的 bootsect，其作用就是陆续把软盘中的操作系统程序载入内存。这样制作的第一扇区就称为启动扇区（boot sector）。第一扇区程序的载入标志着 Linux 0.11 操作系统中的代码即将发挥作用了。

这是非常关键的动作，从此计算机开始与软盘上的操作系统程序产生关联。第一扇区中的程序是由 bootsect.s 中的汇编程序汇编而成（以后简称 bootsect），这是计算机自开机以来，内存中第一次有了 Linux 操作系统自己的代码，虽然只是启动代码。

至此，已经把第一批代码 bootsect 从软盘载入到计算机的内存了。下面的工作就是执行 bootsect 把软盘的第二批和第三批代码载入内存。

点评

注意，BIOS 程序固化在主机板上的 ROM 中，是根据具体的主机板而不是根据具体的操作系统设计的。

理论上，计算机可以安装任何适合其安装的操作系统，既可以安装 Windows，也可以安装 Linux。不难想象每个操作系统的设计者都可以设计出一套自己的操作系统启动方案，而操作系统和 BIOS 通常是由不同的专业团队设计和开发的，为了能协同工作，必须建立操作系统和 BIOS 之间的协调机制。

与已有的操作系统建立一一对应的协调机制虽然麻烦，但尚有可能，难点在于如何与未来的操作系统建立协调机制，现行的方法是“两头约定”和“定位识别”。

对操作系统（这里指 Linux 0.11）而言，“约定”操作系统的设计者必须把最开始执行的程序“定位”在启动扇区（软盘中的 0 盘面 0 磁道 1 扇区），其余的程序可以依照操作系统的设计顺序加载在后续的扇区中。

对 BIOS 而言，“约定”在接到启动操作系统的命令后，“定位识别”只从启动扇区把代码加载到 0x07C00 (BOOTSEG) 这个位置（参见 Seabios 0.6.0/Boot.c 文件中的 boot_disk 函数）。至于这个扇区中是否是启动程序，以及是什么操作系统，则不闻不问并一视同仁。如果不是启动代

码，只会提示错误，其余是用户的责任，与 BIOS 无关。

这样构建协调机制的好处是站在整个体系的高度，统一设计和统一安排，简单而有效。只要 BIOS 和操作系统的生产厂商开发的所有系统版本全部遵循此机制的约定，就可以各自灵活地设计出具有自己特色的系统版本。

1.2.2 加载第二部分代码——setup

1. bootsect 对内存的规划

现在，BIOS 已经把 bootsect（也就是引导程序）载入内存了，它的作用就是把第二批和第三批程序陆续加载到内存中。为了把第二批和第三批程序加载到内存中的适当位置，bootsect 首先做的工作就是规划内存。

通常，我们是用高级语言编写应用程序，这些程序是在操作系统的平台上运行的。我们只管写高级语言的代码和数据，至于这些代码和数据在运行的时候放在内存的什么地方，是否会相互覆盖，我们都不用操心，因为操作系统和高级语言的编译器替我们做了大量的看护工作，以确保不会出错。现在我们讨论的是操作系统本身，启动代码使用的是汇编语言，没有高级语言编译器替其提供保障，只有靠操作系统的设计者把内存的安排想清楚，确保无论操作系统如何运行，都不会出现代码与代码、数据与数据、代码与数据之间相互覆盖的情况。为了更准确地理解操作系统的运行机制，我们必须清楚操作系统的设计者是如何规划内存的。

在实模式状态下，寻址的最大范围是 1MB，为了规划内存，bootsect 首先设计了如下的代码：

```
// 代码路径：boot/bootsect.s

SETUPLEN = 4                          ! nr of setup-sectors
BOOTSEG  = 0x07c0                     ! original address of boot-sector
INITSEG  = 0x9000                     ! we move boot here-out of the way
SETUPSEG = 0x9020                     ! setup starts here
SYSSEG   = 0x1000                     ! system loaded at 0x10000 (65536)
ENDSEG   = SYSSEG + SYSSIZE           ! where to stop loading
```

这些源代码的作用就是对后续操作所涉及的内存位置进行设置，包括将要加载的 setup 程序的扇区数（SETUPLEN）和被加载到的位置（SETUPSEG）、启动扇区被 BIOS 加载的位置（BOOTSEG）和将要移动到的新位置（INITSEG）、内核（kernel）被加载的位置（SYSSEG）、内核的末尾位置（ENDSEG）和根文件系统设备号（ROOT_DEV），这些位置在图 1-5 中都明确地标注了出来。设置这些位置就是为了确保将要载入内存的代码和已经载入内存的代码及数据各在其位，互不覆盖，并且它们各自有够用的内存空间。大家在后续的章节会逐渐看到内存规划的意义和作用。

从现在起，我们的头脑中要时刻牢记这样一个概念：操作系统的设计者要全面、整体地考虑内存的规划。我们会在后续的章节中不断地了解到，精心安排内存是操作系统设计者时时刻刻都要关心的事。我们带着这样的观念继续讲解 bootsect 程序的执行。

图 1-5　实模式下的内存使用规划

2. 复制 bootsect

接下来，bootsect 启动程序将它自身（全部的 512B 内容）从内存 0x07C00（BOOTSEG）处复制至内存 0x90000（INITSEG）处，这个动作和目标位置如图 1-6 所示。

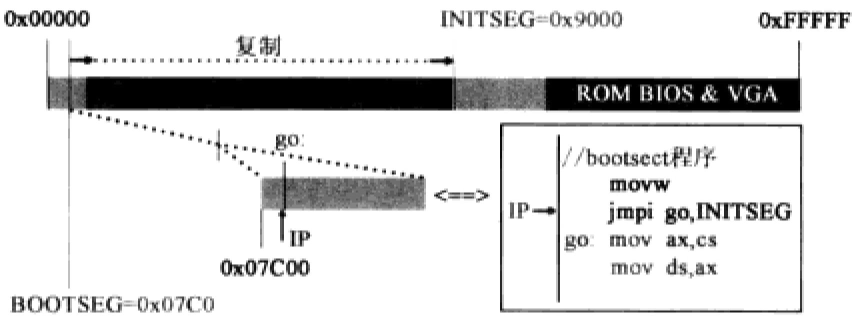


图 1-6　bootsect 复制自身

执行这个操作的代码如下：

```
// 代码路径：boot/bootsect.s

mov ax, #BOOTSEG
mov ds, ax
mov ax, #INITSEG
mov es, ax
mov cx, #256
sub si, si
sub di, di
rep
movw
```

在这次复制过程中，ds（0x07C0）和 si（0x0000）联合使用，构成了源地址 0x07C00；es（0x9000）和 di（0x0000）联合使用，构成了目的地址 0x90000（参看图 1-6 左边），而 mov cx, #256 这一行循环控制量，提供了需要复制的“字”数（一个字为两个字节），256 个字正好是 512 字节，即第一扇区的字节数。

通过代码我们还可以看出，图 1-5 提到的 BOOTSEG 和 INITSEG 现在开始发挥作用了。注

意，此时 CPU 的段寄存器（CS）指向 0x07C0 (BOOTSEG)，即原来 bootsect 程序所在的位置。

点评

由于“两头约定”和“定位识别”的作用，所以 bootsect 在开始时“被迫”加载到 0x07C00 位置。现在将其自身移至 0x90000 处，说明操作系统开始根据自己的需要安排内存了。

bootsect 复制到新位置后，bootsect 会执行下面这行加粗的代码：

```
// 代码路径：boot/bootsect.s

    rep
    movw
    jmpi go, INITSEG
go: mov ax,cs
    mov ds,ax
```

从图 1-6 中我们已经了解到当时 CS 的值为 0x07C0，执行完这个跳转后，CS 值变为 0x9000（INITSEG），IP 的值为从 0x9000（INITSEG）到 go: mov ax, cs 这一行对应指令的偏移。换句话说，此时 CS:IP 指向 go: mov ax, cs 这一行，程序从这一行开始往下执行。图 1-7 形象地表示了跳转到 go: mov ax, cs 这一行执行时 CS 和 IP 的状态，如图右下方所示。

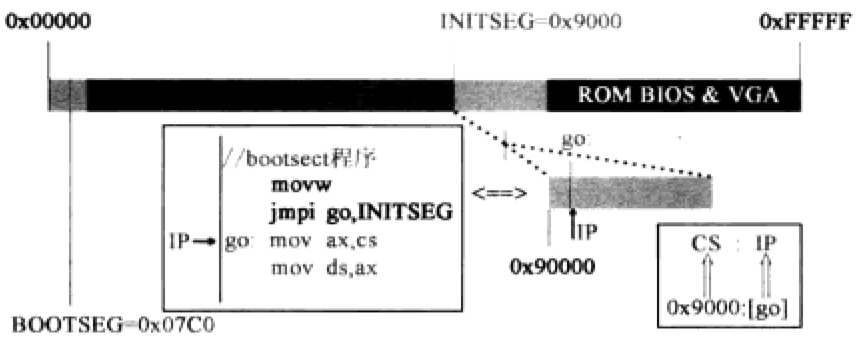


图 1-7　跳转到 go 处继续执行

此前的 0x07C00 这个位置是根据“两头约定”和“定位识别”而确定的。从现在起，操作系统已经不需要完全依赖 BIOS，可以按照自己的意志把自己的代码安排在内存中的某个位置了。

点评

```
jmpi go, INITSEG
go: mov ax, cs
```

这两行代码写得很巧。bootsect 复制完成后，在内存的 0x07C00 和 0x90000 位置处有两段完全相同的代码。请大家注意，复制代码这件事本身也是要靠指令执行的，执行指令的过程就是 CS 和 IP 不断变化的过程。执行到 jmpi go, INITSEG 这行之前，代码的作用就是复制代码自身；执行了 jmpi go, INITSEG 之后，程序就转到执行 0x90000 这边的代码了。Linus 的设计意图是跳转之后在新位置接着执行后面的 mov ax, cs，而不是死循环。jmpi go, INITSEG 与 go: mov ax, cs 配合，巧妙地实现了“到新位置后接着原来的执行序继续执行下去”的目的。

由于 bootscet 复制到了新的地方，并且要在新的地方继续执行。因为代码的整体位置发生了变化，所以代码中的各个段也会发生变化，前面已经改变了 CS，现在对 DS、ES、SS 和 SP 进行调整。我们看看下面的代码：

```
// 代码路径：boot/bootsect.s

go: mov ax, cs
    mov ds, ax
    mov es, ax
! put stack at 0x9ff00.
    mov ss, ax
    mov sp, #0xFF00       ! arbitrary value >>512
! load the setup-sectors directly after the bootblock.
! Note that 'es' is already set up.
```

上述代码的作用是通过 ax，用 CS 的值 0x9000 来把数据段寄存器（DS）、附加段寄存器（ES）、栈基址寄存器（SS）设置成与代码段寄存器 CS 相同的位置，并将栈顶指针 sp 指向偏移地址为 0xFF00 处。图 1-8 对此做了非常直观的描述。

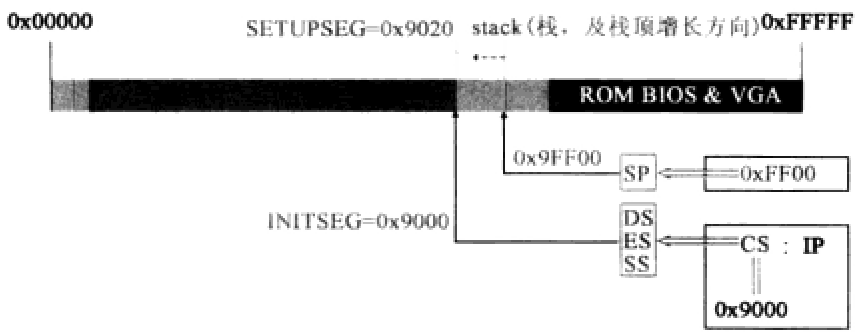


图 1-8　调整各个段寄存器的值

下面着重介绍一下与栈操作相关的寄存器的设置。SS 和 SP 联合使用构成了栈数据在内存中的位置值，对这两个寄存器的设置为后面程序的栈操作（如 push 和 pop 等）打下了基础。

现在可以观察一下 bootsect 中的程序，在执行设置 SS 和 SP 的代码之前，没有出现过栈操作指令，而在此之后就陆续使用了。这里对 SS 和 SP 进行的设置是分水岭。它标志着从现在开始，程序可以执行更为复杂一些的数据运算类指令了。

栈操作是有方向的，图 1-8 中标识了压栈的方向。注意，是由高地址到低地址的方向。

小贴士

DS/ES/FS/GS/SS：数据段寄存器，存在于 CPU 中，其中 SS（Stack Segment）指向栈段，此区域将按栈机制进行管理。

SP（Stack Pointer）：栈顶指针寄存器，指向栈段的当前栈顶。

注意，很多计算机书上使用“堆栈”这个词。本书用堆、栈表示两个概念。栈表示 stack，特指在 C 语言程序的运行时结构中，以“后进先出”机制运作的内存空间；堆表示 heap，特指用 C 语言库函数 malloc 创建、free 释放的动态内存空间。

至此，bootsect 的第一步操作：规划内存并把自身从 0x07C00 的位置复制到 0x90000 的位置的动作已经完成了。

3. 将 Setup 程序加载到内存中

下面，bootsect 程序就要执行它的第二步操作了：将 setup 程序加载到内存中。

加载 setup 这个程序，要借助 BIOS 提供的 int 0x13 中断向量所指向的中断服务程序（也就是磁盘服务程序）来完成，图 1-9 标注了 int 0x13 中断向量的位置以及这个中断向量所指向的磁盘服务程序的入口位置。

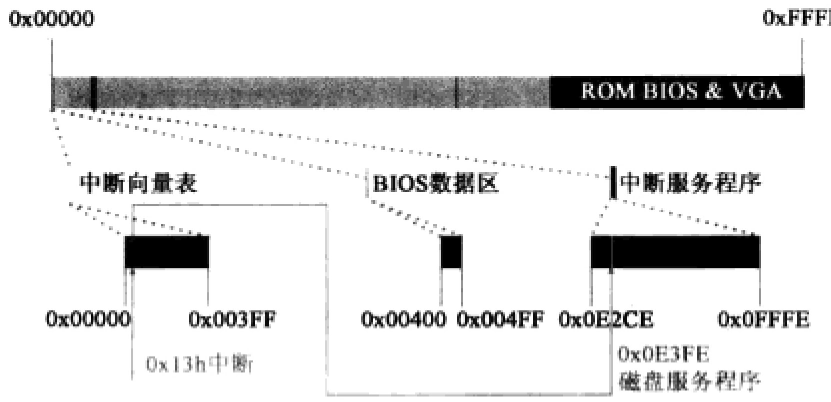


图 1-9　调用 int 0x13 中断

这个中断服务程序的执行过程与图 1-3 和图 1-4 中讲解过的 int 0x19 中断向量所指向的启动加载服务程序不同：

- int 0x19 中断向量所指向的启动加载服务程序是 BIOS 执行的，int 0x13 的中断服务程序是 Linux 操作系统自身的启动代码 bootsect 执行的。
- int 0x19 的中断服务程序只负责把软盘的第一扇区的代码加载到 0x07C00 位置，int 0x13 的中断服务程序则不然，它可以根据设计者的意图，把指定扇区的代码加载到内存的指定位置。

针对服务程序的这个特性，使用 int 0x13h 中断时，就要事先将指定的扇区和加载的内存位置等信息传递给服务程序，即传参。

执行代码如下：

```
// 代码路径：boot/bootsect.s

load_setup:
    mov dx, #0x0000             ! drive 0, head 0
    mov cx, #0x0002             ! sector 2, track 0
    mov bx, #0x0200             ! address = 512, in INITSEG
    mov ax, #0x0200+SETUPLEN    ! service 2, nr of sectors
    int 0x13                    ! read it
    jnc ok_load_setup           ! ok-continue
    mov dx, #0x0000
    mov ax, #0x0000             ! reset the diskette
    int 0x13
    j load_setup
ok_load_setup:
```

从代码开始处的 4 个 mov 指令可以看出，系统给 BIOS 中断服务程序传参是通过几个通用寄存器实现的，这是汇编程序的常用方法，与 C 语言的函数调用形式有很大不同。

参数传递完毕后，执行 int 0x13 指令，产生 0x13 中断，通过中断向量表找到这个中断服务程序，将软盘从第 2 扇区开始的 4 个扇区，即 setup.s 对应的程序加载至内存的 SETUPSEG（0x90200）处。根据图 1-5 的讲解，复制后的 bootsect 的起始位置是 0x90000，占用 512 字节的内存空间，不难看出 0x90200 紧挨着 bootsect 的尾端，所以 bootsect 和 setup 是连在一起的。图 1-10 表示了软盘中所要加载的扇区位置和扇区数，以及载入内存的目标位置和占用的空间。

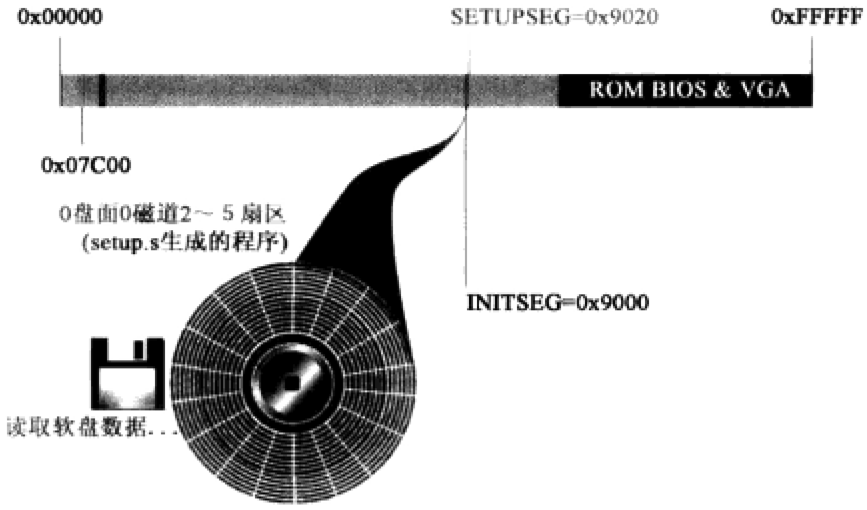


图 1-10　加载 setup 程序

现在，操作系统已经从软盘中加载了 5 个扇区的代码。等 bootsect 执行完毕后，setup 这个程序就要开始工作了。

注意，图 1-8 中 SS:SP 指向的位置为 0x9FF00，这与 setup 程序的起始位置 0x90200 还有很大的距离，即便 setup 加载进来后，系统仍然有足够的内存空间用来执行数据压栈操作。而且，在启动部分，要压栈的数据毕竟也是有限的，大家在后续的章节中会逐渐体会到，设计者在此是进行过精密测算的。

1.2.3　加载第三部分代码——system 模块

第二批代码已经载入内存，现在要加载第三批代码。仍然是使用 BIOS 提供的 int 0x13 中断，如图 1-11 所示，方法与图 1-9 所示的方法基本相同。

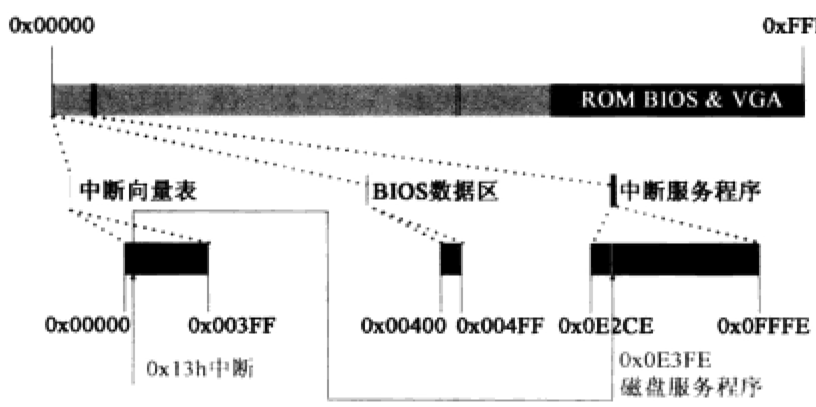


图 1-11　再次调用 int 0x13 中断

接下来 bootsect 程序就要执行第三批程序的载入工作，即将系统模块载入内存。

从底层技术上看，这次载入与前面的 setup 程序的载入没有本质的区别。比较突出的是这次加载的扇区数是 240，是之前 4 个扇区的 60 倍，且所需时间也是之前的几十倍。为了防止加载期间用户误认为是机器故障，而执行不适当的操作，Linus 在此设计了显示一行屏幕信息“Loading system ...”以提示用户计算机此时正在加载系统。值得注意的是，此时操作系统的 main 函数还没有开始执行，在屏幕上显示一行字符串远没有用 C 语言写一句 printf（“Loading system ...\n”）调用那么容易，所有工作都要靠一行一行的汇编代码来实现，从体系结构的角度看，显示器也是一个外设，所以还要用到其他的 BIOS 中断。这些代码比较多，对理解操作系统的启动原理没有特别直接的帮助，只要知道大意就可以了。我们真正需要掌握的是：bootsect 借着 BIOS 中断 int 0x13，将 240 个扇区的 system 模块加载进内存。加载工作主要是由 bootsect 调用 read_it 子程序完成的，这个子程序将软盘第 6 扇区开始的约 240 个扇区的 system 模块加载至内存的 SYSSEG(0x10000) 处往后的 120KB 空间中。图 1-12 中对 system 模块所占用的内存空间给出了形象的说明。

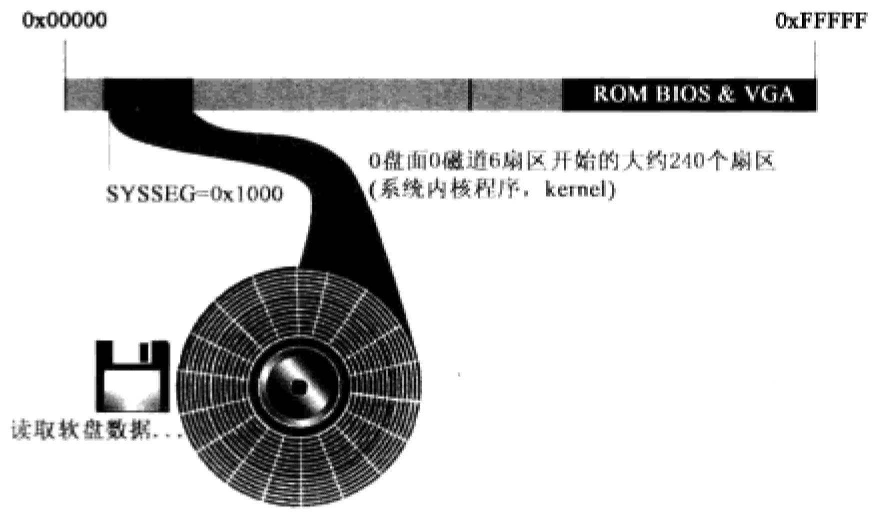


图 1-12　加载 system 模块

由于是长时间操作软盘，所以需要对软盘设备进行更多的监控，需要不断地对读盘结果进行检测，因此 read_it 后续的调用步骤比较多，但读盘工作最终是由 0x13 对应的中断服务程序完成的。

到此为止，第三批程序已经加载完毕，整个操作系统的代码已全部载入内存。bootsect 的主体工作已经做完了，还有一点小事，就是要再次确定一下根设备号，如图 1-13 所示。

经过一系列检测，得知软盘为根设备，所以就把根设备号保存在 root_dev 中，这个根设备号作为机器系统数据之一，将在第 2 章中讲到的“根文件系统加载”中发挥关键性的作用。

小贴士

根文件系统设备（Root Device）：Linux 0.11 使用 Minix 操作系统的文件系统管理方式，要求系统必须存在一个根文件系统，其他文件系统挂接其上，而不是同等地位。Linux 0.11 没有提供在设备上建立文件系统的工具，故必须在一个正在运行的系统上利用工具（类似 FDISK 和 Format）做出一个文件系统并加载至本机。故 Linux 0.11 的启动需要两部分数据，即系统内核镜

像和根文件系统。

注意，这里的文件系统指的不是操作系统内核中的文件系统代码，而是有配套的文件系统格式的设备，如一张格式化好的软盘。

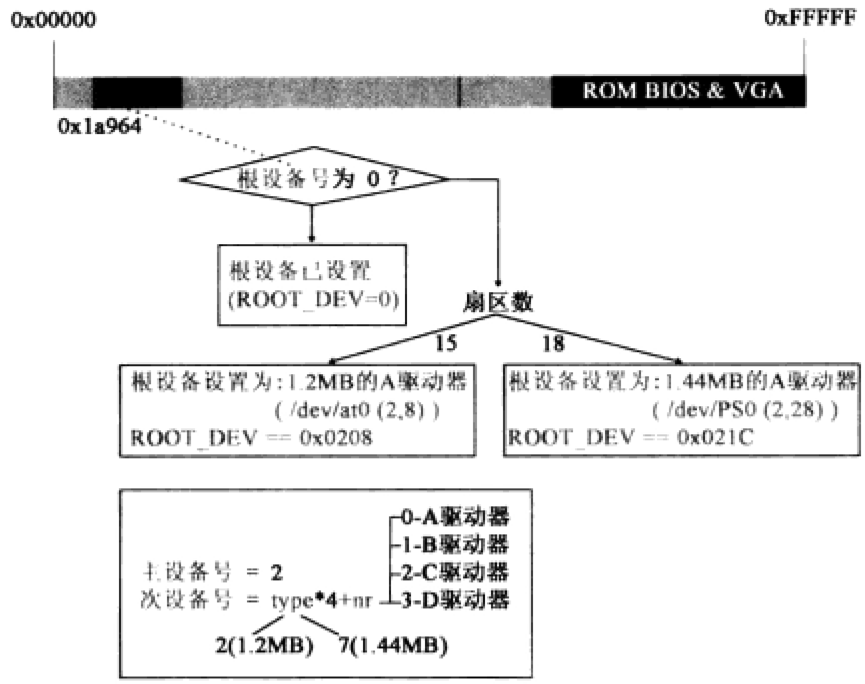


图 1-13　确认根设备号

现在，bootsect 程序的任务都已经完成！

下面要通过执行“jmpi 0, SETUPSEG”这行语句跳转至 0x90200 处，就是前面讲过的第二批程序——setup 程序加载的位置。CS:IP 指向 setup 程序的第一条指令，意味着由 setup 程序接着 bootsect 程序继续执行。图 1-14 形象地描述了跳转到 setup 程序后的起始状态，对应的代码如下：

```
// 代码路径：boot/bootsect.s

jmpi 0, SETUPSEG
```

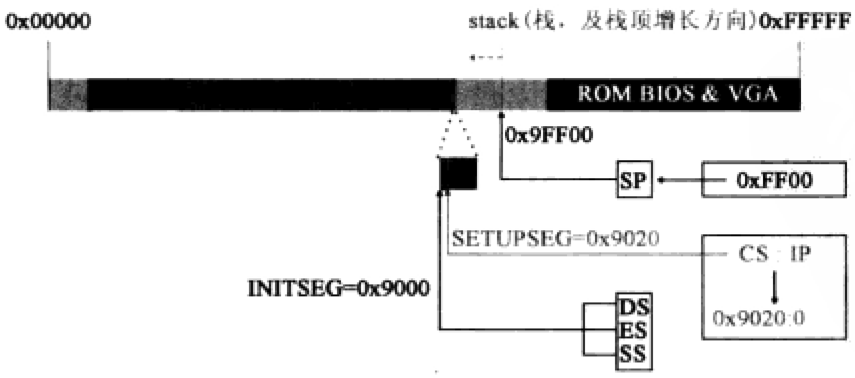


图 1-14　setup 开始执行

setup 程序现在开始执行。它做的第一件事请就是利用 BIOS 提供的中断服务程序从设备上提取内核运行所需的机器系统数据，其中包括光标位置和显示页面等数据，并分别从中断向量 0x41 和 0x46 向量值所指的内存地址处获取硬盘参数表 1 和硬盘参数表 2，把它们存放在 0x9000:0x0080 和 0x9000:0x0090 处。

这些机器系统数据被加载到内存的 0x90000~0x901FC 位置，图 1-15 标出了其内容及准确的位置。这些数据将在以后 main 函数执行时发挥重要作用。

提取机器系统数据的具体代码如下：

```
// 代码路径：boot/setup.s

    mov ax, #INITSEG              ! this is done in bootsect already, but...
    mov ds, ax
    mov ah, #0x03                 ! read cursor pos
    xor bh, bh
    int 0x10                      ! save it in known place, con_init fetches
    mov [0],dx                    ! it from 0x90000.

! Get memory size (extended mem, kB)
    mov ah, #0x88
    int 0x15
    mov [2], ax
…
…
mov cx, #0x10
mov ax, #0x00
rep
stosb
```

这段代码大约 70 行，由于篇幅限制，我们省略了大部分代码。

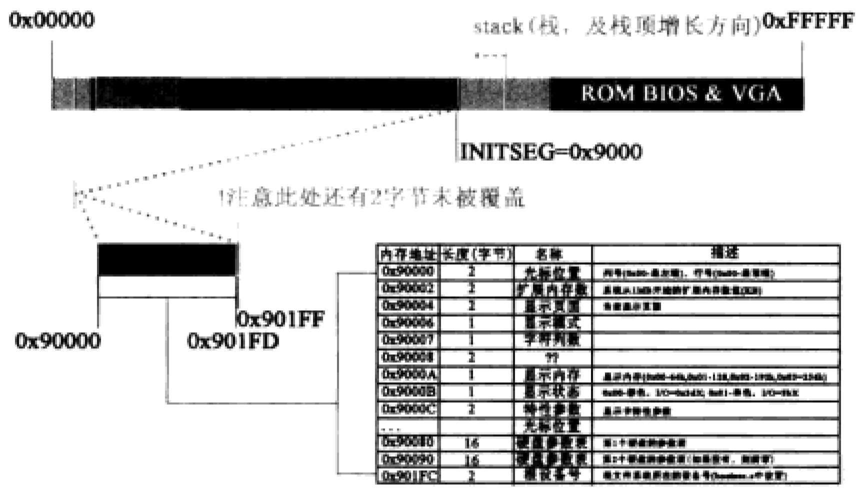


图 1-15　加载机器系统数据

注意，BIOS 提取的机器系统数据将覆盖 bootsect 程序所在的部分区域。由于这些数据是要留用的，因此在它们失去使用价值之前，一定不能被覆盖掉。

点评

机器系统数据所占的内存空间为 0x90000 ～ 0x901FD，共 510 个字节，即原来的 bootsect 只有 2 字节未被覆盖。可见，操作系统对内存的使用是非常严谨的。在空间上，操作系统对内存严格按需使用，要加载的数据刚好占用一个扇区的位置（只差 2 字节），而启动扇区 bootsect 又恰好是一个扇区，内存的使用规划像一个账本，前后对应；在时间上，使用完毕的空间立即挪作他用，启动扇区 bootsect 程序刚结束其使命，执行 setup 时立刻就将其用数据覆盖，内存的使用率

极高。虽然这与当时的硬件条件有限不无关系，但这种严谨的内存规划风格是很值得学习的。

到此为止，操作系统内核程序的加载工作已经完成。接下来的操作对 Linux 0.11 而言具有战略意义，系统通过已经加载到内存中的代码，将实现从实模式到保护模式的转变，使 Linux 0.11 真正成为“现代”操作系统。

1.3 开始向 32 位模式转变，为 main 函数的调用做准备

接下来操作系统要使计算机在 32 位保护模式下工作。这期间要做大量的重建工作，并且持续工作到操作系统的 main 函数的执行过程中。在本节中，操作系统执行的操作包括打开 32 位的寻址空间、打开保护模式、建立保护模式下的中断响应机制等与保护模式配套的相关工作、建立内存的分页机制，以及做好调用 main 函数的准备。

1.3.1 关中断并将 system 移动到内存地址起始位置 0x00000

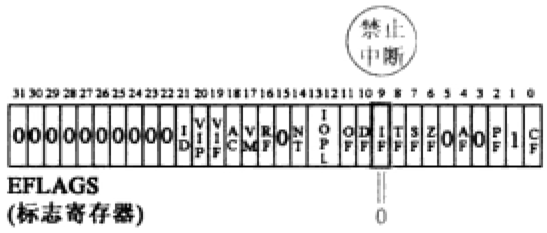


图 1-16 关中断

如图 1-16 所示，这个准备工作先要关闭中断，即将 CPU 的标志寄存器（EFLAGS）中的中断允许标志（IF）置 0。这意味着，在接下来的执行过程中，无论是否发生中断，系统都不再响应此中断，直到下一章要讲解的 main 函数中能够适应保护模式的中断服务体系被重建完毕才会打开中断，而那时候响应中断的服务程序将不再是 BIOS 提供的中断服务程序，取而代之的是由系统自身提供的中断服务程序。

代码为：

```
// 代码路径：boot/setup.s
cli
```

小贴士

EFLAGS：标志寄存器，存在于 CPU 中，32 位，包含一组状态标志、控制标志和系统标志。如第 0 位的 CF（Carry Flag）为 CPU 计算用到的进位标志，以及图中关中断操作涉及的第 9 位 IF（Interrupt Flag）中断允许标志。

点评

关中断（cli）和开中断（sti）操作将在操作系统代码中频繁出现，其意义深刻，用心良苦。慢慢地你会发现，cli 和 sti 总是在一个完整的操作过程的两头出现，目的是为了避免中断在此期

间介入。接下来的代码将为操作系统进入保护模式做准备，此处即将进行实模式下中断向量表和保护模式下中断描述符表（IDT）的交接工作。试想，如果没有 cli，又恰好发生了中断，如用户不小心碰了一下键盘，中断就要切进来，就不得不面对实模式的中断机制已经废除，但保护模式的中断机制尚未完成的尴尬局面，结果就是系统崩溃。cli 和 sti 保证了这个过程中中断描述符表能够完整创建，以避免不可预料的中断进入，从而造成中断描述符表创建不完整或新老中断机制混用。甚至可以理解为 cli 和 sti 是为了保护一个新的计算机生命的完整创建。

下面，setup 程序做了一个影响深远的动作：将位于 0x10000 的内核程序拷贝至内存地址起始位置 0x00000 处！代码如下：

```
// 代码路径：boot/setup.s
do_move:
    mov es,ax              ! destination segment
    add ax, #0x1000
    cmp ax, #0x9000
    jz end_move
    mov ds, ax             ! source segment
    sub di, di
    sub si, si
    mov cx, #0x8000
    rep
    movsw
    jmp do_move
```

图 1-17 准确标识了复制操作系统内核代码的源位置和目标位置及复制动作的方向。

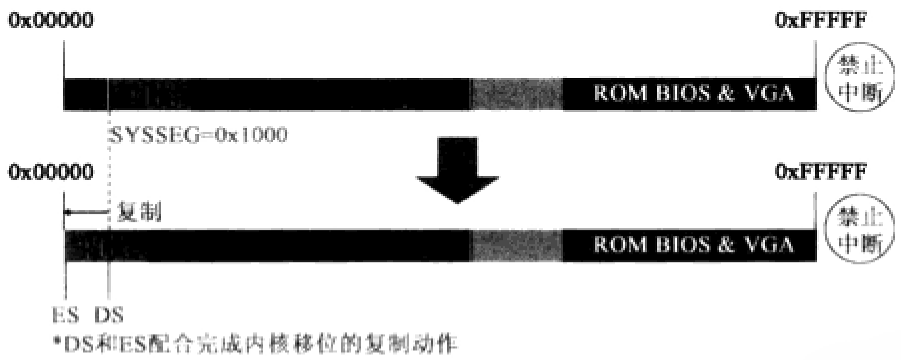


图 1-17　复制 system 模块至内存起始处

回顾一下图 1-2 的内容，0x00000 这个位置原来存放着由 BIOS 建立的中断向量表及 BIOS 数据区，这个复制动作将 BIOS 中断向量表和 BIOS 数据区完全覆盖，使它们不复存在。直到新的中断服务体系构建完毕之前，操作系统不再具备响应并处理中断的能力。现在，我们开始体会到图 1-16 中的关中断操作的意义了。

点评

这样做能取得一箭三雕的效果：

（1）废除 BIOS 的中断向量表，等价于废除了 BIOS 提供的实模式下的中断服务程序。

（2）收回使用寿命刚刚结束的程序所占的内存空间。

（3）让内核代码占据内存物理地址最开始的、最天然的、最有利的位置。

“破旧立新”这个成语用在这里特别贴切，system 模块复制到 0x00000 这个动作废除了 BIOS 的中断向量表，也就是废除了 16 位的中断机制。操作系统是不能没有中断的，对外设的使用、系统调用、进程调度都离不开中断。Linux 操作系统是 32 位的现代操作系统，16 位的中断机制对 32 位的操作系统而言显然是不合适的，这也是废除 16 位中断机制的根本原因。为了建立 32 位的操作系统，我们不但要“破旧”，还要“立新”——建立新的中断机制。

1.3.2 设置中断描述符表和全局描述符表

setup 程序继续为保护模式做准备，此时要通过 setup 程序自身提供的数据信息对中断描述符表寄存器 IDTR 和全局描述符表寄存器 GDTR 进行初始化设置。

小贴士

GDT（Global Descriptor Table，全局描述符表）：它是系统中唯一存放段寄存器内容（段描述符）的数组，配合程序进行保护模式下的段寻址。它在操作系统的进程切换中具有重要意义，可理解为所有进程的总目录表，其中存放着每一个任务（task）局部描述符表（LDT，Local Descriptor Table）地址和任务状态段（TSS，Task Structure Segment）地址，用于完成进程中各段的寻址、现场保护与现场恢复。

GDTR（Global Descriptor Table Register，GDT 基地址寄存器）：GDT 可以存放在内存的任何位置，当程序通过段寄存器引用一个段描述符时，需要取得 GDT 的入口，GDTR 所标识的即为此入口。在操作系统对 GDT 的初始化完成后，可以用 LGDT（Load GDT）指令将 GDT 基地址加载至 GDTR。

IDT（Interrupt Descriptor Table，中断描述符表）：保存保护模式下所有中断服务程序的入口地址，类似于实模式下的中断向量表。

IDTR（Interrupt Descriptor Table Register，IDT 基地址寄存器）：IDT 基地址寄存器，保存 IDT 的起始地址。

内核实现代码如下：

```
//代码路径：boot/setup.s

lidt        idt_48
lgdt        gdt_48

gdt:
    .word   0,0,0,0         ! dummy

    .word   0x07FF          ! 8Mb-limit=2047 (2048*4096=8Mb)
    .word   0x0000          ! base address=0
    .word   0x9A00          ! code read/exec
    .word   0x00C0          ! granularity=4096, 386

    .word   0x07FF          ! 8Mb-limit=2047 (2048*4096=8Mb)
```

```
    .word   0x0000          ! base address=0
    .word   0x9200          ! data read/write
    .word   0x00C0          ! granularity=4096, 386

idt_48:
    .word   0               ! idt limit=0
    .word   0,0             ! idt base=0L

gdt_48:
    .word   0x800           ! gdt limit=2048, 256 GDT entries
    .word   512+gdt,0x9     ! gdt base = 0X9xxxx
```

这些代码设置所需要的数据分别在 idt_48 和 gdt_48 所对应的标号处，它们与寄存器的对应方式如图 1-18 所示。

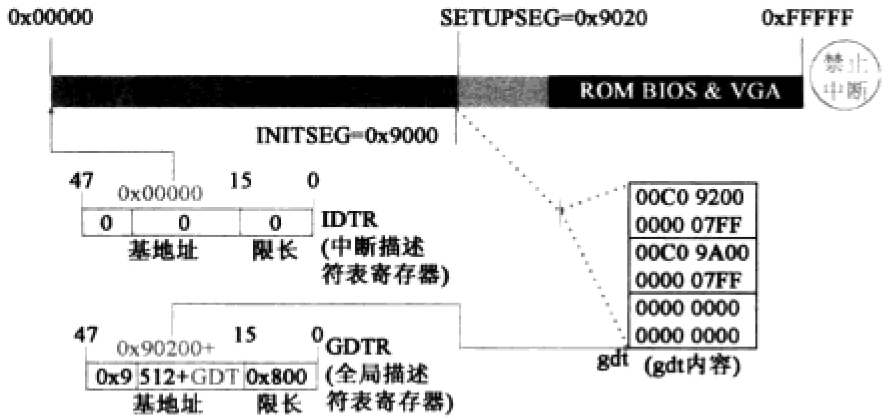


图 1-18　设置 GDTR 和 IDTR

点评

32 位的中断机制和 16 位的中断机制在原理上有比较大的差别，最明显的是 16 位的中断机制用的是中断向量表，中断向量表的起始位置在 0x00000 处，这个位置是固定的。32 位的中断机制用的是中断描述符表 IDT，位置是不固定的，可以由操作系统的设计者根据设计要求灵活安排，由 IDTR 寄存器来锁定其位置。

GDT 表是保护模式下管理段描述符的数据结构，对操作系统自身的运行和管理，以及进程调度有重大意义，后面的章节会有详细讲解。

因为，此时此刻内核尚未真正运行起来，还没有进程，所以现在创建的 GDT 表的第一项为空，第二项为内核代码段描述符，第三项为内核数据段描述符，其余项皆为空。

IDT 表虽然已经设置，实为一张空表，原因是目前已关中断，无须调用中断服务程序。此处反映的是数据“够用即得”的思想。

创建这两个表的过程可理解为是分两步进行的：

（1）在设计内核代码时，已经将两个表写好，并且把需要的数据也写好。

（2）将专用寄存器（IDTR 和 GDTR）指向表。

此处的数据区域是在内核源代码中设定和编译并直接加载至内存形成的一块数据区域。专用寄存器的指向由程序中的 lidt 指令和 lgdt 指令完成，具体操作见图 1-18。

值得一提的是，在内存中做出数据的方法有两种：

（1）划分一块内存区域并初始化数据，“看住”这块内存区域，使之能被找到；

（2）由代码做出数据，如用 push 代码压栈，“做出”数据，此处采用的是第一种方法。

1.3.3 打开 A20，实现 32 位寻址

下面的动作是标志性的——打开 A20！

打开 A20，意味着 CPU 可以进行 32 位寻址，最大寻址空间为 4GB，注意看图 1-19 中内存条范围的变化。从 5 个 F 扩展到 8 个 F，即 0xFFFFFFFF——4GB。

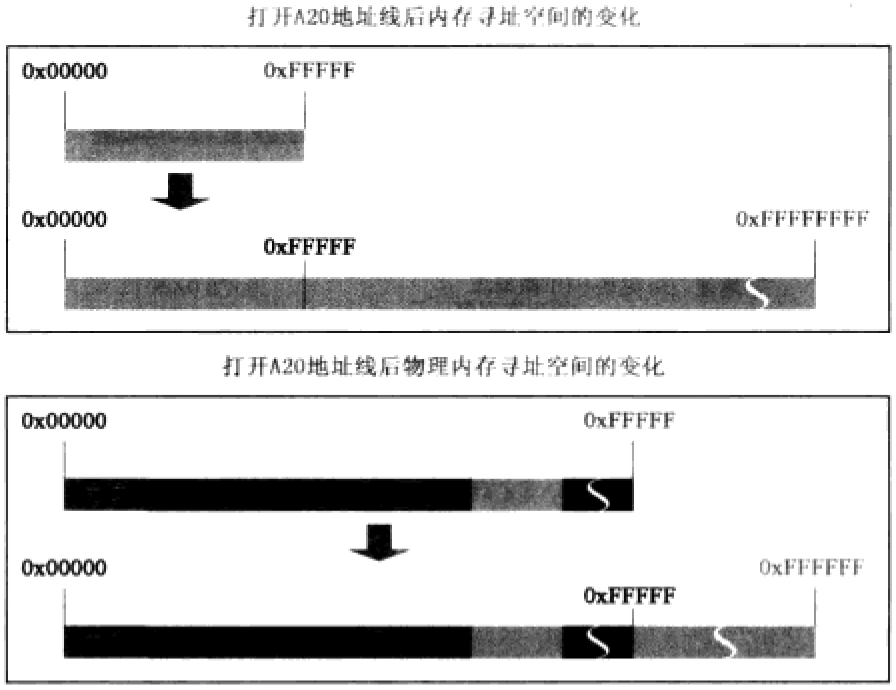


图 1-19　打开 A20

现在看来，Linux 0.11 还显得有些稚嫩，最大只能支持 16MB 的物理内存，但是其线性寻址空间已经是不折不扣的 4GB。

打开 A20 的代码如下：

```
// 代码路径：boot/setup.s

call        empty_8042
mov al,#0xD1                ! command write
out #0x64,al
call        empty_8042
mov al,#0xDF                ! A20 on
out #0x60,al
call        empty_8042
```

点评

实模式下 CPU 的寻址范围为 0 至 0xFFFFF，共 1MB 寻址空间，需要 0 ～ 19 号共 20 根地址线。进入保护模式后，将使用 32 位寻址模式，即采用 32 根地址线进行寻址，第 21 根至第 32 根地址线的选通，将意味着寻址模式的切换。

实模式下，当程序寻址超过 0xFFFFF 时，CPU 将“回滚”至内存地址起始处寻址（注意：在只有 20 位地址线的条件下，0xFFFFF + 1 = 0x00000，最高位溢出）。例如，系统的段寄存器（如 CS）的最大允许地址为 0xFFFF，指令指针（IP）的最大允许段内偏移也为 0xFFFF，两者确定的最大绝对地址为 0x10FFEF，这将意味着程序中可产生的实模式下的寻址范围比 1MB 多出将近 64KB（一些有特殊寻址要求的程序就利用了这个特点）。这样，此处对 A20 地址线的启用相当于关闭了 CPU 在实模式下寻址的“回滚”机制。在后续代码中也将看到利用此特点来验证 A20 地址线是否确实已经打开。

1.3.4 为在保护模式下执行 head.s 做准备

为了建立保护模式下的中断机制，setup 程序将对可编程中断控制器 8259A 进行重新编程。

小贴士

8259A 中断控制器：8259A 是专门为了对 8085A 和 8086/8088 进行中断控制而设计的芯片，它是可以用程序控制的中断控制器。单个的 8259A 能管理 8 级向量优先级中断。在不增加其他电路的情况下，最多可以级联成 64 级的向量优先级中断系统。

具体代码如下：

```
// 代码路径：boot/setup.s

    mov     al,#0x11              ! initialization sequence
    out     #0x20,al              ! send it to 8259A-1
    .word   0x00eb,0x00eb         ! jmp $+2, jmp $+2
    out     #0xA0,al              ! and to 8259A-2
    .word   0x00eb,0x00eb
    mov     al,#0x20              ! start of hardware int's (0x20)
    out     #0x21,al
    .word   0x00eb,0x00eb
    mov     al,#0x28              ! start of hardware int's 2 (0x28)
    out     #0xA1,al
    .word   0x00eb,0x00eb
    mov     al,#0x04              ! 8259-1 is master
    out     #0x21,al
    .word   0x00eb, 0x00eb
    mov al, #0x02                 ! 8259-2 is slave
    out     #0xA1,al
    .word   0x00eb,0x00eb
    mov     al,#0x01              ! 8086 mode for both
    out     #0x21,al
    .word   0x00eb,0x00eb
    out     #0xA1,al
    .word   0x00eb,0x00eb
    mov     al,#0xFF              ! mask off all interrupts for now
    out     #0x21,al
    .word   0x00eb,0x00eb
    out     #0xA1,al
```

重新编程的结果在图 1-20 中有直观的表述。

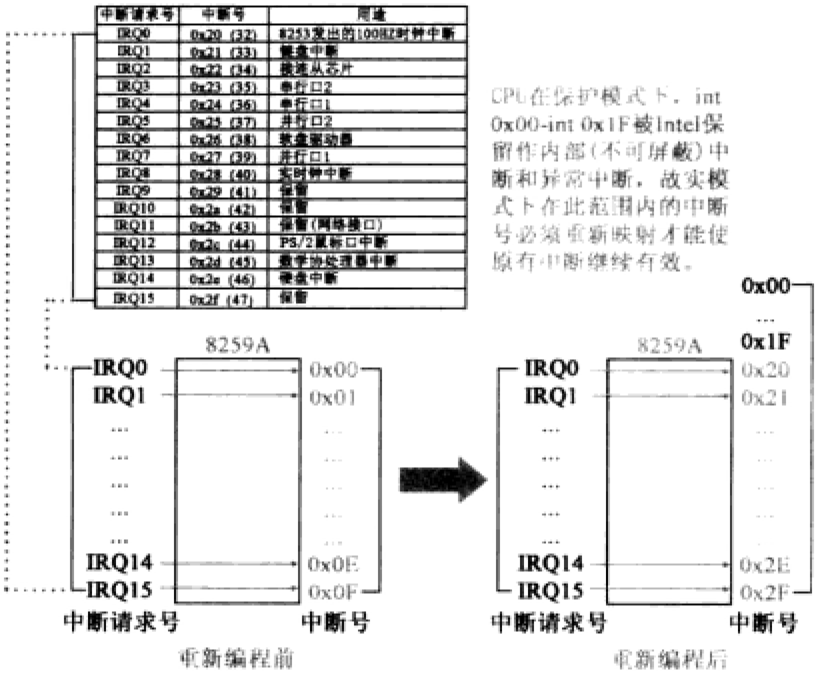


图 1-20　对可编程中断控制器重新编程

在保护模式下，int 0x00 ～ int 0x1F 被 Intel 保留作为内部（不可屏蔽）中断和异常中断。如果不对 8259A 进行重新编程，int 0x00 ～ int 0x1F 中断将被覆盖。例如，IRQ0（时钟中断）为 8 号（int 0x08）中断，但在保护模式下此中断号是 Intel 保留的“Double Fault（双重故障）”。因此，必须通过 8259A 编程将原来的 IRQ0x00 ～ IRQ0x0F 对应的中断号重新分布，即在保护模式下，IRQ0x00 ～ IRQ0x0F 的中断号是 int 0x20 ～ 0x2F。

setup 程序通过下面代码的前两行将 CPU 的工作方式设为保护模式。将 CR0 寄存器的第 0 位（PE）置 1，即设定处理器的工作方式为保护模式。

小贴士

CR0 寄存器：0 号 32 位控制寄存器，存放系统控制标志。第 0 位为 PE(Protected Mode Enable，保护模式使能）标志，置 1 时 CPU 工作在保护模式下，置 0 时为实模式。

```
// 代码路径：boot/setup.s

    mov     ax,#0x0001      ! protected mode (PE) bit
    lmsw    ax              ! This is it!
    jmpi    0,8             ! jmp offset 0 of segment 8 (cs)
```

图 1-21 对此做出了直观的标示。

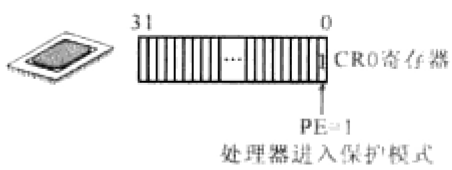


图 1-21　打开保护模式

CPU 工作方式转变为保护模式，一个重要的特征就是要根据 GDT 表来决定后续将执行哪里的程序。

注意看图 1-18 中对 GDT 表的设置，这些设置都是 setup 事先安排好了的默认设置。从 setup 程序跳转到 head 程序的方式，如图 1-22 所示。

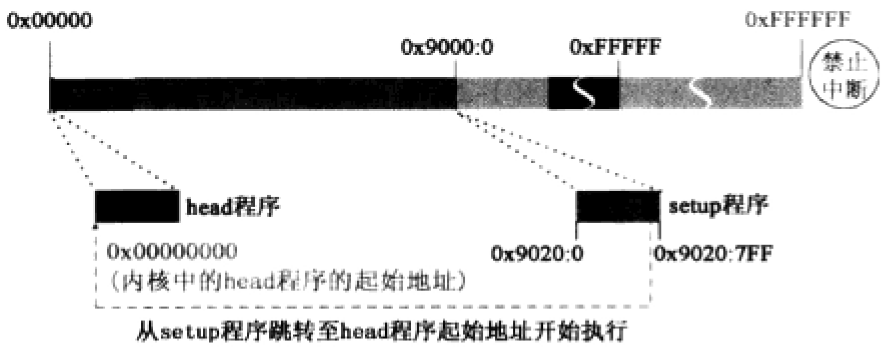


图 1-22 程序段间跳转

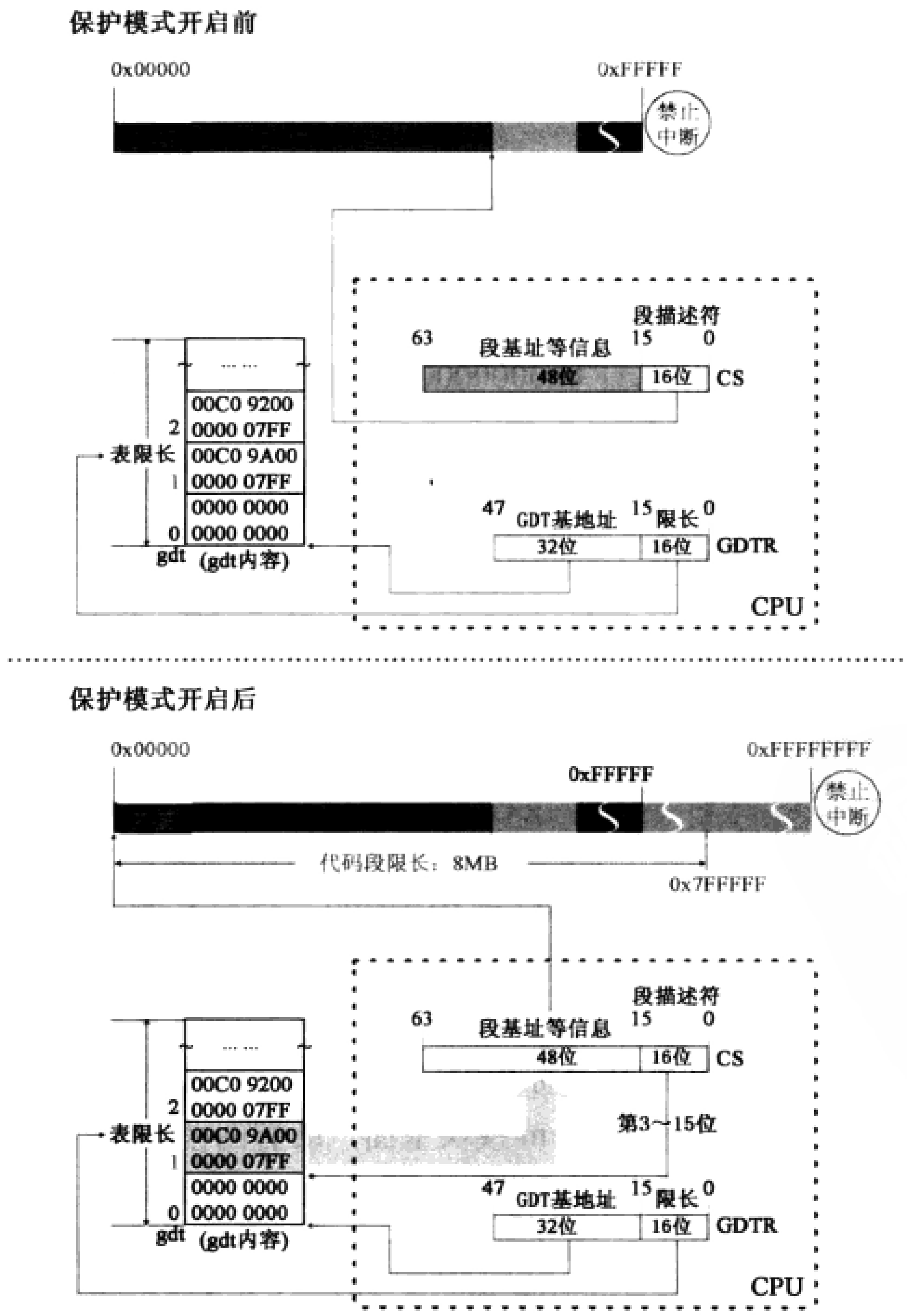


图 1-23 保护模式开启前后的指令寻址方式对比示意图

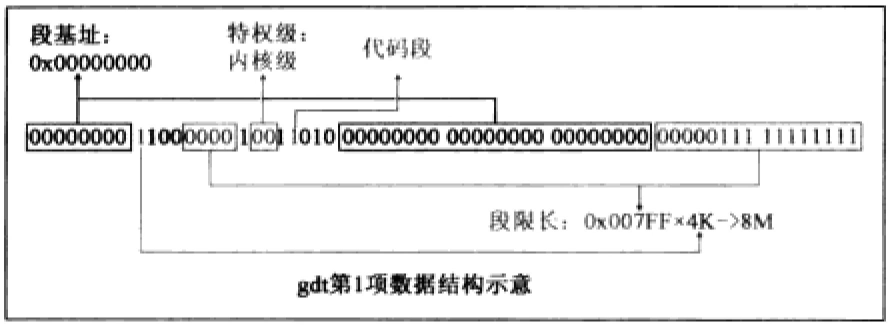


图 1-23 （续）

```
// 代码路径：boot/setup.s
jmpi        0, 8
```

这一句中的“0”是段内偏移，“8”是保护模式下的段选择符，用于选择描述符表和描述符表项以及所要求的特权级。这里的 8 的解读方式很有意思，如果把 8 当作十进制的 8 来看待，这行程序的意思就很难理解了。必须把 8 看成二进制的 1000，再把前后相关的代码联合起来当作一个整体看，并在头脑中形成类似图 1-23 所示的图，才能真正明白这行代码究竟在说什么。注意，这是一个以位为操作单位的数据使用方式，4 个 bit 的每一位都有明确的意义，这是底层源代码的一个特色。

这里 1000 的最后两位 00 表示内核特权级，与之相对的用户特权级是 11，第三位的 0 表示 GDT 表，如果是 1，则表示 LDT。1000 的 1 表示所选的表（在此就是 GDT 表）的 1 项（GDT 表项号排序为 0 项、1 项、2 项，也就是第 2 项）来确定代码段的段基址和段限长等信息，从图 1-22 中我们可以看到，代码是从段基址 0x00000000、偏移为 0 处开始执行的，也就是 head 程序的开始位置，这意味着将执行 head 程序。

到这里为止，setup 就执行完毕了，它为系统能够在保护模式下运行做了一系列的准备工作，但这些准备工作还不够，后续的准备工作将由 head 程序来完成。

1.3.5 head.s 开始执行

在讲解 head 程序之前，我们先介绍一下从 bootsect 到 main 函数执行的整体技术策略。

在执行 main 函数之前，先要执行三个由汇编代码生成的程序，即 bootsect、setup 和 head。之后，才执行由 main 函数开始的用 C 语言编写的操作系统内核程序。

前面我们讲过，第一步：加载 bootsect 到 0x07C00，然后复制到 0x90000；第二步：加载 setup 到 0x90200。值得注意的是，这两段程序是分别加载和分别执行的，head 程序与它们的加载方式有所不同。

大致的过程是，先将 head.s 汇编成目标代码，将用 C 语言编写的内核程序编译成目标代码，然后链接成 system 模块。也就是说，system 模块里面，既有内核程序，又有 head 程序，两者是紧挨着的。要点是，head 程序在前，内核程序在后，所以 head 程序名字叫“head”，head 程序在内存中占有 25KB+184B 的空间，请读者注意这个数字。前面讲解过，system 模块加载到内存后，setup 将 system 模块复制到 0x00000 位置，由于 head 程序在 system 的前面，所以实际上，

head 程序就在 0x00000 这个位置。head 程序和以 main 函数开始的内核程序在 system 模块中的布局示意图如图 1-24 所示。

head 程序除了做一些调用 main 的准备工作之外，还做了一件对内核程序在内存中的布局及内核程序的正常运行有重大意义的事，即用程序自身的代码在程序自身所在的内存空间创建了内核分页机制，即在 0x000000 的位置创建了页目录表、页表、缓冲区、GDT、IDT，并将 head 程序已经执行过的代码所占的内存空间覆盖，这意味着 head 程序自己将自己废弃，main 函数即将开始执行。

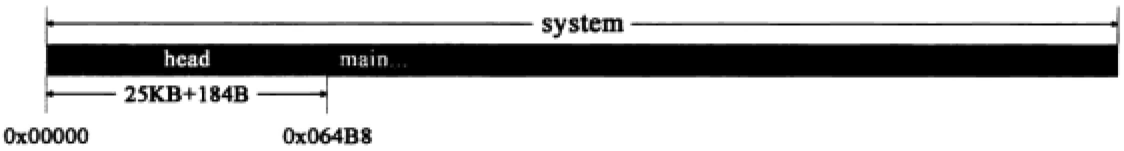


图 1-24　system 在内存中的分布示意图

以上就是 head 程序执行过程的整体策略，我们参照这个策略，看看 head 究竟是怎么执行的。

在讲解 head 程序执行之前，我们先来关注一个标号：_pg_dir。

如下面的代码所示：

```
// 代码路径：boot/head.s

_pg_dir:
startup_32:
movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
```

标号 _pg_dir 用于标识内核分页机制完成后的内核起始位置，也就是物理内存的起始位置 0x000000。head 程序马上就要在此处建立页目录表，为分页机制做准备。这一点非常重要，是内核能够掌控用户进程的基础之一，后续章节将逐步讲解。图 1-25 中描述了页目录表在内存中所占的位置。

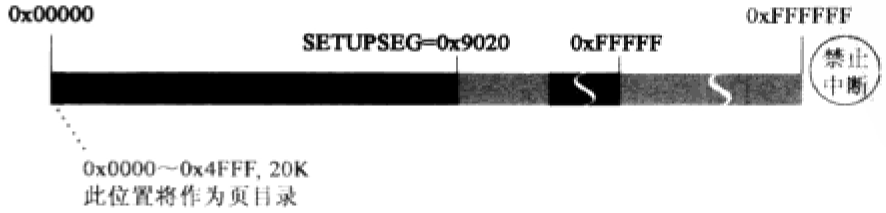


图 1-25　建立内核分页机制

现在 head 程序正式开始执行，一切都是为适应保护模式做准备。在图 1-25 中，其本质就是让 CS 的用法从实模式转变到保护模式。在实模式下时，CS 本身就是代码段基址。在保护模式下时，CS 本身并不是代码段基址，而是代码段选择符。通过图 1-25 的分析得知，jmpi 0, 8 这句

代码使 CS 和 GDT 表的第 2 项关联，并且使代码段的基址指向了 0x000000。

从现在开始，要将 DS、ES、FS 和 GS 等其他寄存器从实模式转变到保护模式。执行代码如下：

```
// 代码路径：boot/head.s

startup_32:
movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    mov %ax,%fs
    mov %ax,%gs
```

执行完毕后，DS、ES、FS 和 GS 中的值都变为 0x10。与前面提到的 jmpi 0，8 中 8 的分析方法相同，0x10 也应看成二进制的 00010000，最后三位与前面讲解的一样，其中最后两位的 00 表示内核特权级、第 3 位的 0 表示选择 GDT 表，第 4、5 两位的 10 是 GDT 表的 2 项，也就是第 3 项。也就是说，4 个寄存器用的是同一个全局描述符，它们的段基址、段限长和特权级都是相同的，特别要注意的是，影响段限长的关键字段的值是 0x7ff，段限长就是 8MB。

图 1-26 的左下部给出了详细示意。

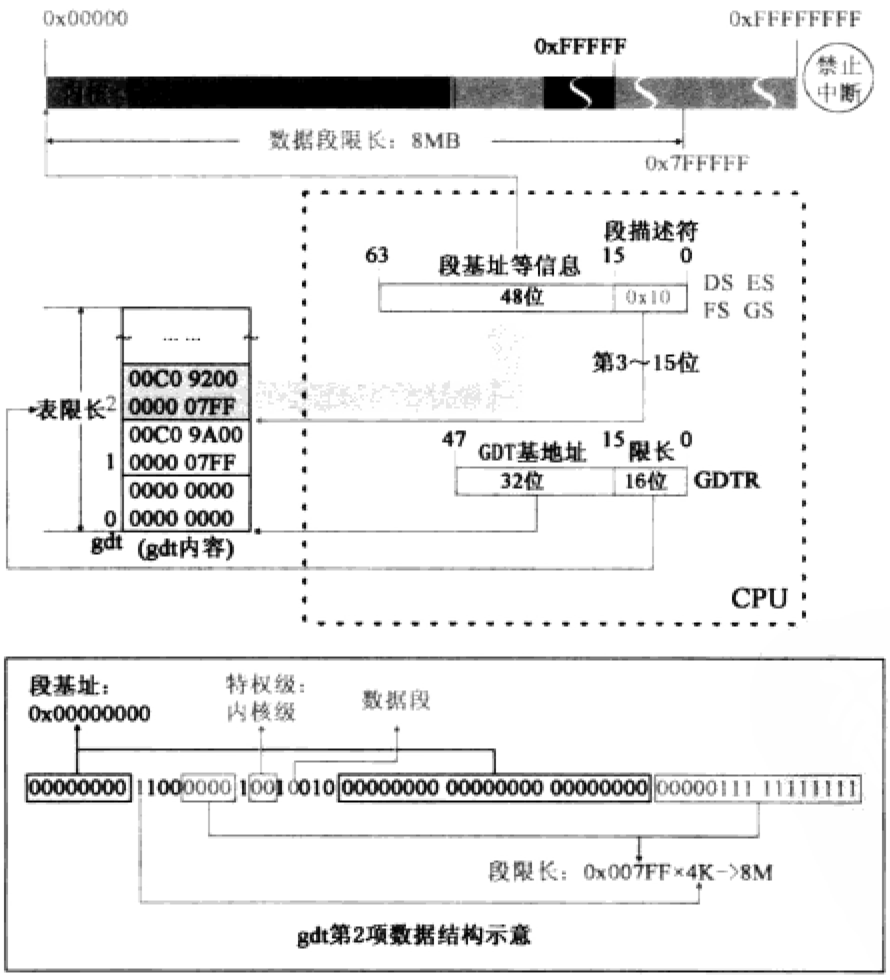


图 1-26 设置 DS、ES、FS、GS

具体的设置方式与图 1-23 类似，即都要参考 GDT 表中的内容，movl $0x10,%eax 中的 0x10 是 GDT 表中的偏移值（用二进制表示就是 10000），即要参考 GDT 表中第 2 项的信息（GDT 表项号排序为第 0 项、第 1 项、第 2 项）来设置这些段寄存器，这一项就是内核数据段描述符。

点评

各段重叠，这样的编码操作方式，需要头脑非常清楚！

SS 现在也要转变为栈段选择符，栈顶指针也成为 32 位的 esp，如下所示：

```
lss _stack_start,%esp
```

在 kernel/sched.c 中，stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 } 这行代码将栈顶指针指向 user_stack 数据结构的最末位置，这个数据结构是在 kernel/sched.c 中定义的，如下所示：

```
long user_stack [ PAGE_SIZE>>2 ]
```

我们测算出其起始位置为 0x1E25C。

小贴士

取段寄存器指令（Load Segment Instruction）：该组指令的功能是把内存单元的一个“低字”传送给指令中指定的 16 位寄存器，把随后的一个“高字”传给相应的段寄存器（DS、ES、FS、GS 和 SS）。其指令格式如下：

```
LDS/LES/LFS/LGS/LSS  Reg, Mem
```

指令 LDS（Load Data Segment Register）和 LES（Load Extra Segment Register）在 8086CPU 中就存在，而 LFS 和 LGS、LSS（Load Stack Segment Register）是 80386 及其以后 CPU 中才有的指令。如果 Reg 是 16 位寄存器，那么 Mem 必须是 32 位指针；如果 Reg 是 32 位寄存器，那么 Mem 必须是 48 位指针，其低 32 位给指令中指定的寄存器，高 16 位给指令中的段寄存器。

0x10 将 SS 的值设置为与前面 4 个段选择符的值相同。这样，SS 与前面讲解过的 4 个段选择符相同，段基址都指向 0x000000，段限长都是 8MB，特权级都是内核特权级，后面的压栈动作就要在这里进行。

特别值得一提的是，现在刚刚从实模式转变到保护模式，段基址的使用方法和实模式差别非常大，要使用 GDT 产生段基址，前面讲到的那几行设置段选择符的指令本身都是要用 GDT 寻址的。现在就能清楚地看出，如果没有 setup 程序在 16 位实模式下模拟 32 位保护模式而创建的 GDT，恐怕前面这几行指令都无法执行。

注意，栈顶的增长方向是从高地址向低地址的。参见图 1-27 的下部，注意栈段基址和 ESP 在图中的位置。

我们现在回忆一下图 1-8 中对栈指针寄存器的设置，那时是设置 sp，而这时是设置 esp，多了一个字母 e，这是为适应保护模式而做的调整。这段内容对应的代码如下：

```
// 代码路径：boot/head.s
lss _stack_start, %esp
```

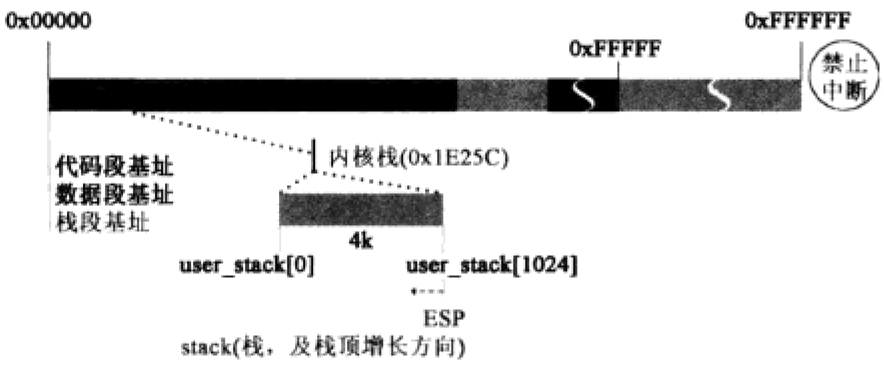


图 1-27　设置栈

head 程序接下来对中断描述符表进行设置，代码如下所示：

```
// 代码路径：boot/head.s
call setup_idt
……
setup_idt:
    lea ignore_int,%edx
    movl $0x00080000,%eax
    movw %dx,%ax            /* selector = 0x0008 = cs */
    movw $0x8E00,%dx        /* interrupt gate-dpl=0, present */
    lea _idt,%edi
    mov $256,%ecx
rp_sidt:
    movl %eax,(%edi)
    movl %edx,4(%edi)
    addl $8,%edi
    dec %ecx
    jne rp_sidt
    lidt idt_descr
    ret
```

小贴士

一个中断描述符的结构如下：

31　　　　23	15				7	0	
中断服务程序偏移地址（OFFSET）位31..16	P	DPL	0	1 1 1 0	0 0 0	未使用	4
段选择符（SELECTOR）	中断服务程序偏移地址（OFFSET）位15..0						0

中断描述符为 64 位，包含了其对应中断服务程序的段内偏移地址（OFFSET）、所在段选择符（SELECTOR）、段特权级（DPL）、段存在标志（P）、段描述符类型（TYPE）等信息，供 CPU 在程序中需要进行中断服务时找到相应的中断服务程序。其中，第 0 ～ 15 位和第 48 ～ 63 位组合成 32 位的中断服务程序的段内偏移地址；第 16 ～ 31 位为段选择符（SELECTOR），定位中断服务程序所在段；第 47 位为段存在标志（P），用于标识此段是否存在于内存中，为虚拟存储提供支持；第 45 ～ 46 位为特权级标志（DPL），特权级范围从 0 ～ 3；第 40 ～ 43 位为段描述符类型标志（TPYE），中断描述符对应的类型标志为 1110（0xE），即将此段描述符标记为“386 中断门”。

这是重建保护模式下中断服务体系的开始，程序先让所有的中断描述符默认指向 ignore_int 这个位置（将来 main 函数里面还要让中断描述符对应具体的中断服务程序），之后还要对中断描述符表寄存器的值进行设置。图 1-28 显示了具体的操作状态。

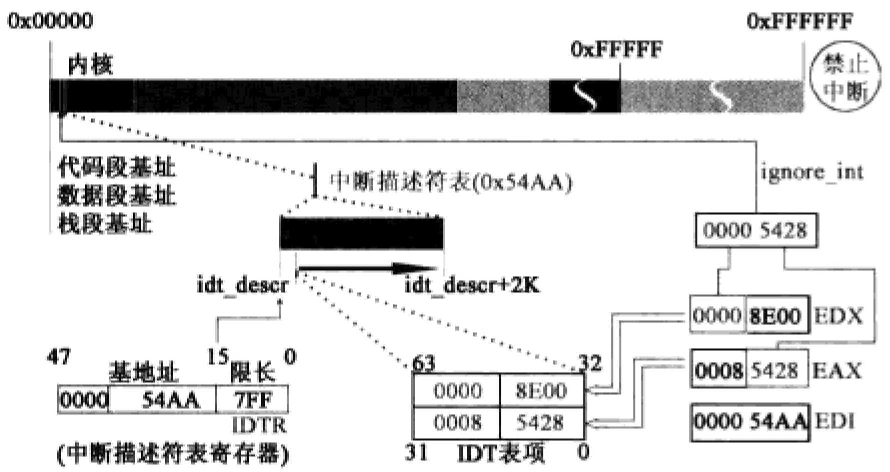


图 1-28　设置中断描述符表

点评

构造中断描述符表，并使所有中断服务程序指向同一段只显示一行提示信息就返回的服务程序，先使中断机制的整体架构搭建起来（实际的中断服务程序挂接则在 main 函数中完成）。从编程技术上讲，这是一个占位的操作方式，也防止了“野指针”。

现在，head 程序要废除已有的 GDT，并在内核中的新位置重新创建全局描述符表，如图 1-29 所示。其中第二项和第三项分别为内核代码段描述符和内核数据段描述符，其段限长均被设置为 16MB，并设置全局描述符表寄存器的值。

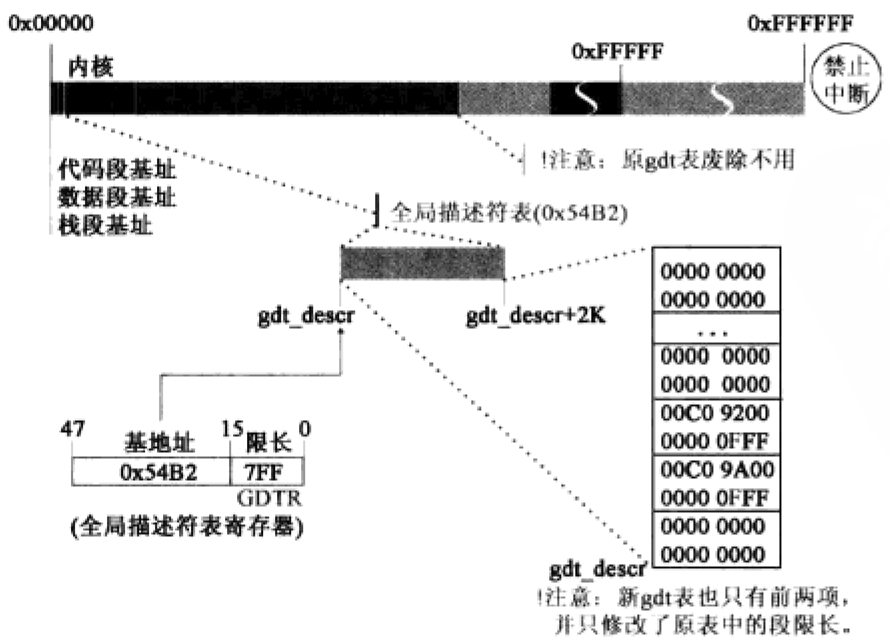


图 1-29　重新创建 GDT

代码如下：

```
// 代码路径：boot/head.s
```

```
setup_gdt
...........
setup_gdt:
lgdt gdt_descr
ret

gdt:       .quad 0x0000000000000000       /* NULL descriptor */
    .quad 0x00c09a0000000fff              /* 16Mb */
    .quad 0x00c0920000000fff              /* 16Mb */
    .quad 0x0000000000000000              /* TEMPORARY-don't use */
    .fill 252,8,0                         /* space for LDT's and TSS's etc */
```

点评

为什么要废除原来的 GDT 而重新设置一套 GDT 呢？

原来 GDT 所在的位置是设计代码时在 setup.s 里面设置的，将来这个 setup 模块所在的内存位置会在设计缓冲区时被覆盖。如果不改变位置，GDT 的内容将来肯定会被缓冲区覆盖掉，从而影响系统的运行。这样一来，将来整个内存中唯一安全的地方就是现在 head.s 所在的位置了。

那么有没有可能在执行 setup 程序时直接把 GDT 的内容拷贝到 head.s 所在的位置呢？肯定不能，如果先复制 GDT 的内容，后移动 system 模块，它就会被后者覆盖掉；如果先移动 system 模块，后复制 GDT 的内容，它又会把 head.s 对应的程序覆盖掉，而这时 head.s 还没有执行呢。所以，无论如何，都要重新建立 GDT。

全局描述符表 GDT 的位置和内容发生了变化，特别要注意最后的三位是 fff，说明段限长不是原来的 8MB，而是现在的 16MB。如果后面的代码第一次使用这几个段选择符就是访问 8MB 以后的地址空间，将会产生段限长超限报警，为了防止这类可能发生的情况，这里再次对一些段选择符进行重新设置，包括 DS、ES、FS、GS 和 SS，方法与图 1-26 类似，主要是段限长增加了一倍，变为了 16MB。上述过程如图 1-30 所示。

调整 DS 和 ES 等寄存器的对应代码如下：

```
//代码路径：boot/head.s
    movl $0x10,%eax                # reload all the segment registers
    mov %ax,%ds                    # after changing gdt. CS was already
    mov %ax,%es                    # reloaded in 'setup_gdt'
    mov %ax,%fs
    mov %ax,%gs
```

通过测试我们发现，这是一种舍近求远的方法，其实只要在 setup 中构建第一个 GDT 表时把控制段限长的 7ff 直接设置为 fff 就可以一步到位了，不需要在这里重新设置段选择符。

现在 user_stack 数据结构的起始位置就是内核栈的栈底，栈顶指针 esp 指向 user_stack 数据结构的外边缘，也就是内核栈的栈顶。这样，当后面的程序需要压栈时，就可以最大限度地使用栈空间。栈顶的增长方向是从高地址向低地址的，如图 1-31 所示。设置 esp 的代码如下：

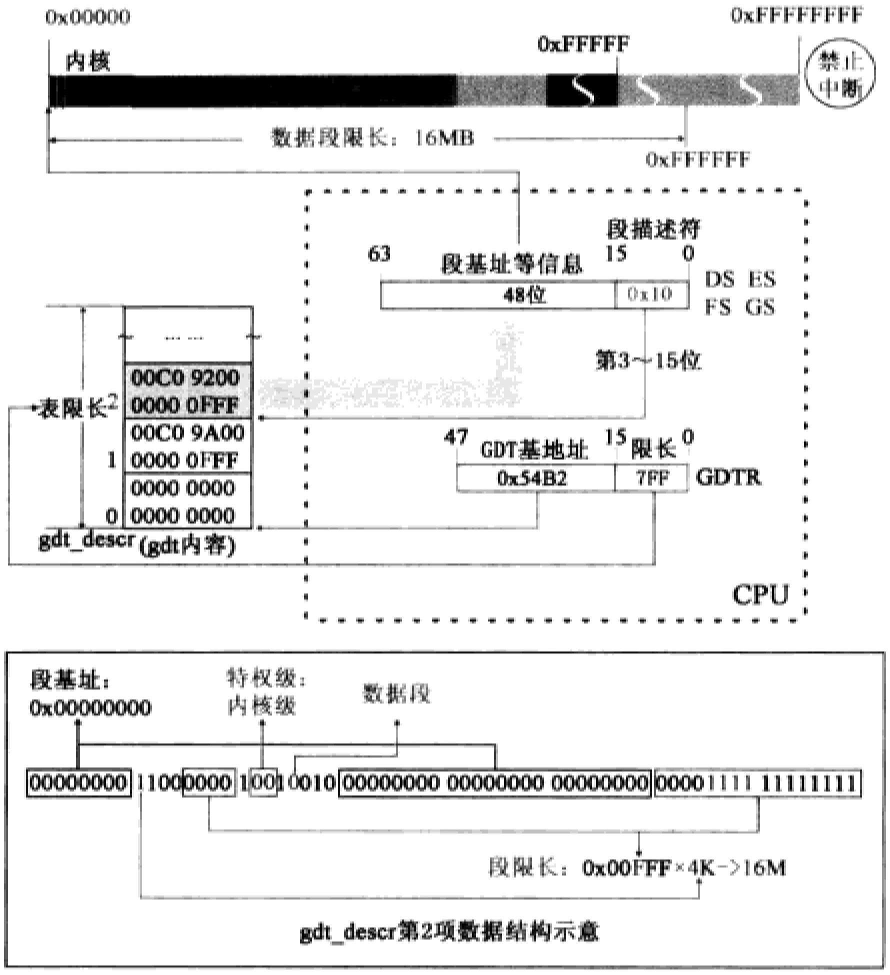


图 1-30　再一次调整 DS、ES、FS、GS

```
// 代码路径：boot/head.s
lss _stack_start,%esp
```

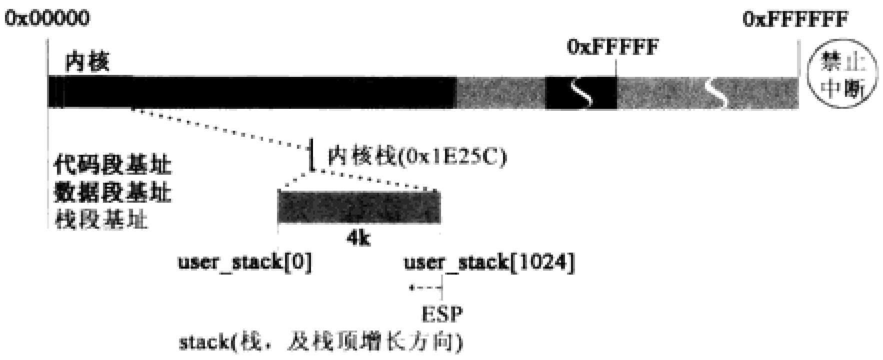


图 1-31　设置内核栈

因为 A20 地址线是否打开是保护模式与实模式的根本区别之一，所以，现在要检验 A20 地址线是否确实打开了。图 1-32 在左下部给出了直观的标示。

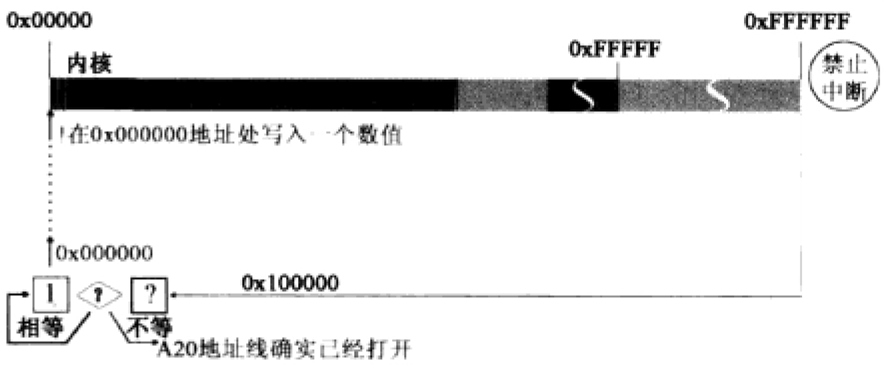


图 1-32　检验 A20 是否打开

检验 A20 是否打开的代码如下：

```
// 代码路径：boot/head.s
xorl %eax, %eax
1:  incl %eax
movl %eax, 0x000000
cmpl %eax, 0x100000
je 1b
```

点评

A20 如果没有打开，则计算机处于实模式下，超过 0xFFFFF 寻址必然“回滚”。一个特例是 0x100000 就会回滚到 0x000000，也就是说，地址 0x100000 处存储的值必然和地址 0x000000 处存储的值完全相同（参见图 1-30 的描述）。通过在内存 0x000000 位置写入一个数据，然后比较此处和 1MB（即 0x100000，注意，已超过实模式寻址范围）处数据是否一致，就可以检验 A20 地址线是否已打开。

确定 A20 地址线已经打开之后，head 程序如果检测到数学协处理器存在，则将其设置为保护模式工作状态，图 1-33 给出了示意。

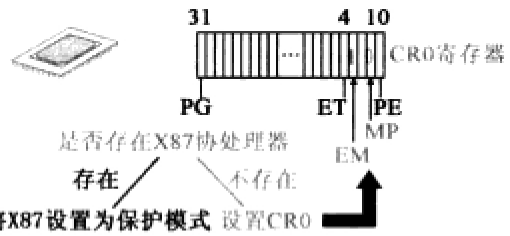


图 1-33　检测数学协处理器

小贴士

x87 协处理器：为了弥补 x86 系列在进行浮点运算时的不足，Intel 于 1980 年推出了 x87 系列数学协处理器，那时还是一个外置的、可选的芯片（笔者当时的 80386 计算机上就没有安装 80387 协处理器）。1989 年，Intel 发布了 486 处理器，自此之后，CPU 一般都内置了协处理器。这样，对于 486 以前的计算机而言，操作系统检验 x87 协处理器是否存在就非常必要了。

检测数学协处理器对应的代码如下：

```
// 代码路径：boot/head.s

movl %cr0,%eax
......
call check_x87

check_x87:
......
ret
```

head 程序将为调用 main 函数做最后的准备，这是 head 程序执行的最后阶段，也是 main 函

数执行前的最后阶段。

执行代码如下：

```
// 代码路径：boot/head.s
jmp after_page_tables

after_page_tables:
pushl $0
pushl  $0
pushl  $0
```

Linus 在代码中的原注释是"These are the parameters to main :-)"，但实际上我们并没有看到，也没有测试到 main 函数使用这三个参数，具体如图 1-34 下方的标示。

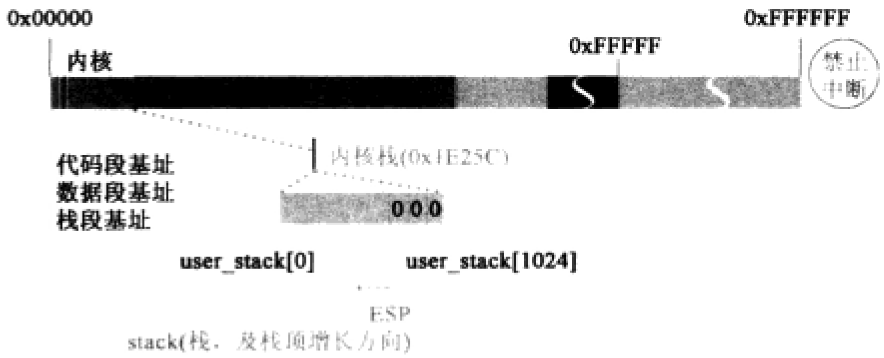


图 1-34　将 envp、argv、argc 压栈

head 程序将 L6 标号和 main 函数入口地址压栈，栈顶为 main 函数地址，目的是使 head 程序执行完后通过 ret 指令就可以直接执行 main 函数，具体请见图 1-35 下方的标示。

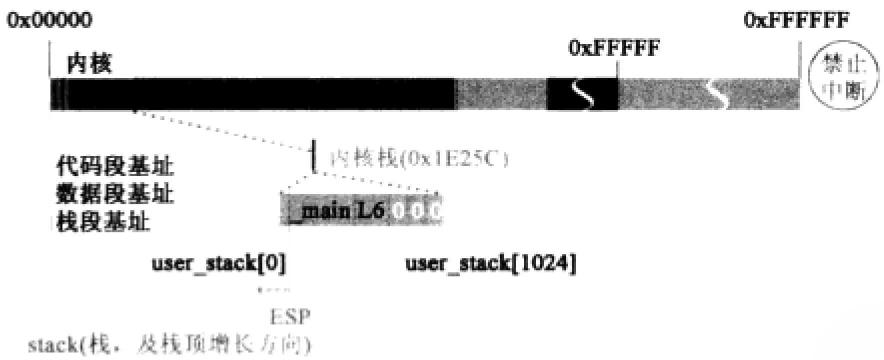


图 1-35　将 main 函数入口地址和 L6 标号压栈

如果 main 函数退出，就会返回到这里的标号 L6 处继续执行下去，并产生死循环。

执行代码如下：

```
// 代码路径：boot/head.s
pushl $L6
pushl $_main
```

这些压栈动作完成后，head 程序将跳转至 setup_paging: 去执行，开始创建分页机制。

首先会将页目录表和 4 个页表放在物理内存的起始位置。从内存起始位置开始的 5 页空间内容全部清零（每页 4KB），为初始化页目录和页表做准备。注意，这个动作起到了用一个页目录表和 4 个页表覆盖 head 程序自身所占内存空间的作用。图 1-36 下方给出了直观的示意图。

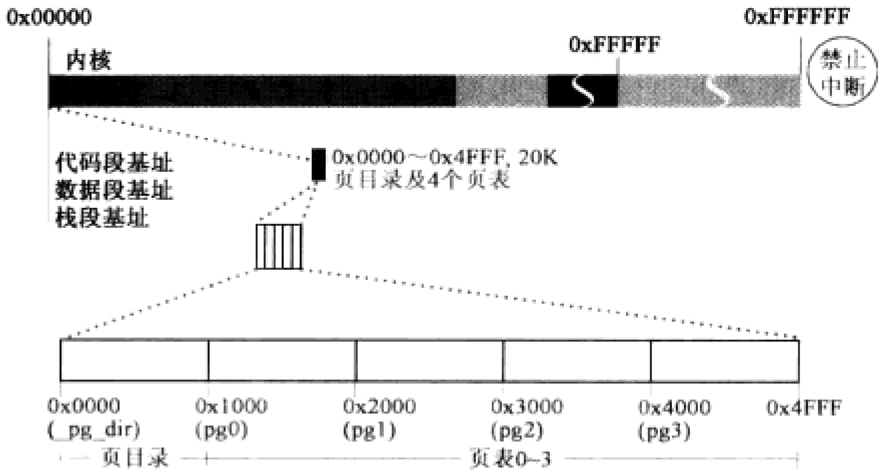


图 1-36　将页目录表和页表放在内存起始位置

上述动作的执行代码如下：

```
// 代码路径：boot/head.s
jmp setup_paging

setup_paging:
movl $1024*5,%ecx
xorl %eax,%eax
xorl %edi,%edi
cld;rep;stosl
```

点评

将页目录表和 4 个页表放在物理内存的起始位置，这个动作的意义重大，是操作系统能够掌控全局、掌控进程在内存中安全运行的基石之一，后续章节会逐步论述。

head 程序将页目录表和 4 个页表所占物理内存空间清 0 后，设置页目录表的前 4 项，使之分别指向 4 个页表，如图 1-37 左下方所示。

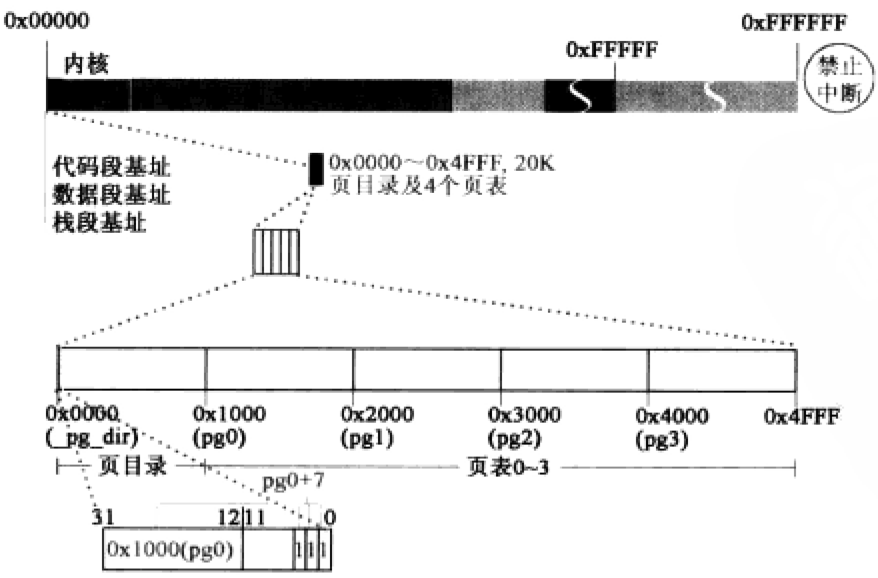


图 1-37　使页目录表的前 4 项指向 4 个页表

上述动作的代码如下：

```
// 代码路径：boot/head.s
movl $pg0+7,_pg_dir              /* set present bit/user r/w */
```

```
movl $pg1+7,_pg_dir+4            /*  --------- " " --------- */
movl $pg2+7,_pg_dir+8            /*  --------- " " --------- */
movl $pg3+7,_pg_dir+12           /*  --------- " " --------- */
movl $pg3+4092,%edi
movl $0xfff007,%eax              /*  16Mb-4096 + 7 (r/w user,p) */
```

head 程序设置完页目录表后，Linux 0.11 在保护模式下支持的最大寻址地址为 0xFFFFFF（16MB），此处将第 4 张页表（由 pg3 指向的位置）的最后一个页表项（pg3+4902 指向的位置）指向寻址范围的最后一个页面，即 0xFFF000 开始的 4KB 字节大小的内存空间。具体请看图 1-38 右下方的标示。

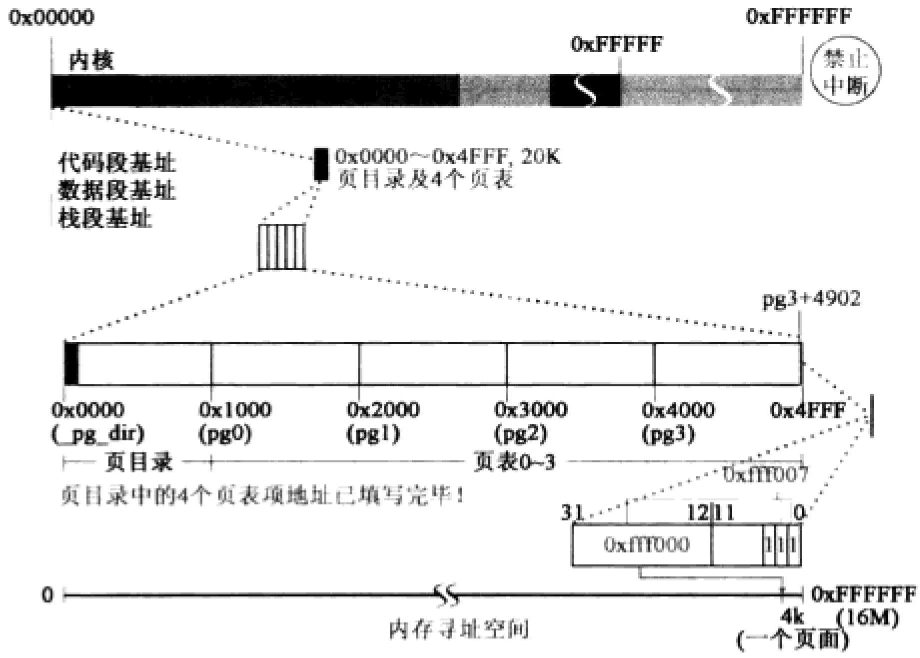


图 1-38　页目录表设置完成后的状态

然后开始从高地址向低地址方向填写全部的 4 个页表，依次指向内存从高地址向低地址方向的各个页面，图 1-38 是首次设置页表。

继续设置页表。将第 4 张页表（由 pg3 指向的位置）的倒数第二个页表项（pg3-4+4902 指向的位置）指向倒数第二个页面，即 0xFFF000-0x1000（0x1000 即 4k，一个页面的大小）开始的 4k 字节内存空间。请读者认真对比图 1-39 和图 1-38，图中有多处位置发生了变化。

最终，从高地址向低地址方向完成全部 4 个页表的填写，页表中的每一个页表项分别指向内存从高地址向低地址方向的各个页面，如图 1-40 下方所示。

这 4 个页表都是内核专属的页表，将来每个用户进程都有它们专属的页表，两者在寻址范围方面的区别，我们将在内存与进程一章中详细介绍。

图 1-38~ 图 1-40 中所发生动作的相应代码如下：

```
// 代码路径：boot/head.s
   movl $pg3+4092,%edi
   movl $0xfff007,%eax           /*  16Mb-4096 + 7 (r/w user,p) */
   std
1: stosl                         /* fill pages backwards-more efficient :-) */
   subl $0x1000,%eax
   jge 1b
```

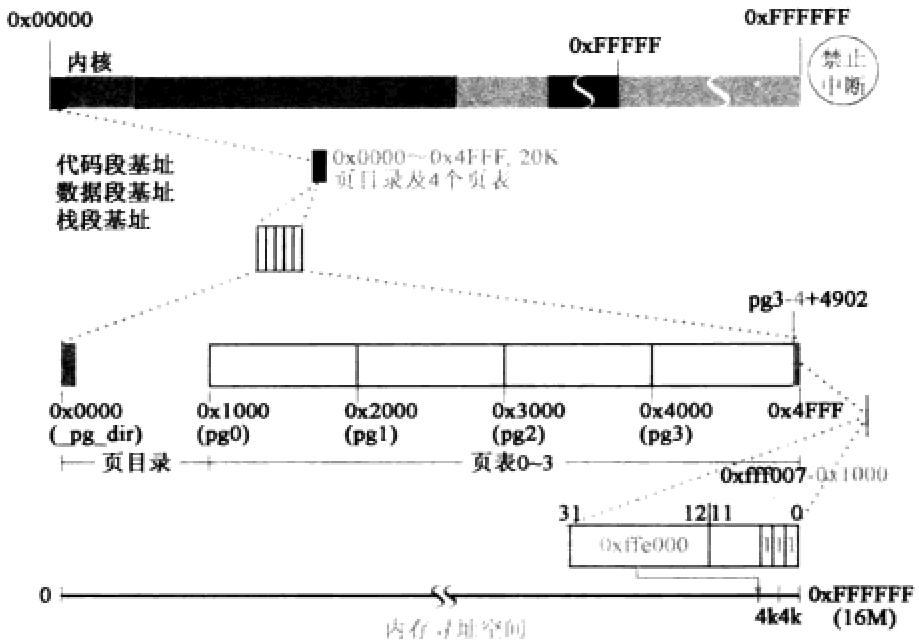


图 1-39 设置页表

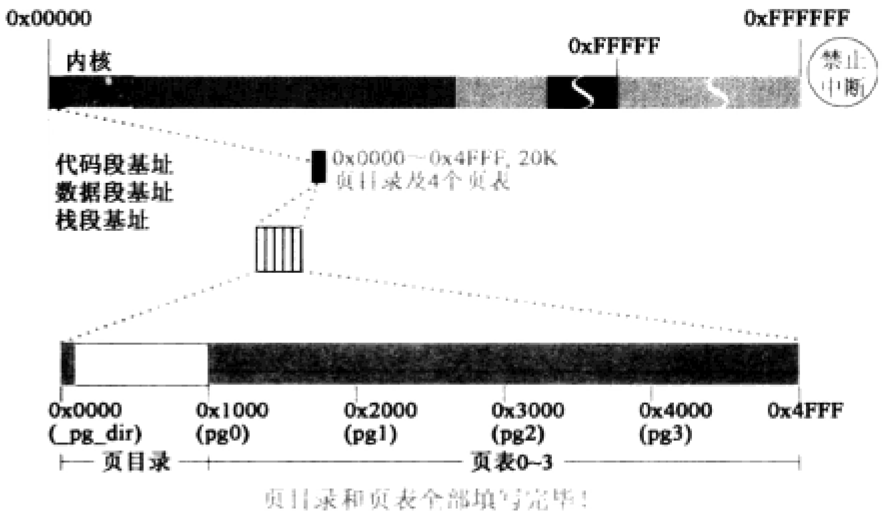


图 1-40 页目录表和页表设置完毕的状态

这些工作完成后，内存中的布局如图 1-41 所示。可以看出，只有 184 个字节的剩余代码，由此可见在设计 head 程序和 system 模块时，其计算是非常精确的，对 head.s 的代码量的控制非常到位。

head 程序已将页表设置完毕了，但分页机制的建立还没有完成。需要设置页目录基址寄存器 CR3，使之指向页目录表，再将 CR0 寄存器设置的最高位（31 位）置为 1，如图 1-42 的右中部 CR0 寄存器的示意图。

小贴士

PG（Paging）标志：CR0 寄存器的第 32 位，分页机制控制位。当 CPU 的控制寄存器 CR0 第 1 位 PE（保护模式）置为 1 时，可设置 PG 位为开启。在开启后，地址映射模式采取分页机制。当 CPU 的控制寄存器 CR0 第 1 位 PE（保护模式）置为 0 时，此时设置 PG 位将引起 CPU 发出异常。

CR3 寄存器：3 号 32 位控制寄存器，高 20 位存放页目录的基地址。当 CR0 中的 PG 标志置位时，CPU 使用 CR3 指向的页目录和页表进行虚拟地址到物理地址的映射。

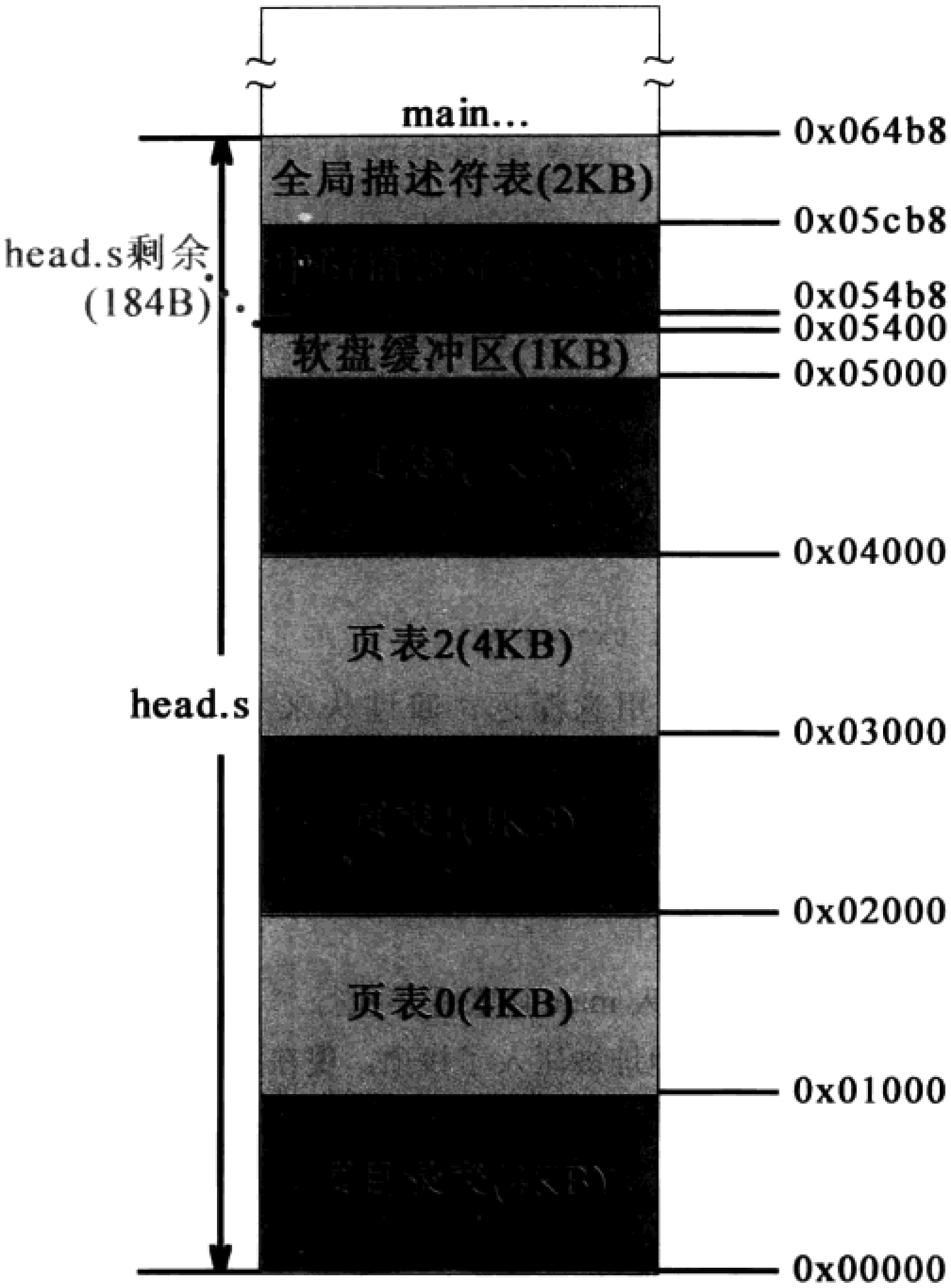


图 1-41　内存分布示意图

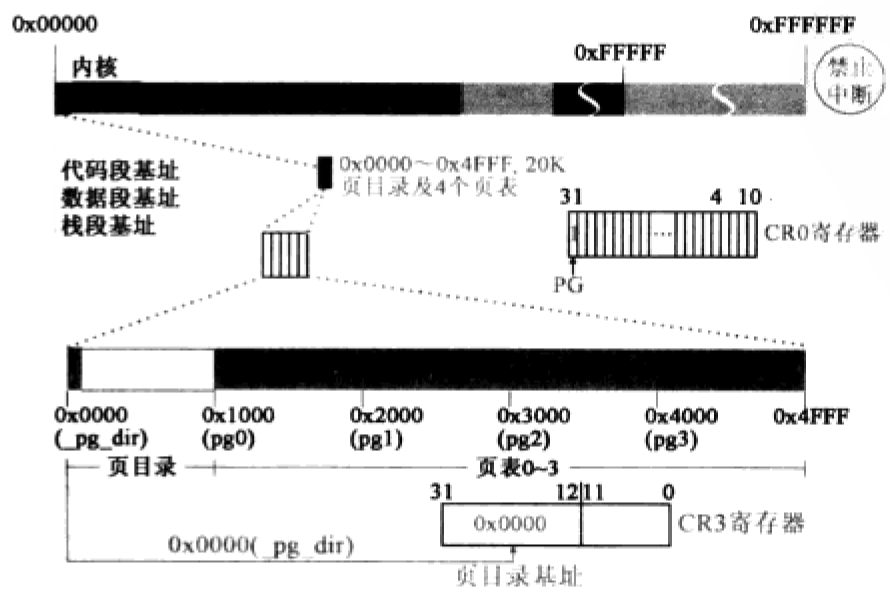


图 1-42　分页机制完成后的总体状态

执行代码如下：

```
// 代码路径：boot/head.s
    xorl %eax,%eax          /* pg_dir is at 0x0000 */
    movl %eax,%cr3          /* cr3-page directory start */
    movl %cr0,%eax
    orl $0x80000000,%eax
    movl %eax,%cr0          /* set paging (PG) bit */
```

前两行的动作是将 CR3 指向页目录表，意味着操作系统认定 0x0000 这个位置就是页目录表的起始位置，后 3 行的动作是启动分页机制开关 PG 标志置位，以启用分页寻址模式，两个动作一气呵成。到这里为止，内核的分页机制构建完毕，后续章节还会讲解如何建立用户进程的分页机制。

最重要的是

```
xorl %eax,%eax  /* pg_dir is at 0x0000 */
```

这一行代码，它看似简单，但用意深远。回过头来看，图 1-17 将 system 模块移动到 0x00000，图 1-25 在内存的起始位置建立内核分页机制，最后就是上面的这行代码，认定页目录表在内存的起始位置，三个动作联合起来为操作系统中最重要的目的——内核控制用户程序奠定了基础，这个位置是内核通过分页机制能够实现线性地址等于物理地址的唯一起始位置。我们会在后续章节逐层展开讨论。

head 程序执行最后一步：ret。跳入 main 函数程序执行。

在图 1-35 中，main 函数的入口地址被压入了栈顶，现在执行 ret 了，正好将压入的 main 函数的执行入口地址弹出给 EIP。图 1-43 中的下方标示了出栈动作。

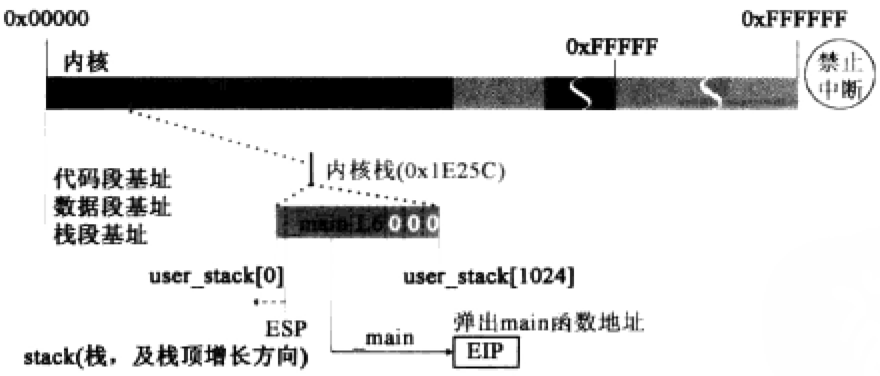


图 1-43　执行 ret，将 main 函数入口地址 pop 给 EIP

这部分代码用了底层代码才会使用的技巧，我们结合图 1-44 对这个技巧进行详细讲解。

我们先看看普通函数的调用和返回方法，因为 Linux 0.11 用返回方法调用 main 函数，返回位置和 main 函数的入口在同一段内，我们只讲解段内调用和返回。见图 1-44（仿 CALL 示意图）的上半部分，CALL 的调用与返回。

CALL 指令会将 EIP 的值自动压栈，保护返回现场，然后执行被调函数的程序。等到执行被调函数的 ret 指令时，自动出栈给 EIP 并还原现场，继续执行 CALL 的下一行指令。这是通常的函数调用方法。对操作系统的 main 来说，这个方法就有些怪异了。main 函数是操作系统的，如

果用 CALL 调用操作系统的 main 函数，那么 ret 时返回给谁呢？难道还有一个更底层的系统程序接收操作系统的返回吗？操作系统已经是最底层的系统了，所以逻辑上不成立。那么如何既调用了操作系统的 main 函数，又不需要返回呢？操作系统的设计者采用了图 1-44（仿 CALL 示意图）的下半部分所示的方法。

这个方法的妙处在于是用 ret 实现的调用操作系统的 main 函数，既然是 ret 调用，当然就不需要再用 ret 了。不过，CALL 做的压栈和跳转的动作谁来完成呢？操作系统的设计者做了一个仿 CALL 的动作，手工编写压栈和跳转代码，模仿了 CALL 的全部动作，实现了调用 setup_paging 函数。注意，压栈的 EIP 值并不是调用 setup_paging 函数的下一行指令的地址，而是操作系统的 main 函数的执行入口地址 _main。这样，当 setup_paging 函数执行到 ret 时，从栈中将操作系统的 main 函数的执行入口地址 _main 自动出栈给 EIP，EIP 指向 main 函数的入口地址，实现了用返回指令“调用”main 函数。

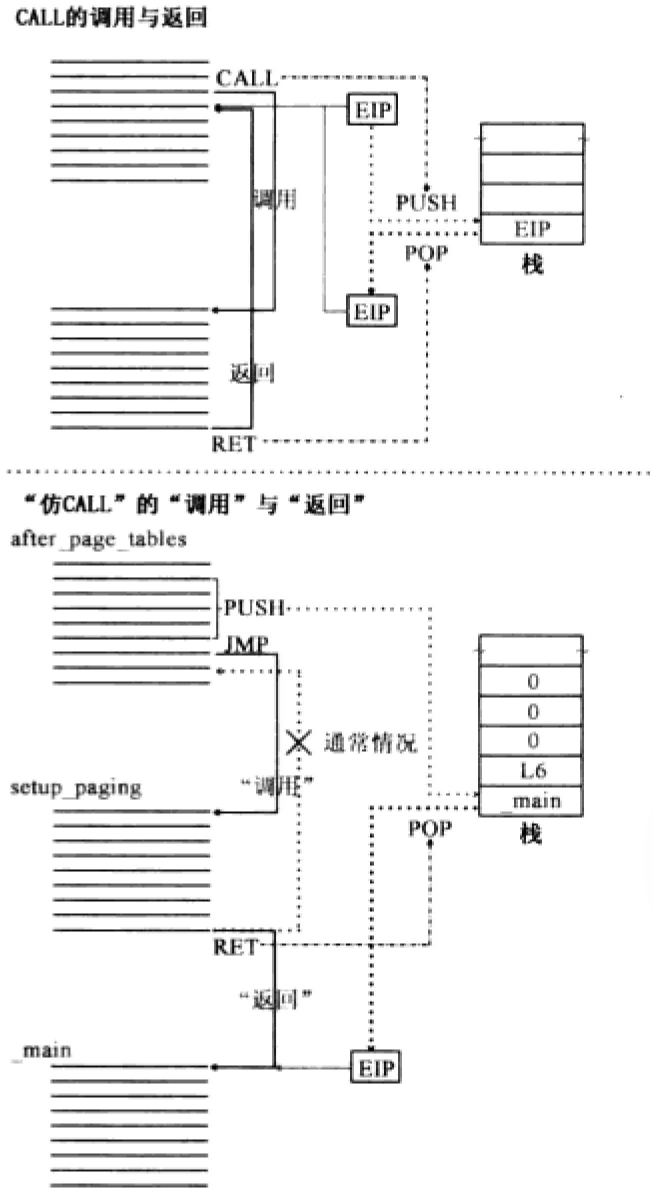


图 1-44 仿 CALL 示意图

在图 1-44 中，将压入的 main 函数的执行入口地址弹出给 CS:EIP，这句话等价于 CPU 开始执行 main 函数程序。图 1-45 在左下方标示了这个状态。

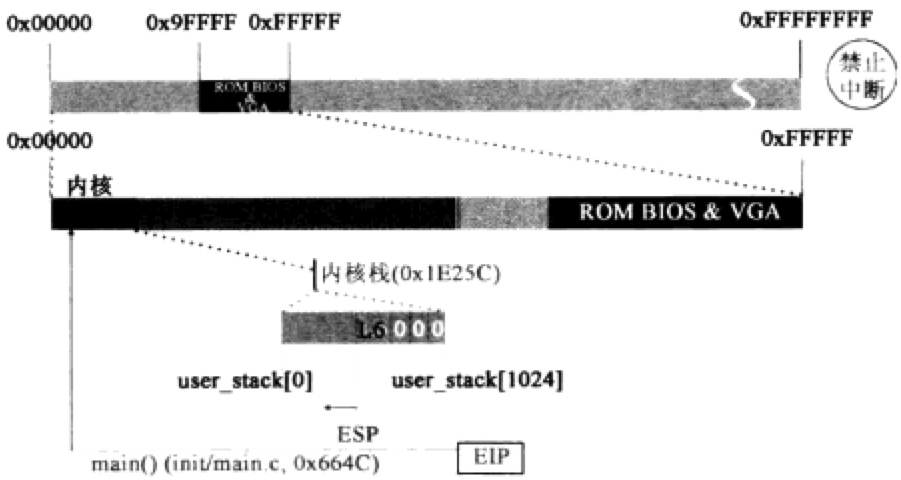


图 1-45 开始执行 main 函数

点评

为什么没有最先调用 main 函数？

学过 C 语言的都知道，用 C 语言设计的程序都有一个 main 函数，而且是从 main 函数开始执行的。Linux 0.11 的代码是用 C 语言编写的，奇怪的是，为什么在操作系统启动时先执行的是三个由汇编写成的程序，然后才开始执行 main 函数？为什么不是像我们熟知的 C 程序那样，从 main 函数开始执行呢？

通常，我们用 C 语言编写的程序都是用户应用程序，这类程序的执行有一个重要的特征，就是必须在操作系统的平台上执行，也就是说要由操作系统为应用程序创建进程，并把应用程序的可执行代码从硬盘加载到内存。现在我们讨论的是操作系统，不是普通的应用程序，这样就出现了一个问题，应用程序是由操作系统加载的，操作系统该由谁加载呢？

从前面的章节中我们知道，加载操作系统的时候，计算机刚刚加电，只有 BIOS 程序在运行，而且此时计算机处于 16 位实模式状态下，通过 BIOS 程序自身的代码形成的 16 位的中断向量表及相关的 16 位的中断服务程序，将操作系统在软盘上的第一扇区（512 字节）的代码加载到内存，BIOS 能主动操作的内容也就到此为止了。准确地说，这是一个约定，对于第一扇区代码的加载，不论是什么操作系统都是一样的。从第二扇区开始，就要由第一个扇区中的代码来完成后续的代码加载工作。

在加载工作完成后，好像仍然没有立即执行 main 函数，而是打开 A20，打开 pe 和 pg，建立 IDT、GDT……然后才开始执行 main 函数，这是为什么？

原因是，Linux 0.11 是一个 32 位的实时多任务的现代操作系统，main 函数肯定要执行的是 32 位的代码，编译操作系统代码时，是有 16 位和 32 位两个不同的编译选项的，如果选了 16 位，C 语言编译出来的代码是 16 位模式的，结果可能是一个 int 型变量只有 2 个字节，而不是 32 位的 4 个字节……这不是 Linux 0.11 想要的，Linux 0.11 要的是 32 位的编译结果，只有这样才能成为 32 位的操作系统代码，这样的代码才能用到 32 位总线（即打开 A20 后的总线），才能用到保护模式和分页，才能成为 32 位的实时多任务的现代操作系统。

开机时的 16 位实模式与 main 函数执行需要的 32 位保护模式之间有很大的差距，这个差距谁来填补？ head.s 做的就是这项工作，这期间，head 程序打开 A20，打开 pe 和 pg，废弃旧的、16 位的中断响应机制，建立新的 32 位的 IDT……这些工作都做完了，计算机已经处在 32 位的保护模式状态了，调用 32 位 main 函数的一切条件已经准备完毕，这时可顺理成章地调用 main 函数，后面的操作就可以用 32 位编译的 main 函数完成。

至此，Linux 0.11 操作系统内核启动的一个重要阶段已经完成，接下来就要进入 main 函数对应的代码了。

需要特别提示的是，此时仍处在关闭中断的状态！

1.4 本章小结

本章的内容主要分为两大部分：第一部分，加载操作系统；第二部分，为 32 位保护模式和分页模式下的 main 函数的执行做准备。

从借助 BIOS 将 bootsect.s 文件加载到内存开始，相继加载了 setup.s 文件和 system 文件，从而完成操作系统程序的加载。接下来为 32 位保护模式和分页模式下的 main 函数的执行做准备。设置 IDT、GDT、页目录表、页表，以及机器系统数据。

一切就绪后，跳转到 main 函数执行入口，开始执行 main 函数。

第 2 章

从 main 到怠速

系统达到怠速[⊖]状态前所做的一切准备工作的核心目的就是让用户程序能够以“进程”的方式正常运行，能够实现这一目的的标准包括三个方面的内容：用户程序能够在主机上进行运算、能够与外设进行交互，以及能够让用户以它为媒介进行人机交互。

系统为此做的准备工作也是逐步完成的，从 1.3 节关闭中断开始算起，一直到本章的第 2.8 节为止，都是在做这项准备工作。该过程大体可分为如下三个阶段。

- 第一阶段：创建进程 0，并让进程 0 具备在 32 位保护模式下在主机中运算的能力。

用户程序的本质就是一套“以计算机形式进行描述的用户事务处理方案”，这套方案要通过计算机的运算得到结果，而计算机的运算是由其“主机”部分完成的。可见，用户程序运行的最根本目的就是通过“主机”对其处理问题的方案进行“运算”，如果这个目的无法实现，其他所有的目的都无从谈起。

从操作系统的角度来看，这些“正在运行的用户程序”就是进程。每一个进程都是由一个已经存在的进程创建的，这个新建的进程将具备创建它的进程所具备的一切能力。因此，系统先要创建一个最原始的进程，即进程 0，并使之具备进程应该具备的最基本能力——在主机中进行运算。

2.1 节将详细讲解这个阶段的工作是如何完成的。

- 第二阶段：以进程 0 为母本创建进程 1，使进程 1 不仅仅具备进程 0 所拥有的能力，而且还能以文件的形式与外设进行数据交互。

通过对第一阶段的介绍我们得知，系统以进程 0 来创建进程 1，那么进程 1 就一定具备了在主机中进行运算的能力。但这个能力只是最基本的能力，进程在执行的时候往往还要与外设进行数据交互，在操作系统中，外设上的数据都是以文件的形式进行存储的，所以，进程 1 要在进程 0 已经具备的运算能力的基础上进一步拓展进程以文件的形式与外设进行数据交互的环境。

⊖ 怠速的意思就是操作系统已经完成了所有的准备工作，随时可以响应用户的激励。此处借用了汽车的“怠速”一词，是为了更加形象，汽车进入怠速状态，就意味着汽车已经完全启动，只要驾驶员踩油门，就可以正常行驶了。

从计算机组成的角度看，计算机主要分为两个部分：一个是主机，另一个就是外设。因此，进程 1 所具备的功能将大大提高用户进程与计算机外设的“接触面”，从而进一步完善用户进程处理问题的综合能力。

本章的 2.2 节～ 2.4 节将详细讲解这个阶段的工作是如何实现的。

- 第三阶段：以进程 1 为母本创建进程 2，使进程 2 在全面具备进程 1 所拥有的能力和环境的基础上，进一步具备支持“人机交互”的能力，最终实现怠速。

由于进程 2 是由进程 1 创建的，因此进程 2 将具备进程 1 的全部能力和环境，即能够在主机上进行运算并能够与外设进行交互。但是，不同的外设有不同的属性，也有不同的交互方式，进程 1 目前还不能全方位地适应所有的外设，比如键盘和显示器，这是用户与计算机进行沟通的交互界面，也是用户操作用户程序的平台，仅仅具备与键盘和显示器交互的最基本的能力是不够的，构建这个交互界面和平台需要更强大的能力。因此，进程 2 就在进程 1 的基础上进一步加强自己的能力，表现为：它将构建一套人机交互界面，以支持用户与计算机的交互，最终进入怠速状态。

怠速以后，进程 2 将成为创建第一个用户进程的母本，它所具备的能力将为用户程序能够以“进程”的形式正常运行提供保障。

本章的 2.5 节～ 2.8 节将完成这个阶段的工作。

本章将要讲解操作系统运行的最主要的基础工作，涉及操作系统所有基础内容的 85% 以上，是全书的重点之一，也是理解操作系统的一个难点。掌握本章内容将对学习后续的内容起到事半功倍的作用。

2.1 开中断之前的准备工作

本节将完成第一阶段的工作：创建进程 0，并让进程 0 具备在 32 位保护模式下在主机中运算的能力。这项工作从第 1 章的 1.3 节关中断后就已经开始了，其中包括：开启 A20、初始化全局描述符表、初始化中断描述符表、设置页表和页目录表，等等。这里将继续之前的准备工作，以实现第一阶段的目标。

Linux 0.11 是一个支持多进程的现代操作系统，这就意味着，各个用户进程在运行过程中，彼此不能相互干扰，这样才能保证进程在主机中正常地运算。然而，进程自身并没有一个天然的“边界”来对其进行保护，要靠系统“人为”地给它设计一套“边界”来保护它，这套边界就是系统为进程提供的进程管理信息数据结构。这套进程管理信息数据结构包括：进程管理结构 task_struct、进程槽 task[64]、全局描述符表 GDT 等。task_struct 是每个进程所独有的结构，它标识了进程的各项属性值，包括剩余时间片、进程执行状态、局部数据描述符表 LDT 和任务状态描述符表 TSS，等等。task[64] 和 GDT 是为管理多进程而提供的数据结构，task[64] 结构中存储着系统中所有进程的 task_struct 结构指针，如果系统需要对多个进程加以比较并选择，就可以通过遍历 task[64] 结构来实现；GDT 中存储着一套针对所有进程的索引结构，通过索引项，系

统可以间接地与每个进程中的 LDT 和 TSS 建立关系。

对于如何通过以上三个结构实现对保护的全方位支持这一问题，将在 2.1.9 节中详细讲解，因此这一节的内容也是本章的重中之重。它们都是由于系统对多进程的支持才存在的，如果没有多进程，它们就没有存在的必要了。

本节还将讲解系统是如何对内存、CPU、串行口、显示器、键盘、硬盘、软盘等诸多硬件进行设置的，并将这些硬件所对应的中断服务程序与中断描述符表相挂接，以此提高进程 0 的运算能力，同时也为由进程 0 以及其直接或间接创建的所有后续进程与外设的沟通构建了环境。

下面我们开始详细介绍准备工作是如何进行的。

2.1.1 复制根设备号和硬盘参数表

在对各个硬件设备进行设置和规划前，系统会先对“根设备号”和“硬盘参数表”进行备份。这是因为启动部分的初始化设置是逐步完成的，“根设备号”和“硬盘参数表”现在所占用的内存空间会在以后进行缓冲区的初始化设置时被重新规划。由于内存是主机的一部分，对内存中缓冲区的规划，必然会影响到进程 0 在主机中的运算能力的实现。因此，这时候要事先备份，以免将来被干扰。

系统用 ROOT_DEV 备份“根设备号”，用以从软盘上加载根文件系统，具体执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
    ROOT_DEV = ORIG_ROOT_DEV;
    ...
}
```

系统用 drive_info 备份硬盘参数表，它的大小为 32 字节，定义在 main.c 中，如下所示：

```
// 代码路径：init/main.c:
...
struct drive_info { char dummy[32]; } drive_info;
...
```

而每个硬盘参数表为 16 字节，所以 drive_info 能够将两个硬盘参数表全部备份。具体执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
    ...
    drive_info = DRIVE_INFO;
    ...
}
```

具体的复制动作及完成后的数据在内存中的位置如图 2-1 所示。

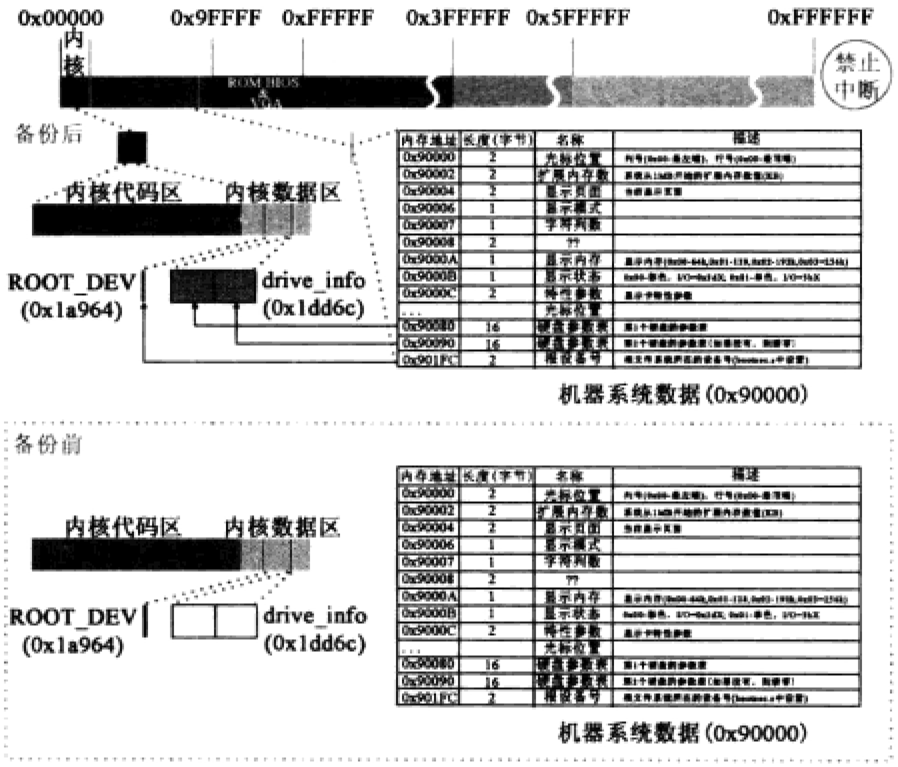


图 2-1 复制根设备号和硬盘参数表

2.1.2 物理内存规划格局

进程 0 在主机中的运算主要是通过 CPU 和内存相互配合工作而得以实现的，因此，这里系统将对主机中物理内存的使用及管理进行规划，这样才能为进程 0 具备运算能力打下基础。具体操作是这样的：除了 1MB 以内的内核区之外，其余物理内存要完成的工作是不同的，“主内存区”主要用来承载进程的相关信息，包括进程管理结构、进程对应的程序等；“缓冲区”主要作为主机与外设进行数据交互的中转站；“虚拟盘区”是一个可选的区域，如果选择使用虚拟盘，就可以将外设上的数据先复制到虚拟盘区，然后再使用。由于从内存中操作数据的速度远高于外设，因此这样可以提高系统执行效率。

这里系统就要对内存中的这三种不同性质的区域，在大小、位置及管理方式上进行规划。

先根据内存条大小对“缓冲区”和“主内存区”的位置和大小进行初步设定。分别针对内存条：大于 16MB、大于 12MB 且小于等于 16MB、大于 6MB 且小于等于 12MB、小于等于 6MB 这四种情况进行设定。具体执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
     ...
     memory_end = (1<<20) + (EXT_MEM_K<<10);
    memory_end &= 0xfffff000;
    if (memory_end > 16*1024*1024)
        memory_end = 16*1024*1024;
    if (memory_end > 12*1024*1024)
        buffer_memory_end = 4*1024*1024;
    else if (memory_end > 6*1024*1024)
        buffer_memory_end = 2*1024*1024;
```

```
        else
            buffer_memory_end = 1*1024*1024;
        main_memory_start = buffer_memory_end;
         ...
    }
```

其中 memory_end 为主内存区的末端位置，main_memory_start 为主内存区的起始位置。buffer_memory_end 为缓冲区的末端位置，至于缓冲区的起始位置，我们将在 2.1.13 节中详细介绍。通过 main_memory_start = buffer_memory_end; 这一行可以看出，现在缓冲区末端的位置就是主内存的起始位置。

图 2-2 展示了设置完成后的物理内存的规划格局。

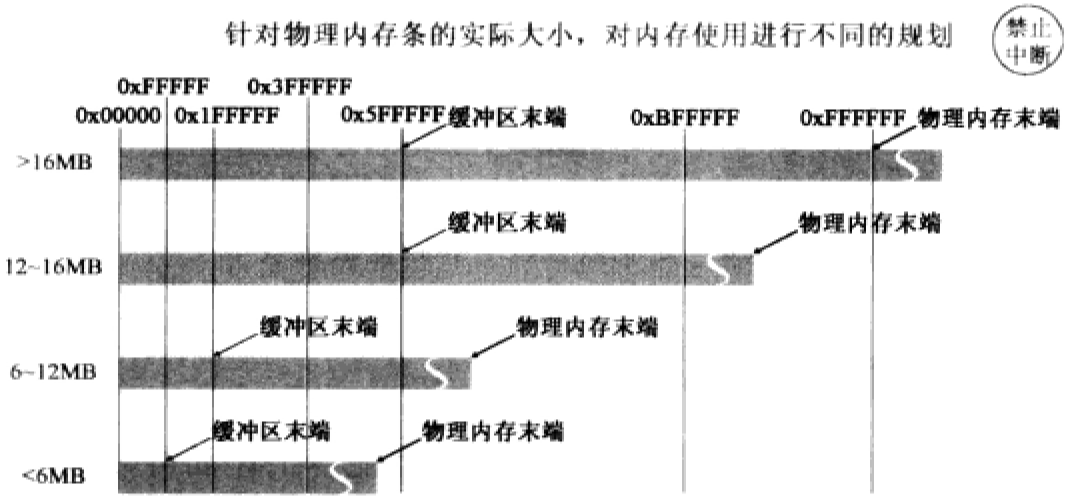


图 2-2　物理内存规划格局

2.1.3　虚拟盘设置与初始化

接下来将对外设中的虚拟盘区进行设置。检查 makefile 文件中“虚拟盘使用标志”是否被设置，以此确定本系统是否使用了虚拟盘。我们假设系统需要虚拟盘，且将虚拟盘大小设置为 2MB，表现为在主内存起始处为虚拟盘开辟一段 2MB 的内存空间。

于是调用 rd_init 函数，开始对虚拟盘进行设置，具体执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
     ...
     #ifdef RAMDISK
             main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
#endif
     ...
}
// 代码路径：kernel/ramdisk.c:
long rd_init(long mem_start, int length)
{
     ...
    blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
    ...
    for (i=0; i < length; i++)
```

```
            *cp++ = '\0';
    return(length);
}
```

进入该函数后，先要将虚拟盘区的请求项处理函数 do_rd_request 与系统挂接，即与图 2-3 左上部分的请求项函数控制结构 blk_dev[7] 的第一项挂接。有些书中将 blk_dev[7] 翻译为块设备结构，我们认为该结构的主要功能是将某一块设备与它对应的请求项处理函数挂钩，因此叫做请求项函数控制结构更为合适。这里所提到的请求项将在本章 2.1.6 节中详细解释。这个挂接动作意味着今后系统要通过调用 do_rd_request 这个函数才能处理与虚拟盘区相关的请求项操作。挂接之后，将虚拟盘所在的内存区域全部初始化为 '\0'，该过程如图 2-3 右上部分所示。

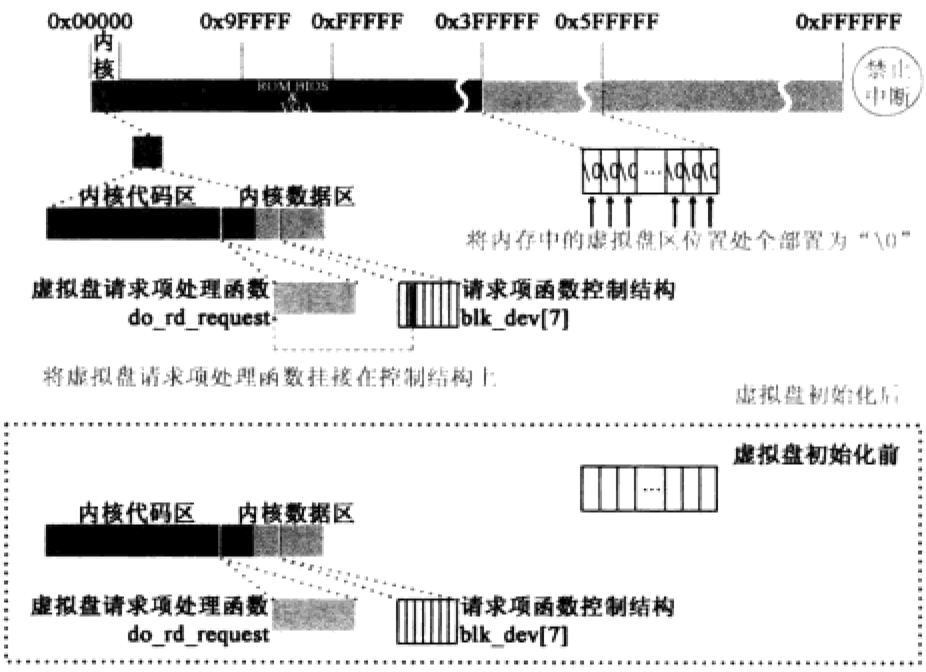


图 2-3 虚拟盘设置与初始化

最后将虚拟盘区的长度值返回，这个返回值将用来重新设置主内存区的起始位置。

2.1.4 内存管理结构 mem_map 初始化

对主内存区起始位置的重新确定，标志着主内存区和缓冲区的位置和大小已经全都确定了，于是系统开始调用 mem_init 函数。先对主内存区的管理结构进行设置，该过程如图 2-4 所示。

具体执行代码如下：

```
//代码路径：init/main.c:
void main(void)
{
...
mem_init(main_memory_start,memory_end);
...
}

mm/memory.c:
void mem_init(long start_mem, long end_mem)
{
...
```

```
for (i=0 ; i<PAGING_PAGES ; i++)
mem_map[i] = USED;
...
while (end_mem-->0)
mem_map[i++]=0;
}
```

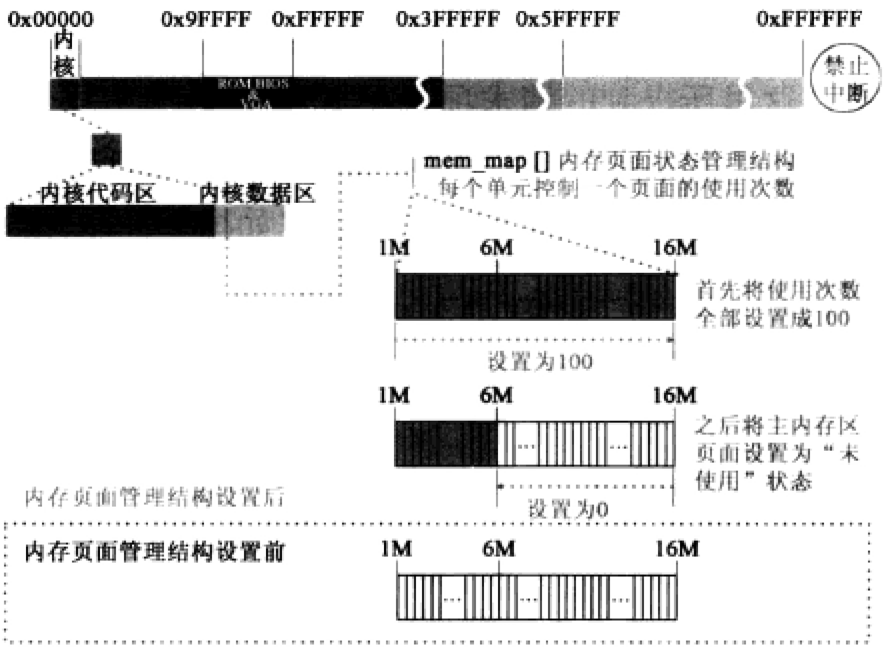


图 2-4　内存管理结构 mem_map 初始化

系统对 1MB 以上的内存都是分页管理的，于是系统就通过一个叫做 mem_map 的数组记录每一个页面的使用次数。先将所有的内存页面使用次数均设置成 100，然后再依据主内存的起始位置和终止位置将处于主内存中的所有页面的使用次数全部清 0，系统以后只把使用次数为 0 的页面视为空闲页面。

那么，为什么系统不把 1MB 以内的内存当作用户空间进行分页管理呢？这是因为，这样管理并记录页面使用次数本身就意味着进程可以对这部分被管理的内存进行操作，而系统是不可能允许进程对 1MB 以下的内核代码区进行操作的，所以也就不能对 1MB 以内的内存进行这样的管理了。至于分页管理内存的深层次原因，将在第 8 章的高端点评中详细介绍。

2.1.5　异常处理类中断服务程序挂接

CPU 是主机中关键的组成部分，进程 0 在主机中的运算肯定离不开 CPU，而 CPU 在参与运算过程中免不了进行"异常处理"，这些异常处理都需要具体的服务程序来执行。因此，这里调用 trap_init 函数将"异常处理"一类的中断服务程序与中断描述符表进行挂接，开始逐步地重建中断服务体系，以此来支持进程 0 在主机中的运算，挂接的具体过程及异常处理类中断服务程序在中断描述符表中所占用的位置如图 2-5 所示。

"异常处理"就是由 CPU 探测到的一些不可预知的错误而导致的中断，具体包括：除 0 错误、溢出错误、边界检查错误、缺页错误，等等。进入 trap_init 函数后，系统主要通过调用 set_trap_gate 这样的宏函数来实现挂接。具体的挂接过程是：将每个异常处理中断服务程序的地址填写到中断描述符表中指定的位置，比如"除 0 错误"中断处理程序的挂接，执行代码如下：

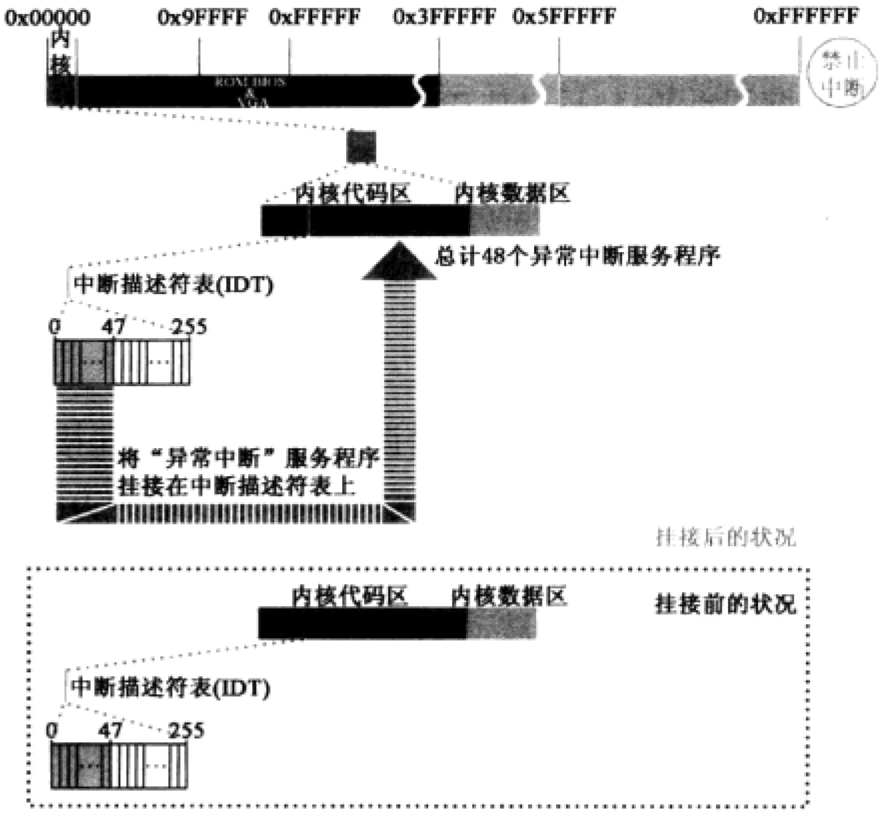


图 2-5 异常处理类中断服务程序挂接

```
// 代码路径：init/main.c:
void main(void)
{
...
trap_init();
...
}

// 代码路径：kernel/traps.c:
void trap_init(void)
{
...
set_trap_gate(0,&divide_error);
...
}
```

参数中的 0 表示将该函数地址挂接在中断描述符表的第 0 项位置处，&divide_error 代表了"除 0 错误"处理函数的地址。

另外，挂接工作除了要将服务程序的地址值载入描述符表外，还要对每个与之建立关系的描述符进行设置。每个中断描述符为"8"个字节，挂接的时候会将中断处理程序的地址值按照一定的规则进行拆分，然后分散存储在这 8 个字节中。同时还要将一些针对本中断的属性信息记录在这 8 个字节的其余空间中，比如说中断的优先权值等，从而实现挂接，具体的执行代码可以参考 system.h 中 set_trap_gate 的宏展开，如下所示：

```
// 代码路径：include/system.h:
...
#define _set_gate(gate_addr,type,dpl,addr) \
__asm__ ("movw %%dx,%%ax\n\t" \
    "movw %0,%%dx\n\t" \
    "movl %%eax,%1\n\t" \
```

```
        "movl %%edx,%2" \
        : \
        : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
        "o" (*((char *) (gate_addr))), \
        "o" (*(4+(char *) (gate_addr))), \
        "d" ((char *) (addr)),"a" (0x00080000))
...
#define set_trap_gate(n,addr) \
    _set_gate(&idt[n],15,0,addr)
...
```

其他异常处理服务程序的挂接方式与“除 0 错误”服务程序的挂接方式大体一致。本节后面所介绍的所有中断服务程序与中断描述符表的挂接大体都是以这种方式进行的，32 位保护模式下的中断服务体系就是通过不断地建立这种挂接关系形成的。

这种 32 位中断服务体系是为适应一种被动响应中断信号的机制而建立的。我们先来观察一下这套中断服务体系的特点，它们的执行路线是这样的：

一方面，硬件产生信号，并传达给 8259A，8259A 对信号进行初步处理，然后视 CPU 执行情况传递中断信号。

另一方面，如果 CPU 没有接收到信号，就不断地处理正在执行的程序；如果接收到信号，就打断正在执行的程序并通过中断描述符表找到具体的中断服务程序，让其执行，执行完后，从刚才打断的程序点继续执行。如果没有接收到中断信号，就不断地执行，如果又接收到了，就再次处理中断。可见 CPU 采取的是一套“被动响应”的机制来处理中断的，这样 CPU 就可以把全部精力都放在为用户程序服务上，对于随时可能产生而又不可能时时都产生的中断信号，不用刻意去考虑，这就提高了操作系统的综合效率。

最原始的设计其实不是这样，那时候 CPU 每隔一段时间就要对所有硬件进行轮询，以检测它刚才工作的过程中有没有信号产生，这样就分散了 CPU 处理用户程序的精力，从而降低了系统的综合效率。不仅如此，每一个硬件在一次轮询的过程中产生多少信号也不确定，所以这个硬件还需要存储自身产生的信号，这就提高了硬件设计的要求。可见，CPU 以“主动轮询”的方式来处理信号是非常不划算的，以“被动响应”模式替代“主动轮询”模式来处理中断问题是现代操作系统之所以称之为“现代”的一个重要标志。

2.1.6 初始化块设备请求项结构

进程 0 以及其后续创建的进程要想与块设备进行沟通，就要依托系统为此构建的工作环境。主机中内存的缓冲区与外设中的块设备（如软盘和硬盘等）就是通过“请求项管理结构 request[32]”建立关系的，这里对请求项管理结构进行设置将成为系统构建此环境中的一部分工作。

请求项管理结构 request[32] 是块设备与缓冲区打交道的唯一中转站，系统会对缓冲块操作与否进行权衡，并把需要操作的缓冲块登记在请求项这个账本上。块设备得到操作指令后，只根据请求项中的记录来决定当前需要处理哪个设备的哪个逻辑块，可见请求项结构充当了中转站的角色。另外，从用户使用操作系统的心理角度来讲，用户更急于看到从设备上读出来的数据，至

于数据的写入速度，相比之下用户并没有更多的关注，所以请求项结构设计的时候，会考虑给“读”数据操作更高的空间占用率，比“写”数据操作多一部分空间，具体方法是 request[32] 中的前 2/3 部分，由读 / 写请求共享；后 1/3 的空间，专门留给读请求。此结构的每一项包括：使用的设备号、是读操作还是写操作、操作时产生的错误、对应的缓冲块指针、下一请求项指针，等等。有些数据只有在请求项具体使用的时候才有意义，比如对应的缓冲块指针，在没有和具体的缓冲块建立关系之前，这个数据是没有意义的。因此，并不是全部都初始化，只将“使用的设备号”和“下一请求项指针”这两项进行了初始化，这是调用 blk_dev_init 函数完成的，执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
...
blk_dev_init();
  ...
}

// 代码路径：kernel/blk_dev/ll_rw_block.c:
void blk_dev_init(void)
{
...
for (i=0 ; i<NR_REQUEST ; i++) {
            request[i].dev = -1;
            request[i].next = NULL;
    }
}
```

request[i].dev = −1 说明了这个请求项还没有具体对应哪个设备，这个标志将来会被用来判断对应该请求项的当前设备是否空闲。request[i].next = NULL 说明这个请求项还没有与请求项队列进行挂接。初始化的过程和效果如图 2-6 所示。

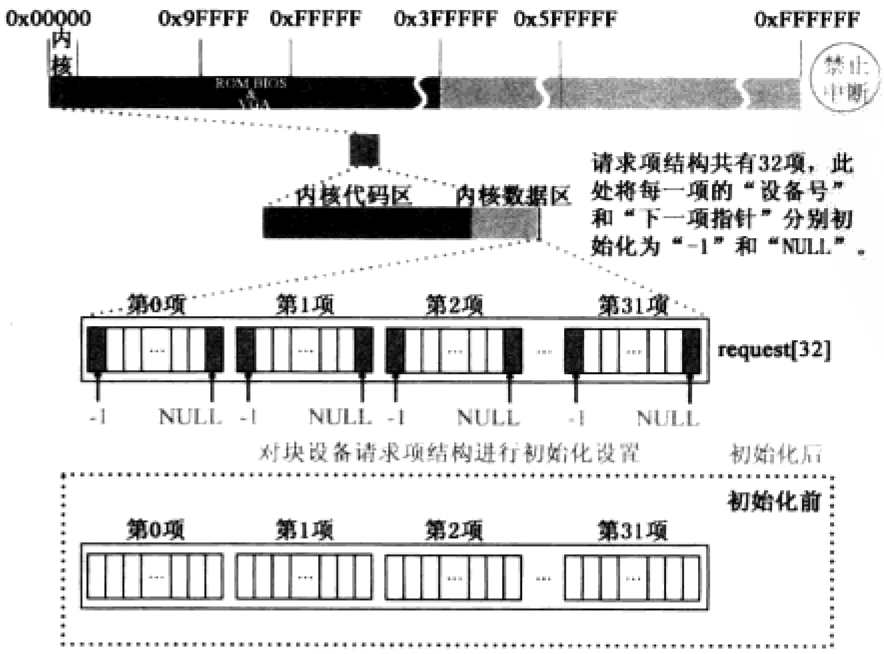


图 2-6　初始化块设备请求项结构

2.1.7 与建立人机交互界面相关的外设的中断服务程序挂接

进程 0 以及其后续创建出的进程要想与外设中的串行口（可以连接鼠标）、显示器以及键盘进行沟通，就要依托系统为此构建的工作环境。这里要讲解的对串行口、显示器、键盘的设置，以及与此相关的中断服务程序与中断描述符表的挂接，将成为系统构建此环境中的一部分工作。

在这个设置过程中，与这些外设相关的一些中断服务程序也将被挂接到中断描述符表中，这将进一步构建中断服务体系。在 main 中调用的 tty_init 函数中，继续调用 rs_init 函数来设置串行口，调用 con_init 函数来设置显示器，具体执行代码如下：

```
//代码路径：init/main.c:
void main(void)
{
...
tty_init();
...
}

//代码路径：kernel/chr_dev/tty_io.c:
void tty_init(void)
{
rs_init();
con_init()
}
```

下面将详细讲解如何对串行口、显示器和键盘进行设置。

1. 对串行口进行设置

首先，把两个串行口中断服务程序与中断描述符表相挂接；然后，根据 tty_table 数据结构中的内容对这两个串行口进行初始化设置，其中包括：设置线路控制寄存器的 DLAB 位、设置发送的波特率因子、设置 DTR 和 RTS，等等；最后，允许主 8259A 芯片的 IRQ3 和 IRQ4 发送中断请求。挂接的具体过程及挂接后的效果如图 2-7 上半部分所示。

具体执行代码如下所示：

```
//代码路径：kernel/chr_dev/serial.c:
void rs_init(void)
{
    set_intr_gate(0x24,rs1_interrupt);
    set_intr_gate(0x23,rs2_interrupt);
    init(tty_table[1].read_q.data);
    init(tty_table[2].read_q.data);
    outb(inb_p(0x21)&0xE7,0x21);
}
```

两个串行口中断处理程序与中断描述符表的挂接函数 set_intr_gate 与 2.1.5 节中介绍过的 set_trap_gate 函数大体类似，已经进行了详细说明。

2. 对显示器进行设置

根据机器系统数据中提供的显卡属性信息“单色还是彩色”来设置其他属性信息。由于在

Linux 0.11 那个时代，大部分显卡的属性都是单色的，所以我们现在假设显卡的属性是单色的 EGA，那么显存的位置就要被设置为 0xb0000 ～ 0xb8000，索引寄存器端口被设置为 0x3b4，数据寄存器端口被设置为 0x3b5；然后，再将显卡的属性——EGA 这三个字符，显示在屏幕上；另外，再初始化一些用于滚屏的变量，其中包括滚屏的起始显存地址，滚屏结束显存地址，最顶端行号及最底端行号。具体如图 2-8 所示。

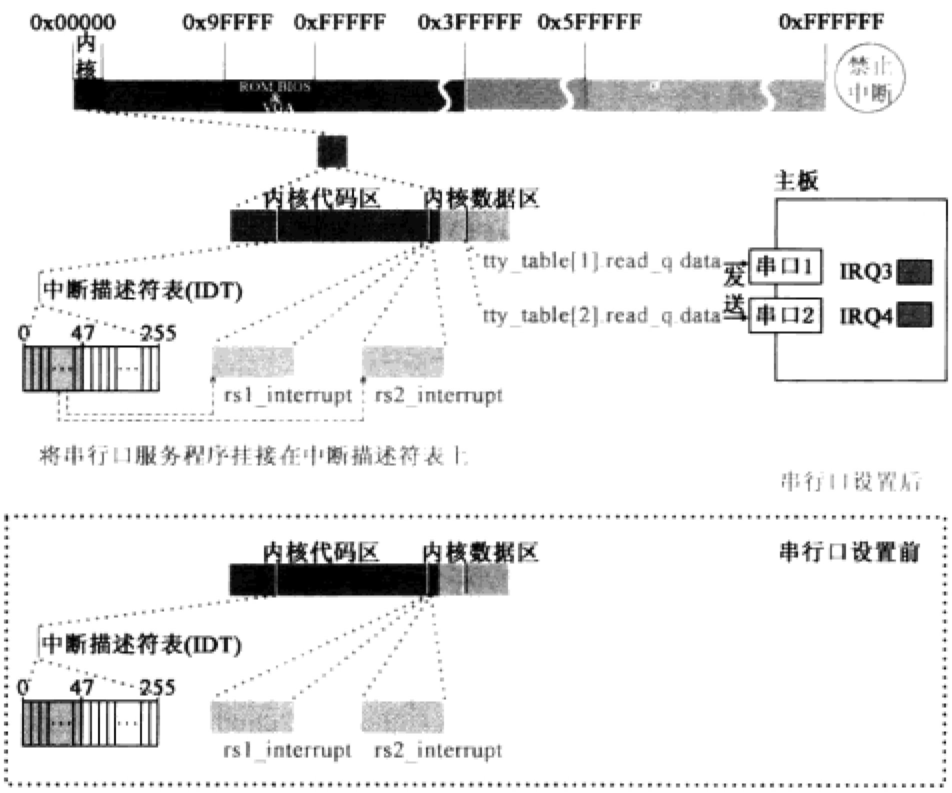


图 2-7　串行口中断服务程序挂接

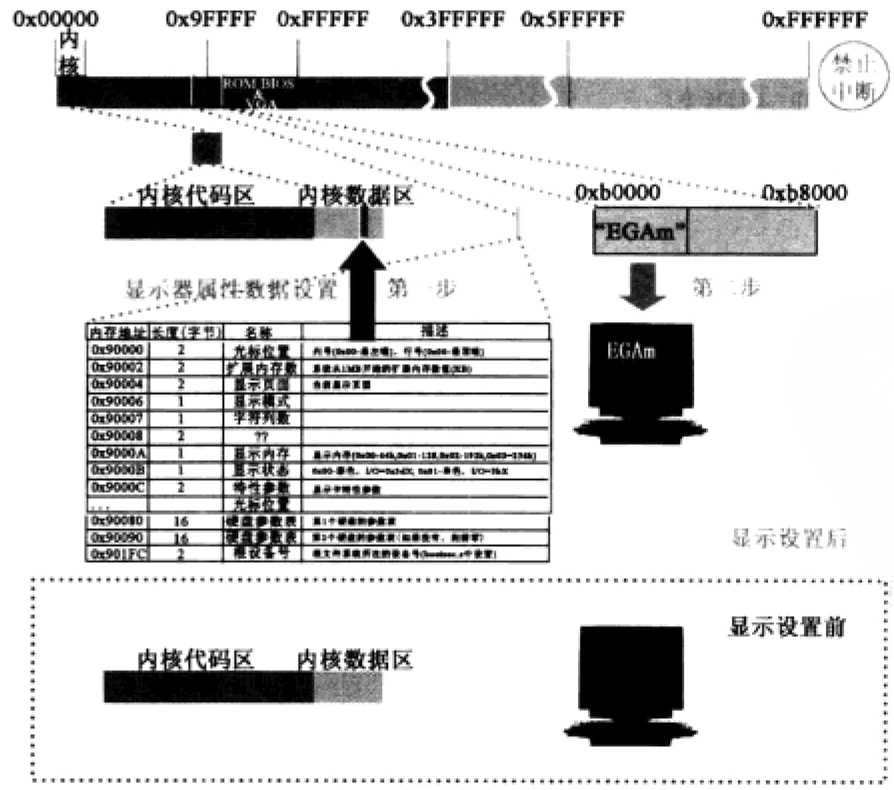


图 2-8　显示器设置

具体的执行代码如下：

```
// 代码路径：kernel/chr_dev/console.c:
void con_init(void)
{
     ...
     video_num_columns = ORIG_VIDEO_COLS;
     video_size_row = video_num_columns * 2;
     video_num_lines = ORIG_VIDEO_LINES;
     video_page = ORIG_VIDEO_PAGE;
     video_erase_char = 0x0720;
     if (ORIG_VIDEO_MODE == 7)
     {
           video_mem_start = 0xb0000;
              video_port_reg = 0x3b4;
              video_port_val = 0x3b5;
              if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
              {
                     video_type = VIDEO_TYPE_EGAM;
                     video_mem_end = 0xb8000;
                     display_desc = "EGAm";
              }
           ...
}

display_ptr = ((char *)video_mem_start) + video_size_row - 8;
    while (*display_desc)
    {
            *display_ptr++ = *display_desc++;
            display_ptr++;
    }

    origin  = video_mem_start;
    scr_end = video_mem_start + video_num_lines * video_size_row;
    top     = 0;
    bottom  = video_num_lines;

    gotoxy(ORIG_X,ORIG_Y);
...
}
```

3. 对键盘进行设置

首先，将键盘中断服务程序与中断描述符表相挂接；然后，取消 8259A 中对键盘中断的屏蔽，允许 IRQ1 发送中断信号，这样，键盘便能够使用了。具体执行代码如下：

```
// 代码路径：kernel/chr_dev/console.c:
void con_init(void)
{
...
set_trap_gate(0x21,&keyboard_interrupt);
    outb_p(inb_p(0x21)&0xfd,0x21);
    ...
}
```

键盘中断处理程序与中断描述符表的挂接函数 set_trap_gate 在 2.1.5 节中已经详细说明。如图 2-9 所示。

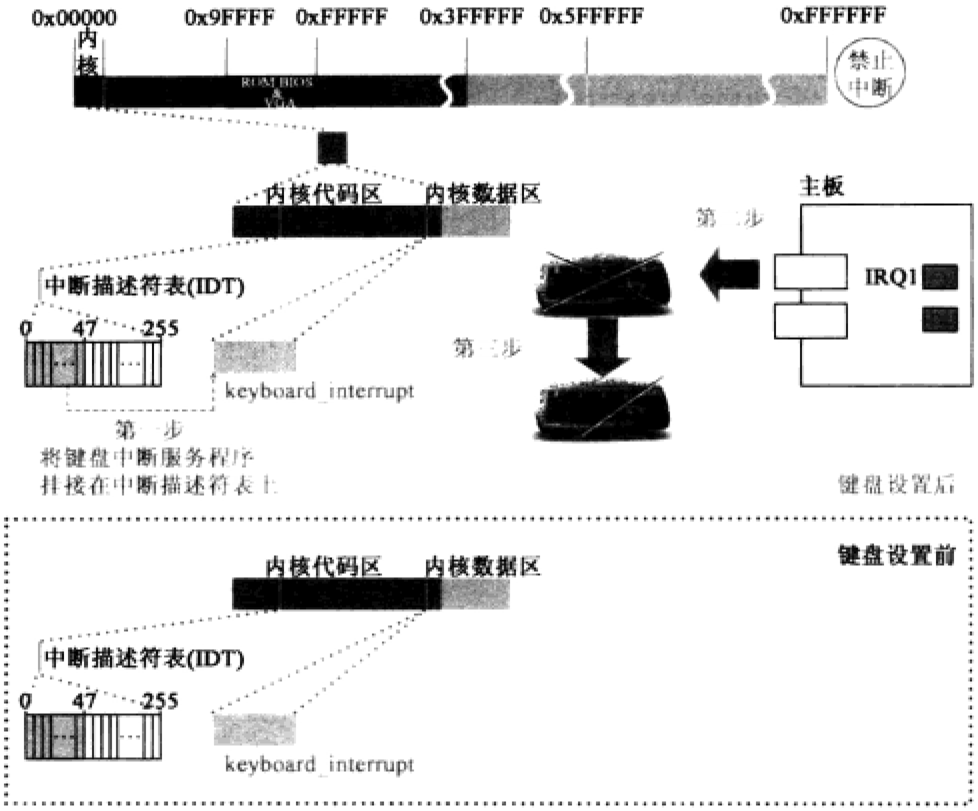


图 2-9　键盘设置

2.1.8　开机启动时间设置

进程 0 在主机中正常运算需要具备对时间的掌控能力。比如，运算多少时间后就需要切换执行另外的逻辑运算，这是在程序执行时经常会遇到的情况，系统在此设置开机启动时间就是在为进程 0 以及后续创建的进程能够掌控时间奠定基础。

这个开机启动时间将作为其他与时间相关计算的基础。在操作系统中，除了程序运算之外，很多事务的处理也都要用到时间，比如说，文件修改的时间、文件最近访问的时间、i 节点自身的修改时间等，而且用户也需要了解相关的时间信息。有了这个基础，其他时间就可以根据这个基础推算出来。

具体执行步骤是这样的：CMOS 是主板上一个只读的芯片，系统通过调用 time_init 函数，先对它上面记录的时间数据进行采集，提取不同等级的时间要素，比如秒 time.tm_sec、分钟 time.tm_min、年 time.tm_year 等。具体执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
...
time_init();
    ...
}
static void time_init(void)
{
...
do {
            time.tm_sec = CMOS_READ(0);
            time.tm_min = CMOS_READ(2);
```

```
            time.tm_hour = CMOS_READ(4);
            time.tm_mday = CMOS_READ(7);
            time.tm_mon = CMOS_READ(8);
            time.tm_year = CMOS_READ(9);
    } while (time.tm_sec != CMOS_READ(0));
  ...
}
```

然后再对这些要素进行整合，并最终得出开机启动时间：startup_time，执行代码如下：

```
// 代码路径：init/main.c:
static void time_init(void)
{
...
BCD_TO_BIN(time.tm_sec);
    BCD_TO_BIN(time.tm_min);
    BCD_TO_BIN(time.tm_hour);
    BCD_TO_BIN(time.tm_mday);
    BCD_TO_BIN(time.tm_mon);
    BCD_TO_BIN(time.tm_year);
    time.tm_mon--;
    startup_time = kernel_mktime(&time);
}
```

上述的整个过程及开机启动时间在内存中的存储位置如图 2-10 所示。

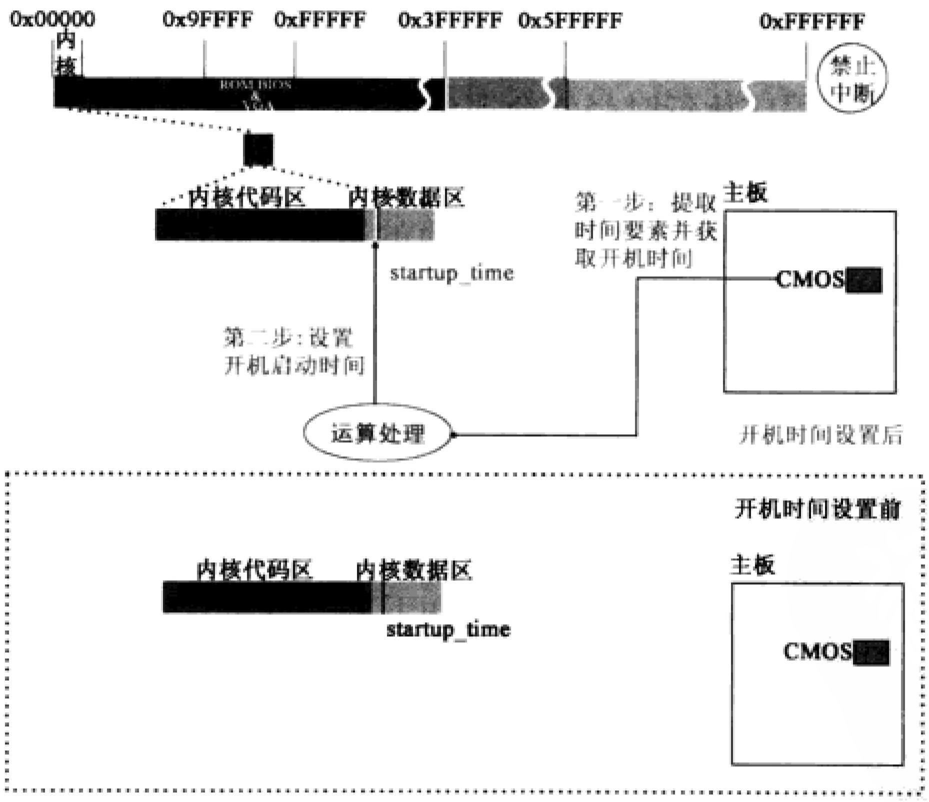


图 2-10　开机启动时间设置

2.1.9　系统开始激活进程 0

下面讲解的内容是对进程 0 能够在主机中正常运算影响最为重要和深远的一步，主要包含如下三个方面的内容。

1. 系统先要将进程 0 激活，使其具备运算及创建其他进程的能力。

进程 0 管理结构 task_struct 的母本已经在代码设计阶段事先设计好了，但这并不代表进程 0 已经可用了，还要将进程 0 的管理结构中的数据结构与全局描述符表相挂接，并对全局描述符表、进程槽以及与进程调度相关的寄存器进行初始化设置，这样才能将进程 0 激活，使进程 0 具备运算及创建其他进程的能力。

2. Linux 0.11 作为一个现代操作系统，其最重要的标志就是能够支持多进程轮流执行，这就导致在进程的运算中，它要时刻具备参与多进程轮询的能力。因此，系统在这里对时钟中断进行设置，以便在进程 0 激活后，为进程 0 以及它后续创建出来的进程在运算时能够参与轮询奠定基础。

3. 进程 0 要具备处理系统调用的能力。每个进程在运算时都有可能需要与内核进行交互，而交互的端口就是系统调用程序。系统通过函数 set_system_gate 将 system_call 与中断描述符表相挂接，这样进程 0 就具备了处理系统调用的能力。这个 system_call 就是系统调用的总入口。

进程 0 只有具备了以上三种能力才能保证其在主机中正常地运行，并将这些能力遗传给后续要建立的进程。

这三点的实现都是从调用 sched_init 函数开始的，具体代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
...
sched_init();
    ...
}
```

2.1.10　进程相关事务初始化设置

前面已经提过，进程 0 的管理信息 task_struct 是在代码设计阶段就事先设置好的，系统现在要把其中的“任务状态描述符表”和“局部数据描述符表”挂接到全局描述符表 GDT 中，执行代码如下：

```
// 代码路径：kernel/sched.c:
void sched_init(void)
{
...
set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    ...
}
```

之后再对全局描述符表 GDT 和进程管理结构 task[64] 中除了与进程 0 相关的位置外，全部进行清空设置，执行后的效果如图 2-11 上半部分中的 GDT 和 task[64] 所示，执行代码如下：

```
// 代码路径：kernel/sched.c:
void sched_init(void)
{
...
p = gdt+2+FIRST_TSS_ENTRY;
    for(i=1;i<NR_TASKS;i++) {
        task[i] = NULL;
        p->a=p->b=0;
        p++;
        p->a=p->b=0;
        p++;
    }
  ...
}
```

最后，再将全局描述符表中关于进程 0 的“任务状态描述符”和“局部数据描述符”分别记录在 CPU 中的“任务状态寄存器”和“局部数据寄存器”中，执行代码如下：

```
// 代码路径：kernel/sched.c:
void sched_init(void)
{
    ...
    ltr(0);
    lldt(0);
    ...
}
```

上述任务的执行过程以及执行完成后与进程 0 相关的管理结构在全局描述符表 GDT 和进程管理结构 task[64] 中的位置如图 2-11 所示。

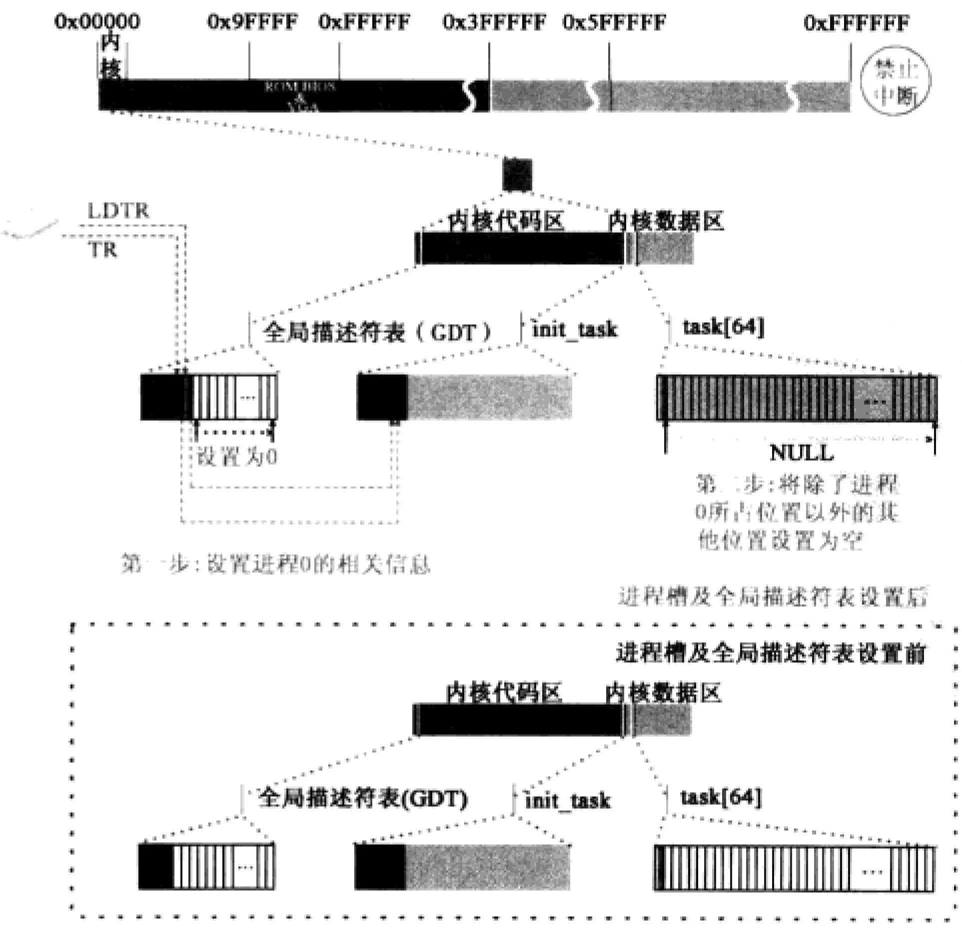


图 2-11　进程相关事务初始化设置

2.1.11 时钟中断设置

上文已经设置好了与进程 0 相关的管理结构，接下来就对时钟中断进行设置，时钟中断是进程 0 及其他由它创建的进程轮询的基础。对时钟中断进行设置的过程具体分为如下三个步骤。

（1）对支持轮询的 8253 定时器进行设置，这一步操作如图 2-12 中的第一步所示，执行代码如下：

```
// 代码路径：kernel/sched.c:
void sched_init(void)
{
    ...
    outb_p(0x36,0x43);
    outb_p(LATCH & 0xff , 0x40);
    outb(LATCH >> 8 , 0x40);
   ...
}
```

其中 LATCH 最关键，LATCH 是通过一个宏定义的，它在 sched.c 中的定义“#define LATCH (1193180/HZ)”表明，系统将每 10 毫秒发生一次时钟中断。

（2）对与轮询相关的服务程序进行设置，如图 2-12 中的第二步所示，执行代码如下：

```
// 代码路径：kernel/sched.c:
 void sched_init(void)
{
...
set_intr_gate(0x20,&timer_interrupt);
    ...
}
```

timer_interrupt 函数挂接后，在发生时钟中断时，系统就可以通过中断描述符表找到这个服务程序来进行具体的处理，这个程序的挂接也是保护模式下重建中断服务体系的一部分。

（3）将 8259A 芯片中与时钟中断相关的屏蔽码打开，执行代码如下：

```
// 代码路径：kernel/sched.c:
void sched_init(void)
{
...
outb(inb_p(0x21)&~0x01,0x21);
    ...
}
```

打开后，时钟中断就可以产生了，从现在开始，时钟中断每 1/100 秒就产生一次。由于此时处于“关中断”状态，所以这些产生的信号并不响应，但进程 0 已经具备参与进程轮询的能力。

2.1.12 系统调用服务程序挂接

将系统调用处理函数 set_system_gate 与中断描述符表相挂接。system_call 是整个操作系统中系统调用软中断的总入口，所有用户程序产生了系统调用软中断后，系统都是通过这个总入口进

一步找到具体的系统调用函数的，该过程如图 2-13 所示。

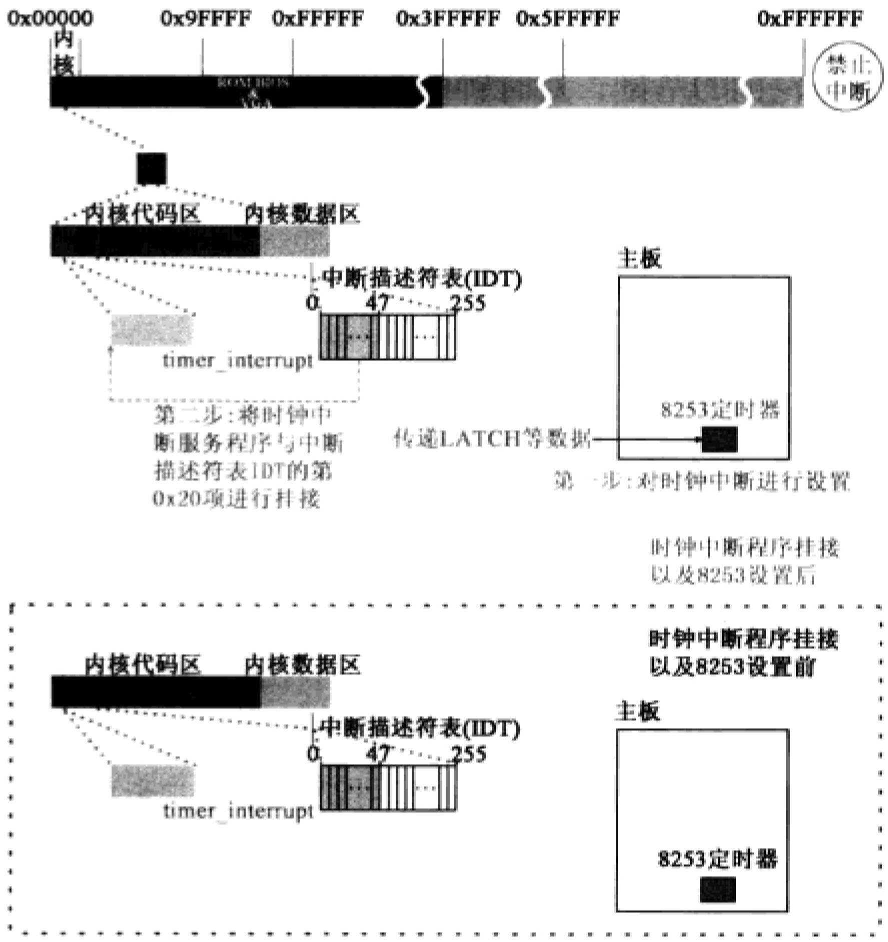


图 2-12　时钟中断设置

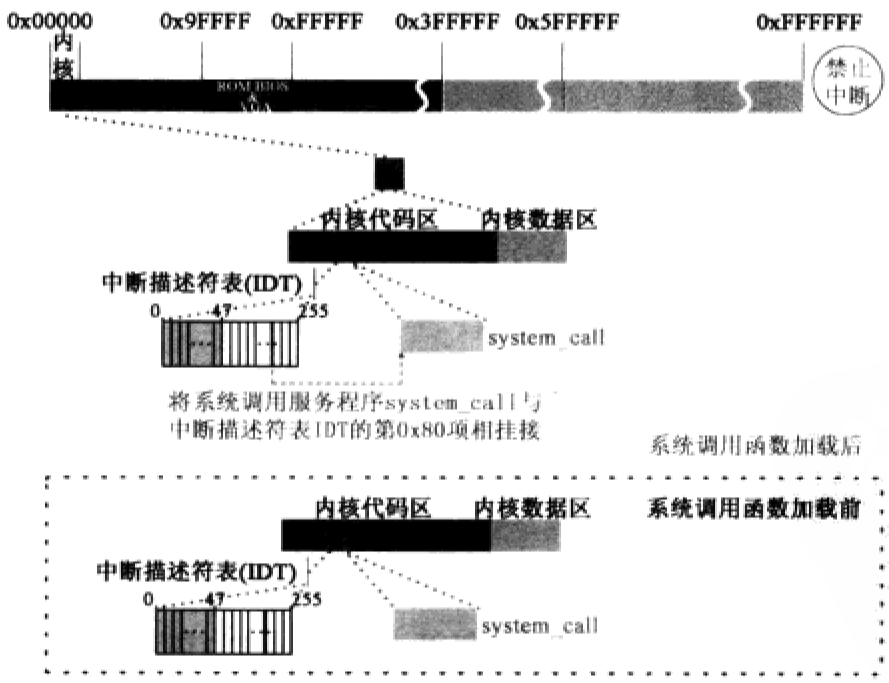


图 2-13　系统调用服务程序挂接

执行代码如下：

```
// 代码路径：kernel/sched.c:
void sched_init(void)
{
...
```

```
set_system_gate(0x80,&system_call);
}
```

系统调用函数是操作系统对用户程序最基本的支持，这行代码就是将总入口与中断描述符表相挂接，这样就奠定了这一支持的基础。这个挂接本身也是重建中断服务体系的一部分。

这个挂接还有更深层次的意义：在操作系统中，为了对内核进行保护，是不允许用户进程直接访问内核的，但处理具体事务又需要内核代码的支持，比如说：读文件和写文件等事务的处理，为了解决这个矛盾，系统的设计者提供了这样一个方案——系统调用软中断，即提供一套系统服务接口，用户进程只要想和内核打交道，就调用这套接口程序。之后，就会立即引发软中断，后面的事情就不需要用户程序负责了，而是通过另一条执行路线——由 CPU 对这个信号进行响应，通过中断描述符表找到系统调用端口，进而调用到具体的系统调用函数，来处理事务，这样矛盾就解决了。可见，所有的系统调用函数，其实也是中断服务程序，更准确地说，都是软中断服务程序。这里，系统调用服务程序与中断描述符表的挂接，就是实施这一策略的关键。

2.1.13　初始化缓冲区管理结构

本章 2.1.2 节中已经介绍过，内存是主机的重要组成部分，因此，对内存的设置与规划，将会影响到进程 0 在主机中的正常运算，2.1.2 节中已经针对主内存区进行了规划，现在开始对内存中另一部分“缓冲区”进行初始化设置，这也是对进程 0 在主机中运算的重要支持。

缓冲区是内存与外设（块设备，如硬盘等）进行数据交互的媒介。内存与外设最大的区别在于：外设（如硬盘）的作用仅仅就是对数据信息以逻辑块的形式进行断电保存，并不参与运算（因为 CPU 无法到硬盘上进行寻址）；而内存除了需要对数据进行保存以外，还要通过与 CPU 和总线的配合，进行数据运算；缓冲区则介于两者之间，它既能够对数据信息进行保存，也能够直接参与运算。这样，有了缓冲区这个媒介以后，对外设而言，它仅需要考虑与缓冲区进行数据交互是否符合要求，而不需要考虑内存如何使用这些交互的数据；对内存而言，它也仅需要考虑与缓冲区交互的条件是否成熟，而并不需要关心此时外设对缓冲区的交互情况。它们两者的组织、管理和协调将由操作系统统一进行，这样就大大降低了数据处理的维护成本。系统通过调用 buffer_init 函数对缓冲区进行设置，执行代码如下：

```
//代码路径：init/main.c:
void main(void)
{
    ...
    buffer_init(buffer_memory_end);
    ...
}
```

进入 buffer_init 函数后，具体的设置步骤如下：

系统分别通过空闲表和哈希表这两套数据结构对缓冲区进行管理，所以要分别针对这两套数据结构进行设置。先对空闲表进行设置，使之成为一个“双环链表”，其中的成员，比如对应的

设备号、引用次数、“更新”标志、“脏”标志、“锁定”标志等均被设置为 0。如图 2-14 所示，请读者注意图 2-14 顶部所示的内存的变化，在紧靠系统内核的部分，多出了一块用黑色表示的内存区域，那里面存储的就是缓冲区管理结构，由于它管理着 3000 多个缓冲块，因此它占用的内存空间的大小与内核几乎差不多。图中也对空闲表的双向链表结构给出了形象的说明。对应的代码如下：

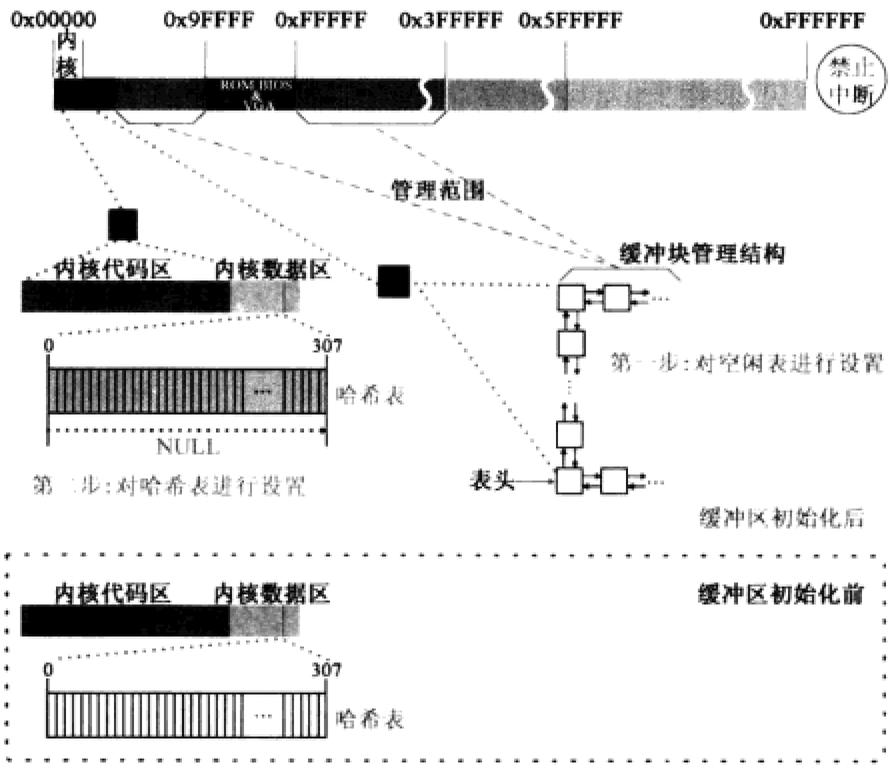


图 2-14　初始化缓冲区管理结构

```
// 代码路径：fs/buffer.c:
void buffer_init(long buffer_end)
{
    struct buffer_head * h = start_buffer;
    void * b;
    int i;

    if (buffer_end == 1<<20)
            b = (void *) (640*1024);
    else
            b = (void *) buffer_end;
    while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
            h->b_dev = 0;
            h->b_dirt = 0;
            h->b_count = 0;
            h->b_lock = 0;
            h->b_uptodate = 0;
            h->b_wait = NULL;
            h->b_next = NULL;
            h->b_prev = NULL;
            h->b_data = (char *) b;
            h->b_prev_free = h-1;
            h->b_next_free = h+1;
            h++;
```

```
            NR_BUFFERS++;
            if (b == (void *) 0x100000)
                b = (void *) 0xA0000;
    }
    h--;
    free_list = start_buffer;
    free_list->b_prev_free = h;
    h->b_next_free = free_list;
    ...
}
```

然后，对哈希表进行设置，哈希表控制数组 hash_table 总共有 307 项，全部被设置为 NULL，如图 2-14 第二步所示。执行代码如下：

```
// 代码路径：fs/buffer.c:
void buffer_init(long buffer_end)
{
    ...
    for (i=0;i<NR_HASH;i++)
        hash_table[i]=NULL;
}
```

注意看代码中 struct buffer_head * h = start_buffer; 这一行，这一行中的 start_buffer 确定了缓冲区的起始位置，这也就回答了本章 2.1.2 节中关于缓冲区起始点位置的这个问题。它是在 buffer.c 中定义的：

```
// 代码路径：fs/buffer.c:
struct buffer_head * start_buffer = (struct buffer_head *) &end;
```

这个 end 就是内核代码末端的地址，在代码编写阶段，设计者事先较难准确估算这个地址，于是就在内核模块链接期间设置了 end 这个值，然后在这里使用。

至此，缓冲区的设置就全部结束了。

2.1.14　初始化硬盘

进程 0 以及后续创建的进程，要想与硬盘这种块设备进行沟通，就要依托系统为此构建的工作环境。在 2.1.6 节中，系统针对块设备请求项结构进行了设置，这无疑是环境构建的重要部分，但还不是全部，这里系统要对硬盘的操作方式进行进一步设置，并将硬盘中断服务程序与中断描述符表相挂接。这样，工作环境才算全部构建完毕。

要通过调用 hd_init 函数来实现此目的，执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
    ...
    hd_init();
    ...
}
```

图 2-15 形象地给出了初始化过程：

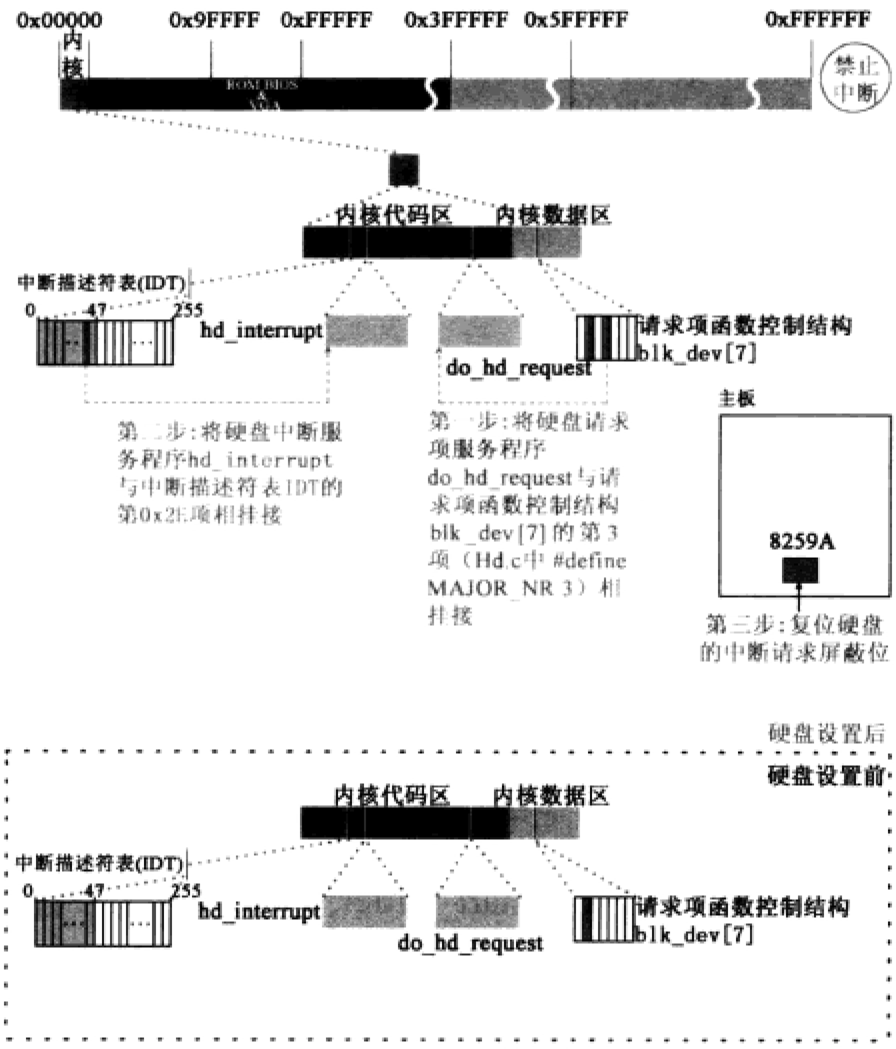


图 2-15　初始化硬盘

进入 hd_init 函数后，将硬盘请求项服务程序 do_hd_request 与 blk_dev 控制结构相挂接；这意味着以后硬盘与请求项的交互工作将由 do_hd_request 函数来处理，如图 2-15 中第一步所示，执行代码如下：

```
//代码路径：kernel/blk_dev/hd.c:
void hd_init(void)
{
    blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
    ...
}
```

然后将硬盘中断服务程序 hd_interrupt 与中断描述符表相挂接，从而进一步完善中断服务体系，如图 2-15 中第二步所示，执行代码如下：

```
//代码路径：kernel/blk_dev/hd.c:
void hd_init(void)
{
    ...
    set_intr_gate(0x2E,&hd_interrupt);
    ...
}
```

最后，复位主 8259A int2 的屏蔽位，允许从片发出中断请求信号；复位硬盘的中断请求屏蔽位（在从片上），允许硬盘控制器发送中断请求信号。执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
void hd_init(void)
{
     ...
     outb_p(inb_p(0x21)&0xfb,0x21);
    outb(inb_p(0xA1)&0xbf,0xA1);
}
```

2.1.15　初始化软盘

进程 0 以及后续创建的进程，要想与软盘这种块设备进行沟通，也要依托系统为此构建的工作环境。本章 2.1.6 节中，系统针对块设备请求项结构进行了设置，这无疑是环境构建的重要部分，但还不是全部，这里系统要对软盘的操作方式进行进一步设置，并将软盘中断服务程序与中断描述符表相挂接。这样，与软盘进行沟通的工作环境才算全部构建完毕。

要通过调用 floppy_init 函数来实现此目的，执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
     ...
     floppy_init ();
     ...
}
```

图 2-16 中给出了软盘初始化的主要步骤和完成后的效果。

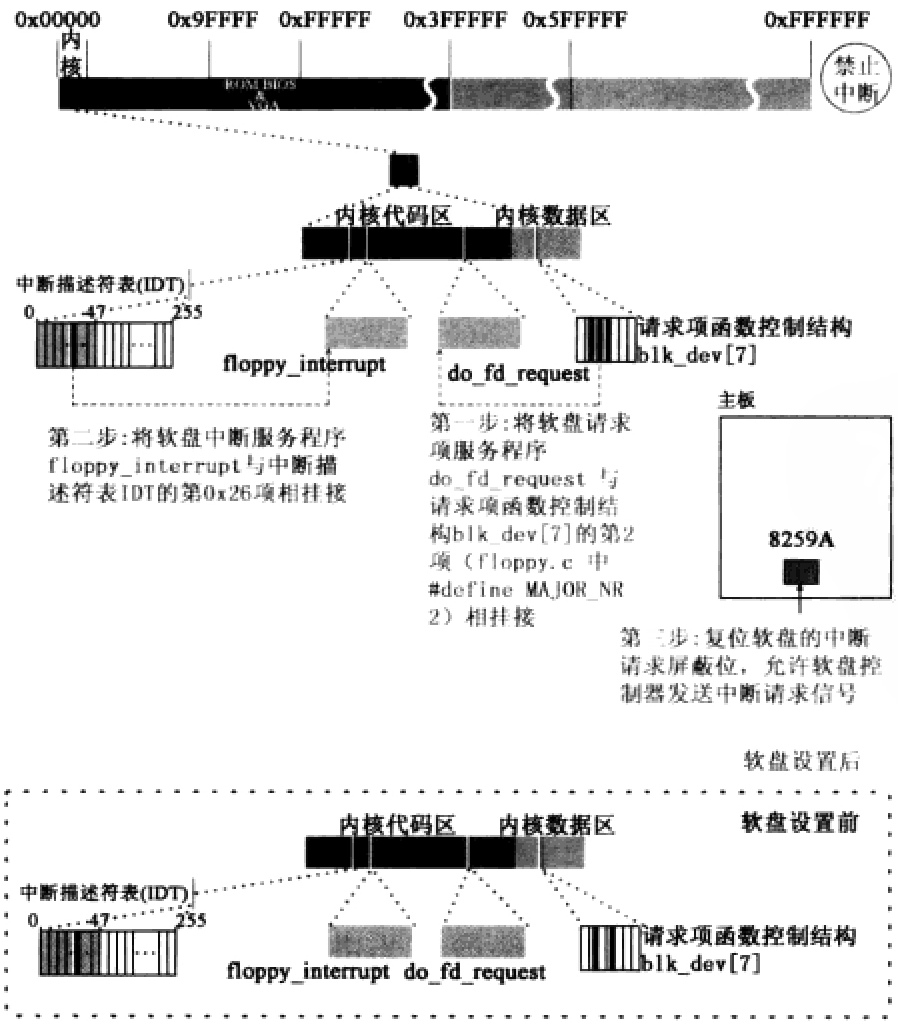


图 2-16　初始化软盘

进入 floppy_init 函数后，将软盘请求项服务程序 do_fd_request 与 blk_dev 控制结构相挂接，这意味着以后软盘与请求项的交互工作将由 do_fd_request 函数来处理，如图 2-16 中第一步所示，执行代码如下：

```
//代码路径：kernel/blk_dev/floopy.c:
void floppy_init (void)
{
    blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
    ...
}
```

然后将软盘中断服务程序 floppy_interrupt 与中断描述符表相挂接，这是构建 32 位保护模式中断服务体系的最后一步，这一步的完成标志着 Linux 0.11 保护模式下的中断服务体系彻底构建完毕了，如图 2-16 中第二步所示，执行代码如下：

```
//代码路径：kernel/blk_dev/floopy.c:
void floppy_init (void)
{
    ...
    set_trap_gate(0x26,&floppy_interrupt);
    ...
}
```

最后，复位软盘的中断请求屏蔽位，允许软盘控制器发送中断请求信号。

执行代码如下：

```
//代码路径：kernel/blk_dev/floopy.c:
void floppy_init (void)
{
    ...
outb(inb_p(0x21)&~0x40,0x21);
}
```

2.1.16 开中断

现在，系统中所有中断服务程序都已经和中断描述符表正常挂接，这意味着中断服务体系已经构建完毕，系统可以在 32 位保护模式下处理中断信号了，所以要将中断开启。执行代码如下：

```
//代码路径：init/main.c:
void main(void)
{
    ...
    sti();
    ...
}
```

图 2-17 中给出了开中断后的效果，注意其中 EFLAGS 寄存器中的变化。

从第 1 章的第 1.3 节关闭中断开始，系统就在为进程 0 能够在主机中正常运算做准备，同时还在为进程 0 及其后续创建的进程与外设沟通构建环境，到这里为止，这些工作就全部做完了，

接下来，进程 0 将在此环境下正式开始工作。

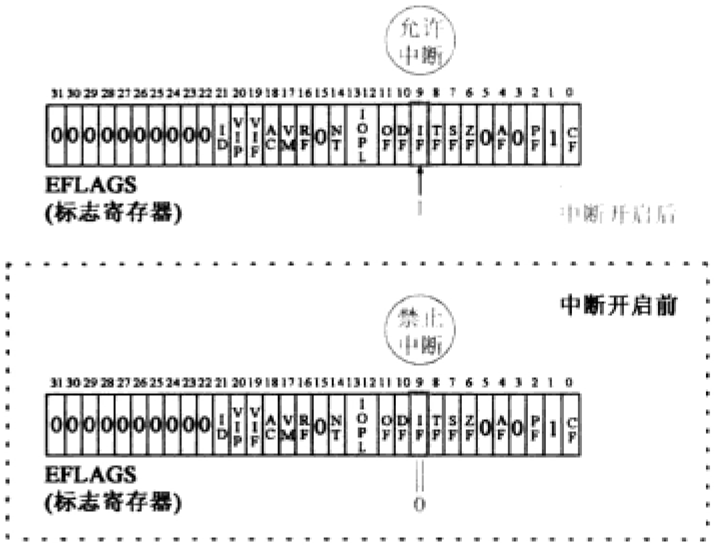


图 2-17 开启中断

2.2 进程创建的最基本动作

2.2 节～2.4 节将详细介绍第二阶段的内容，即与进程 1 有关的工作。本节先来介绍进程 0 是如何创建进程 1 并切换到进程 1 的。在 2.1 节中，系统为进程 0 能够在主机中正常运算做了充分的准备，同时还为进程 0 及其后续创建的进程与外设沟通构建了基本环境。在这些工作的基础上，由进程 0 来创建进程 1，则进程 1 也会像进程 0 一样，具备在主机中正常执行的能力，并享有系统在此之前构建的与外设沟通的环境。

2.2.1 操作系统为进程 0 创建进程 1 做准备

在 Linux 0.11 中，除进程 0 外，所有进程都是由一个已有进程在用户态下完成创建的。为了遵守这个规则，在进程 0 正式创建进程 1 之前，要将进程 0 由内核态转变为用户态，方法是调用 move_to_user_mode 函数，模仿中断返回动作，实现进程 0 的特权级从内核态转变为用户态。执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
    ...
    move_to_user_mode();
    ...
}

// 代码路径：include/system.h:
#define move_to_user_mode() \
__asm__ ("movl %%esp,%%eax\n\t" \
    "pushl $0x17\n\t" \
    "pushl %%eax\n\t" \
    "pushfl\n\t" \
```

```
    "pushl $0x0f\n\t" \
    "pushl $1f\n\t" \
    "iret\n" \
    "1:\tmovl $0x17,%%eax\n\t" \
    "movw %%ax,%%ds\n\t" \
    "movw %%ax,%%es\n\t" \
    "movw %%ax,%%fs\n\t" \
    "movw %%ax,%%gs" \
    :::"ax")
```

当执行 iret 时，硬件会将从上至下的 5 个 push 压栈的数据分别出栈，给 ss、esp、eflags、cs、eip。压栈顺序与通常中断时硬件的压栈动作一样，只是 iret 前面并没有一个真正的中断引起的压栈动作，所以说是模仿，等到执行 iret 时，返回的效果却是一样的。

下面我们对用户态、内核态以及这两种状态切换的方式做个介绍。Intel 在 80386 以后的 CPU 中为了保护代码和数据设计了多种特权级，目的是为了防止低特权级的代码直接跳转执行高特权级代码或访问高特权级数据，避免引发灾难性后果。但操作系统的实际运行中有大量低特权级的用户进程需要使用高特权级的内核的程序，如用户进程使用系统调用。为此 Intel 设计了几种特权级转换的方法，在此处，Linux 0.11 采用的是中断和中断返回的方法来转换特权级。段描述符中有两位是用来表示当前所执行代码的特权级的，可以表示 0、1、2、3 共四种特权级，在 Linux 0.11 中，采用其中的两种特权级，分别是用户态和内核态，用户态时特权级值为 3，内核态时特权级为 0。

Intel 将 CPU 设计为：当执行中断和中断返回（iret）时，硬件改变这个特权级的值。注意，“pushl $0x17\n\t” \ 代表的是 ss 段选择符，0x17 中的 7 用二进制表示就是 00010111，最后两位是 3，所以是用户特权级，倒数第 3 位是 1，表示从 LDT 中获取描述符，因此第 4 ～ 5 位的 10 表示从 LDT 的第 3 项中得到有关用户栈段的描述符。同理，"pushl $0x0f\n\t" \ 代表的是 cs 段选择符，依此类推 0x0f 中的 f 就是 1111，也是用户特权级，从 LDT 的第 2 项中取得用户态代码段描述符。当 iret 执行时，硬件把压栈的这些值恢复给 ss、esp、eflags、cs、eip，iret 后面执行的程序特权级就转变为用户特权级，也就是程序处在用户态。

在执行完 move_to_user_mode() 后，相当于进行了一次中断返回，这就会导致 CS 特权级的值，从 0 转换成 3，即，从内核态转换成用户态，从现在开始，进程 0 将在用户态进行工作。同时，用户态的进程 0 将从 LDT 中取得代码段描述符，并与弹出的 EIP 组成线性地址。

进程 0 正式开始运行，调用 fork 函数，开始创建进程 1，所有用户进程在创建新进程的时候，都要调用这个函数，该过程如图 2-18 所示：

执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
    ...
    if (!fork())
    ...
}
```

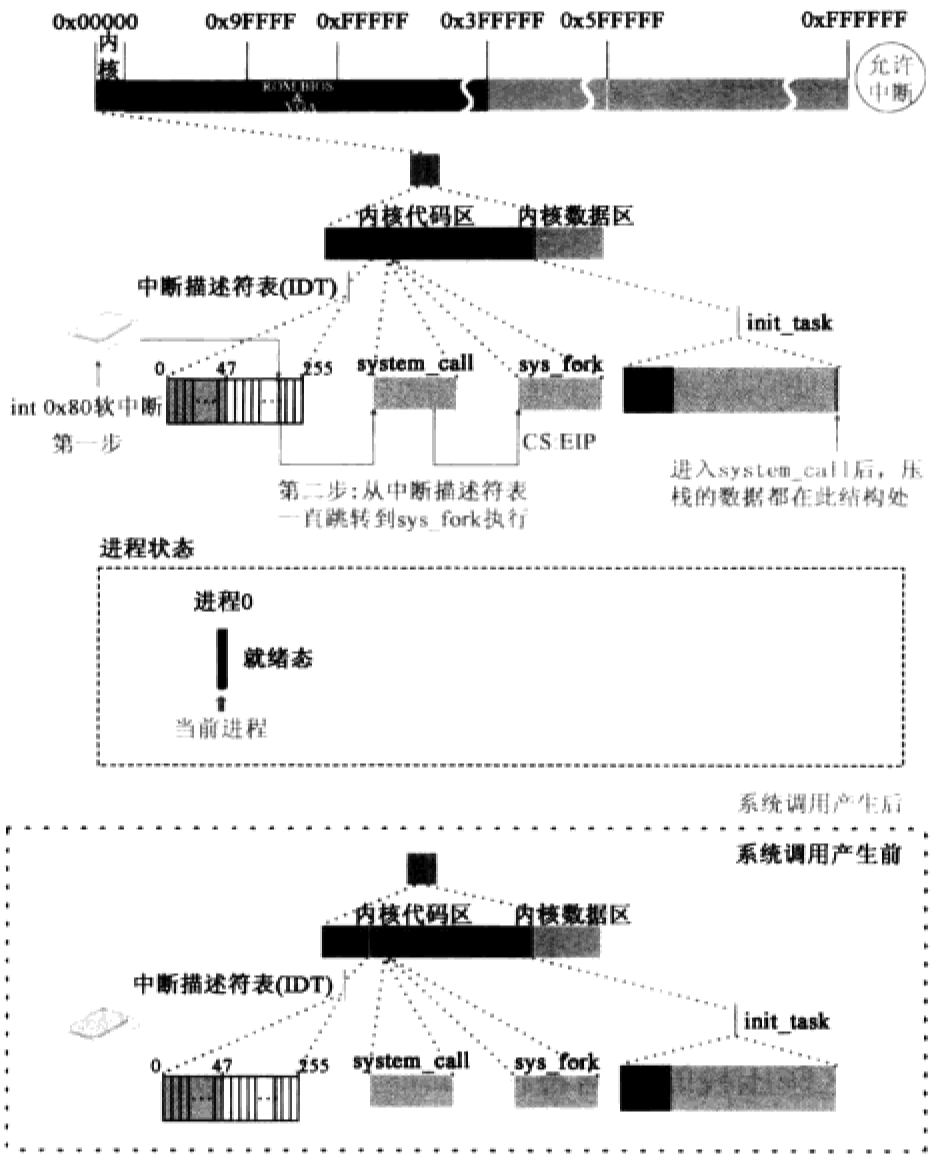


图 2-18　操作系统为创建进程 1 进行的准备工作

我们可以参考 main.c 函数中对 fork 的使用声明，对应代码如下：

```
static inline _syscall0(int,fork)
```

这就意味着，之后将执行到 unistd.h 中的宏函数 syscall0 中去，对应代码如下：

```
// 代码路径：include/unistd.h:
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

展开后，看上去像下面的样子：

```
int fork(void)
{
```

```
long __res;
__asm__ volatile ("int $0x80"
    : "=a" (__res)
    : "0" (__NR_ fork));
if (__res >= 0)
    return (int) __res;
errno = -__res;
return -1;
}
```

执行步骤是这样的，先执行 : "0" (__NR_##name));，这一行的意思是将 fork 对应的函数编号赋值给 eax。由于在 unistd.h 中对 fork 这个标号是这样定义的 :

```
#define __NR_fork  2
```

所以 __NR_##name 此时对应的应该是 __NR_fork，这说明给 eax 中赋的值为 2，这个编号将作为 fork 函数，即 sys_fork 函数在系统调用函数表 sys_call_table 中的偏移值。

紧接着就执行 "int $0x80" ，这一行的意思是产生一个软中断，这样进程 0 的特权级就转变到内核态了，CS 中的特权值由 3 转换为 0。

进入内核态后，硬件会将 ss、esp、eflags、cs、eip 这五个寄存器的数值压入图 2-18 中右上部分所示的 init_task 中的进程 0 内核栈，注意其中 init_task 结构后面的线条，它表示了刚刚压入内核栈的寄存器数值。前面刚刚提到的 move_to_user_mode 这个函数中进行的压栈动作就是完全模仿中断的硬件压栈，这些压栈的数据将在 2.2.7 节中用来初始化进程 1“任务状态描述符表 TSS”中的数据。

值得注意的是，EIP 这时指向的是当前指令 "int $0x80" 的下一行，即 if (__res >= 0) 这一行，读者会在 2.3.1 节中看到，那时它将成为进程 1 执行的第一条指令的位置，也将成为进程 0 从 fork 这个系统调用中断返回后的第一条指令的位置。

压栈完成后，跳转到 _system_call: 处执行，继续将 ds、es、fs、edx、ecx、ebx 压栈，以上一系列的压栈操作都是为了初始化进程 1 中的 TSS 结构做准备的。最终，内核通过 2.2.1 节中所设置的 eax 的偏移值“2”去查询“系统调用函数表（sys_call_table）”，从中得知本次系统调用对应的函数是 sys_fork，这样，就跳转到 _sys_fork: 处执行，准备工作告一段落。上述过程的执行代码如下 :

```
//代码路径：kernel/system_call.s:
...
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx
    pushl %ebx
    movl $0x10,%edx
```

```
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx
    mov %dx,%fs
    call _sys_call_table(,%eax,4)
 ...
 _sys_fork:
 ...
```

代码中 call _sys_call_table(,%eax,4) 这一行就是查询“系统调用函数表（sys_call_table）”，这个表是在 sys.h 中定义的，对应代码是：

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open,...}
```

由于 eax 数值为 2，所以偏移量就是 2，于是就对应到 sys_fork 函数去执行了。

代码中继 ss、esp、eflags、cs、eip 这五个寄存器数据压栈后，继续将一些寄存器的数值压栈，对应代码如下所示：

```
// 代码路径：kernel/system_call.s:
...
_system_call:
......
push %ds
push %es
push %fs
pushl %edx
pushl %ecx
pushl %ebx
......
```

这些压栈的数据也将用来初始化进程 1“任务状态描述符表 TSS”中的数据。

2.2.2　在进程槽中为进程 1 申请一个空闲位置并获取进程号

2.1.7 节中已经介绍过，在激活进程 0 的过程中，已经对其所在的进程槽 task[64] 进行了清空设置，创建出了空闲项，这些空闲项在这里就要发挥作用了。

先要调用 find_empty_process 函数，为这个进程 1 获得一个可用的进程号和一个空闲的任务号。图 2-19 左上部分形象地表示了这个调用。

执行代码如下：

```
// 代码路径：kernel/system_call.s:
...
   _sys_fork:
call _find_empty_process
...
```

具体执行步骤是这样的：内核中使用全局变量 last_pid 来存放系统自开机以来累计的进程数，也将此变量用作新建进程的进程号。在内核的数据区中有一个 task[NR_TASKS]，也就是进程槽，此数组有 64 项，用来存放进程的 task_struct 指针。内核第一次遍历 task 数组，判断获得

的进程号是否可以用于新的进程。如果可以，则第二次遍历 task 数组以获得一个空的 task_struct 指针，在此数组中对应的索引值将作为新进程在该数组中的“项号”。现在的进程号 last_pid 就是 1，在进程槽 task[64] 中占据第二项，数值为“1”，图 2-19 的右半部分标识了这个结果。如果 find_empty_process 函数返回负数，意味着当前已经有 64 个进程在运行。task 这个数组只有 64 项，因为 Linux 0.11 最多只能创建 64 个进程，当然这种情况现在还不会发生。执行代码如下：

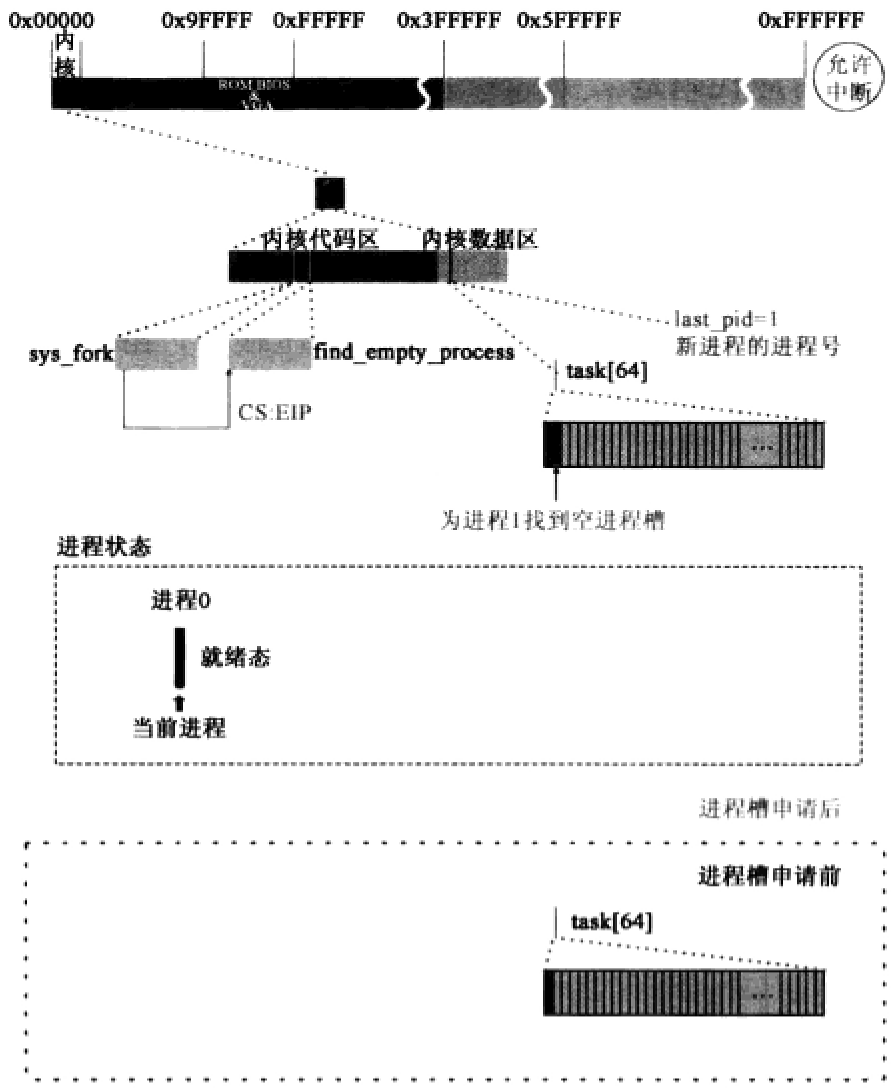


图 2-19　在内核数据区中查找进程空闲项

```
// 代码路径：kernel/fork.c:
 int find_empty_process(void)
{
    int i;

    repeat:
            if ((++last_pid)<0) last_pid=1;
            for(i=0 ; i<NR_TASKS ; i++)
                if (task[i] && task[i]->pid == last_pid) goto repeat;
    for(i=1 ; i<NR_TASKS ; i++)
            if (!task[i])
                return i;
    return -EAGAIN;
}
```

可见，在本章 2.1.7 节中，将进程槽中除了进程 0 所占用表项之外的其余表项都清空，为现

在从进程槽中方便地找到空闲项创造了条件。

2.2.3　复制进程信息之前，先将一些数据压栈

进程 1 在进程槽中的位置和新进程号确定后，即将创建的新进程 1 就等于有了身份。接下来，系统要将进程 0 管理结构拷贝给进程 1 的管理结构，这就是进程创建最关键的一步。但在此之前，我们先来关注一下系统为这次拷贝做的准备工作，执行代码如下：

```
// 代码路径：kernel/system_call.s:
_sys_fork:
...
testl %eax,%eax
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
...
```

这里继续将一些寄存器的数值压栈，这些压栈的数据也将用来初始化进程 1“任务状态描述符表 TSS”中的数据，如图 2-20 中的第一步所示。值得注意的是：pushl %eax 这一行中的 eax 就是 2.2.2 节中调用 find_empty_process 函数后的返回值，这个值就是进程 1 在 task[64] 数组中申请的项号，数值为“1”。

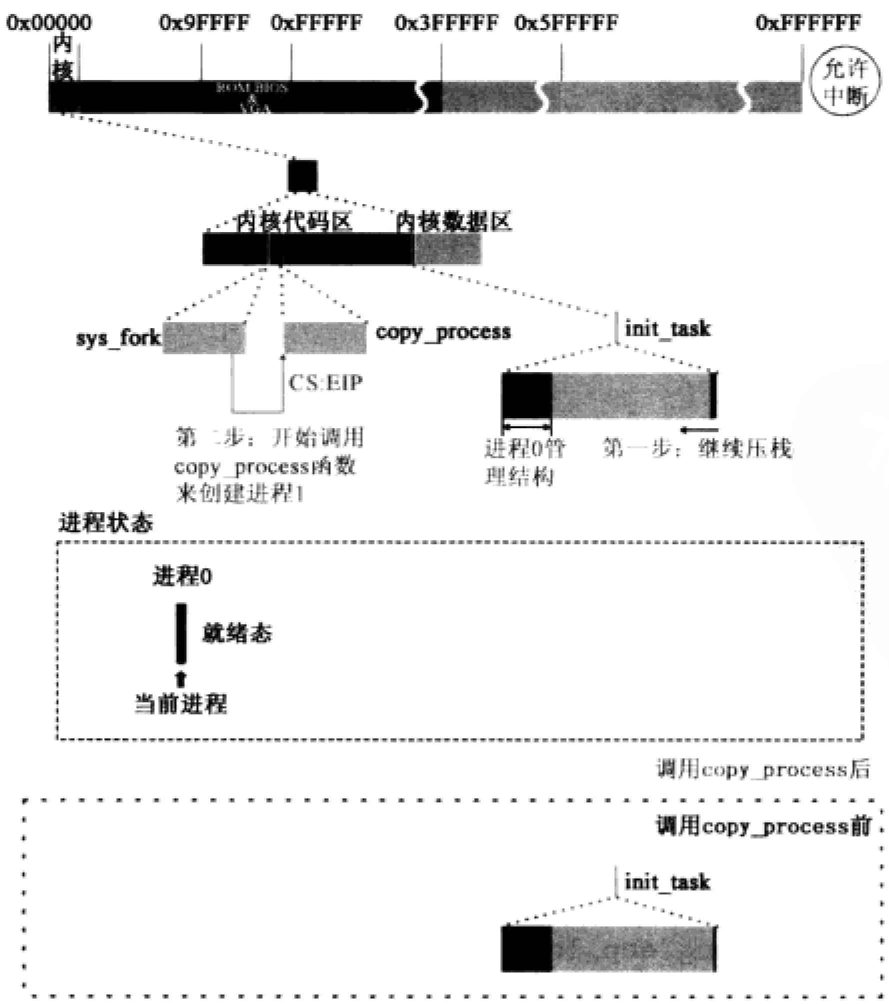


图 2-20　调用 copy_process 之前的压栈动作

压栈结束后，才开始调用进程复制函数 copy_process，如图 2-20 中第二步所示。

2.2.4 初步设置进程 1 管理结构

在 2.1 节的一系列工作中，进程 0 已经成为了一个可以创建其他进程的母本了，这个母本包括“进程 0 的管理结构”和“进程 0 的页表项”等，接下来这个母本就要发挥作用，即将它的内容拷贝给进程 1，这里具体表现为：将进程 0 管理信息的内容拷贝给进程 1，这些内容将成为进程 1 管理结构的雏形。具体执行情况如下。

进入 copy_process 函数后，执行如下代码：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
  long ebx,long ecx,long edx,
  long fs,long es,long ds,
  long eip,long cs,long eflags,long esp,long ss)
{
    ...
    p = (struct task_struct *) get_free_page();
    ...
    task[nr] = p;
    *p = *current;
    ...
}
```

调用 get_free_page 函数，在主内存末端申请一个空闲页面，如图 2-21 中的“进程 1 管理结构所在页面”所示。p = (struct task_struct *) get_free_page() 这行代码将空闲页面的首地址认定为进程管理结构 task_struct 的首地址，并载入进程槽 task[64] 的 nr 项中，这里 nr 就是 2.2.3 节中 pushl %eax 压栈的值，所以它的值是 1，即第二项，因为 nr 是 1，task[nr] 是 task[64] 的第二项，也就是进程 1 对应的项。所以 p 就指向进程 1 的 task_struct，如图 2-21 中第一步所示。对应代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
  long ebx,long ecx,long edx,
  long fs,long es,long ds,
  long eip,long cs,long eflags,long esp,long ss)
{
    ...
    p = (struct task_struct *) get_free_page();
    ...
    task[nr] = p;
    ...
}
```

现在是进程 0 在创建进程 1，当前进程肯定就是进程 0，所以当前进程指针 current 指向进程 0。对应代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
  long ebx,long ecx,long edx,
```

```
    long fs,long es,long ds,
    long eip,long cs,long eflags,long esp,long ss)
{
    ...
    *p = *current;
    ...
}
```

把进程 0 的管理信息 task_struct 全部拷贝到 p 指向的进程 1 的管理信息 task_struct 中，这意味着，此时进程 1 已经具有了进程 0 的绝大部分能力。该过程如图 2-21 中的第二步所示。

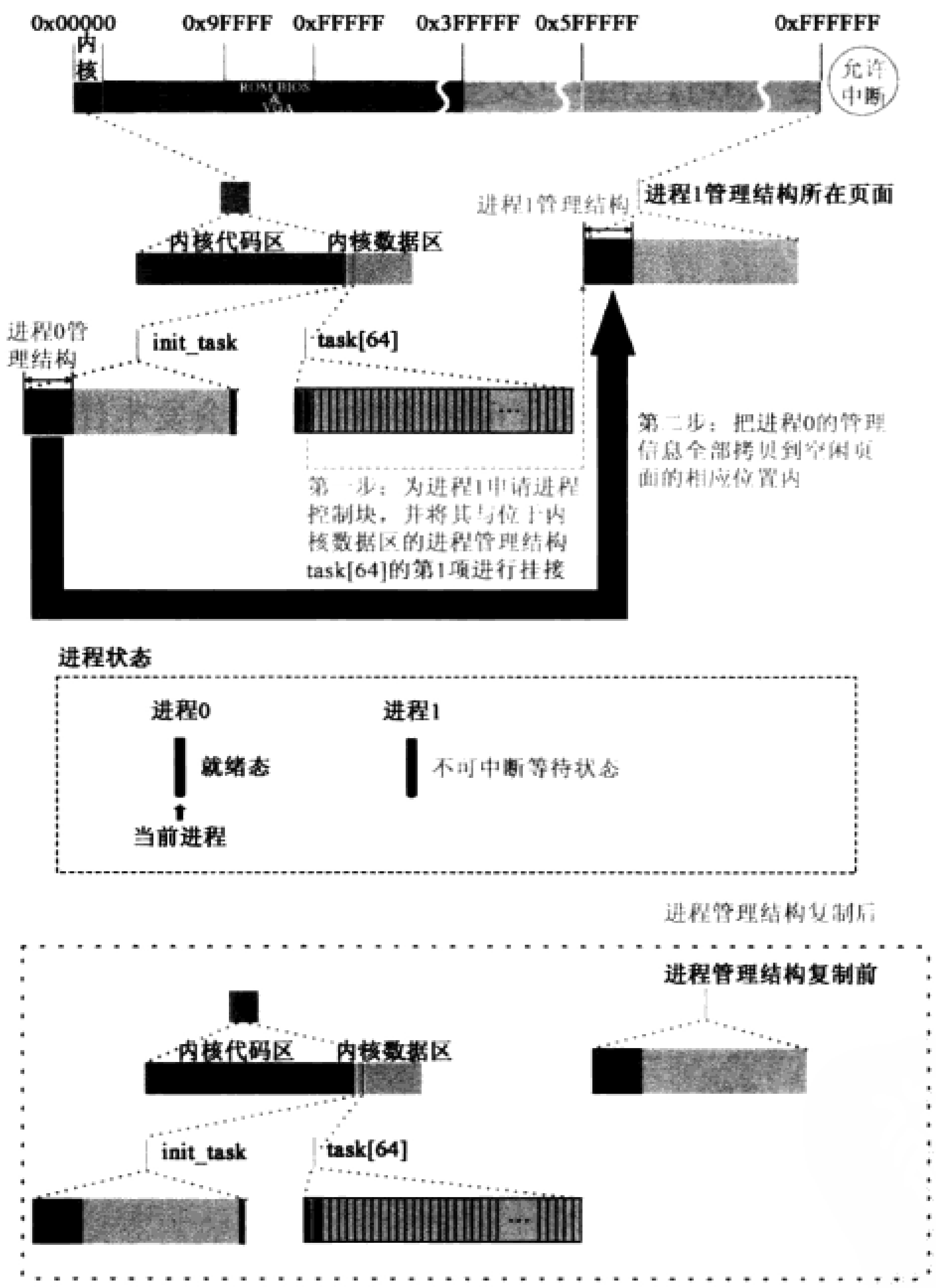


图 2-21　初步设置进程 1 的管理结构

然后，将进程 1 的状态设置为不可中断等待状态，执行代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
            long ebx,long ecx,long edx,
            long fs,long es,long ds,
            long eip,long cs,long eflags,long esp,long ss)
{
    ...
```

```
    p->state = TASK_UNINTERRUPTIBLE;
    ...
}
```

将进程设置为“不可中断等待状态”意味着，只有内核代码中明确表示将该进程设置为就绪状态，它才能被唤醒。除此之外，没有任何办法将其唤醒。之所以对其这样设置，是因为此时进程 1 尚未创建完毕，它的管理结构还需要进一步调整，这就导致进程 1 还不能马上参与多进程的轮询。为此，这里将它设置为“不可中断等待状态”就是要绝对地避免进程 1 受到任何中断类信号的打扰，以免出现混乱。

2.2.5 进程 0 创建进程 1 的过程中发生时钟中断

2.1.7 节中对时钟中断的设置已经为轮询奠定了基础，现在我们来关注一下此时发生时钟中断是否会导致轮询。

我们现在暂时先把进程 1 创建的步骤打断一下，来做一个假设。由于此时中断已经开启，虽然时钟中断的产生是在 2.1.7 节中设定的，但是对进程而言，时钟中断的发生是不可预计的，这也是操作系统的特点之一。假设此时就发生了时钟中断，看看会发生什么。时钟中断的处理过程如图 2-22 所示。

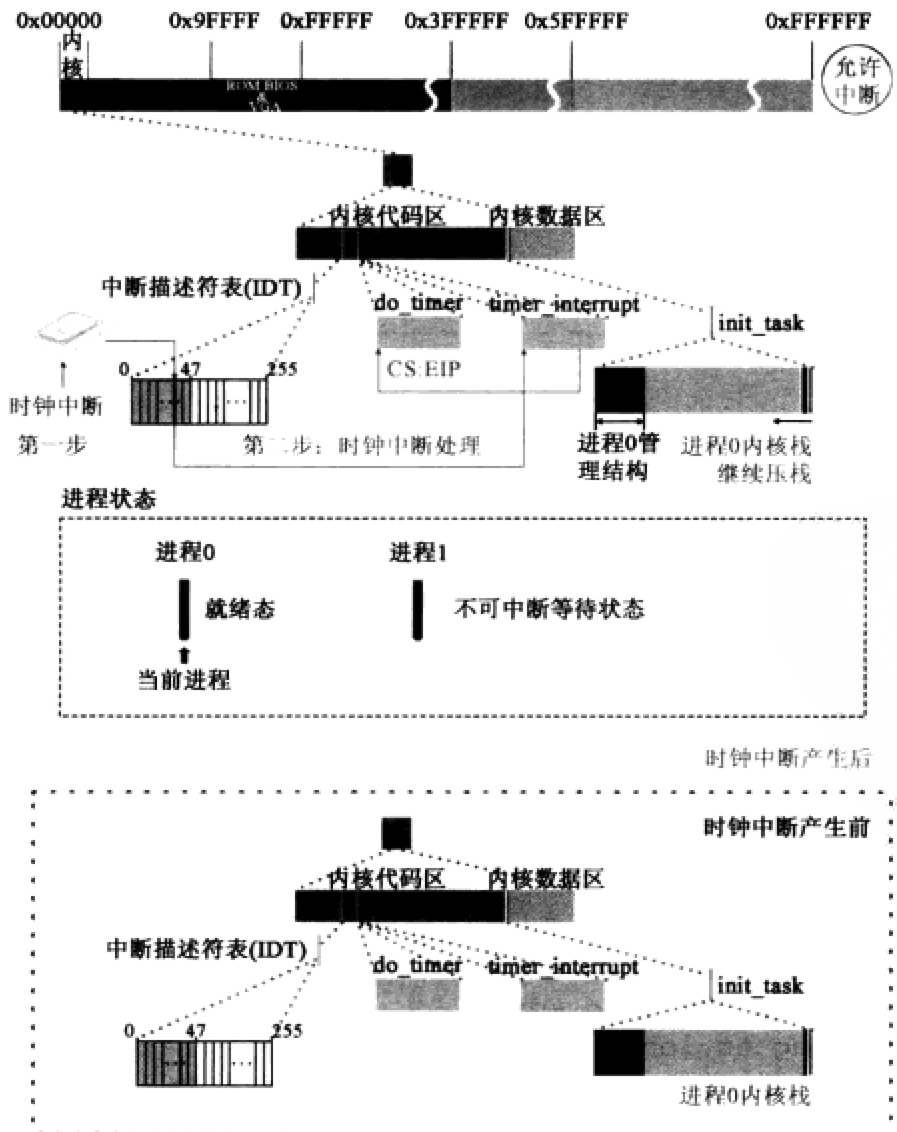


图 2-22 处理时钟中断

图 2-22 中时钟中断产生前，进程 0 内核栈中的数据是响应 fork 系统调用阶段压栈的，它们用来初始化进程 1 的管理结构。时钟中断发生后，其对应的中断服务程序将开始工作，于是跳转到 _timer_interrupt: 执行，系统先将 ds、es、fs、edx、ecx、ebx、eax 压入进程 0 内核栈，以便保护被打断程序执行的“现场”，然后调用 _do_timer 来判断是否需要进程切换。图 2-22 中的第一步和第二步操作描述了这个过程，特别请读者注意图中右侧的进程 0 内核栈中的数据，与时钟中断产生前相比，又有一部分数据压入了进程 0 内核栈。执行代码如下：

```
// 代码路径：kernel/system_call.s:
...
_timer_interrupt:
    push %ds                # save ds,es and put kernel data space
    push %es                # into them. %fs is used by _system_call
    push %fs
    pushl %edx              # we save %eax,%ecx,%edx as gcc doesn't
    pushl %ecx              # save those across function calls. %ebx
    pushl %ebx              # is saved as we use that in ret_sys_call
    pushl %eax
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    movl $0x17,%eax
    mov %ax,%fs
    incl _jiffies
    movb $0x20,%al          # EOI to interrupt controller #1
    outb %al,$0x20
    movl CS(%esp),%eax
    andl $3,%eax            # %eax is CPL (0 or 3, 0=supervisor)
    pushl %eax
    call _do_timer          # 'do_timer(long CPL)' does everything from
...
```

进入 do_timer 函数后，要对进程 0 的时间片剩余与否进行判断，执行代码如下：

```
// 代码路径：kernel/sched.c:
void do_timer(long cpl)
{
    ...
   if ((--current->counter)>0) return;
    ...
}
```

进程 0 的时间片是进程 0 的管理结构中的一个成员，进程 0 的管理结构已经在代码设计阶段事先设计好了，sched.h 中定义了这个管理结构，对应代码如下：

```
// 代码路径：include/sched.h:
#define INIT_TASK \
/* state etc */    { 0,15,15, \
...
}
```

代码中的两个 15 分别代表进程 0 的时间片和优先级，可见进程 0 的时间片被初始化成 15。

因此，--current->counter 将会把进程 0 的时间片削减“1”，我们不妨设时间片从 10 削减为 9，

这表示并没有削减完，所以不发生进程切换。于是 do_timer 函数退出，并跳转到 ret_from_sys_call: 处，执行代码如下：

```
// 代码路径：kernel/system_call.s:
_timer_interrupt:
...
jmp ret_from_sys_call
...
```

2.2.6 从时钟中断返回

时钟中断处理程序继续执行，由于在 2.2.5 节中它已经跳转到 ret_from_sys_call，所以从 ret_from_sys_call：开始执行，准备处理当前进程接收到的信号（关于信号的内容，我们将在第 7 章中详细说明）。

然而，因为现在当前进程是进程 0，所以不对它的信号做任何处理（这种情况仅限于进程 0，其余进程都要处理信号），直接就跳转到标号 3 处，将刚才压栈的寄存器数值还原，并返回，到这里，此次时钟中断结束。如图 2-23 所示，请读者注意与图 2-22 比较进程 0 内核栈的变化。

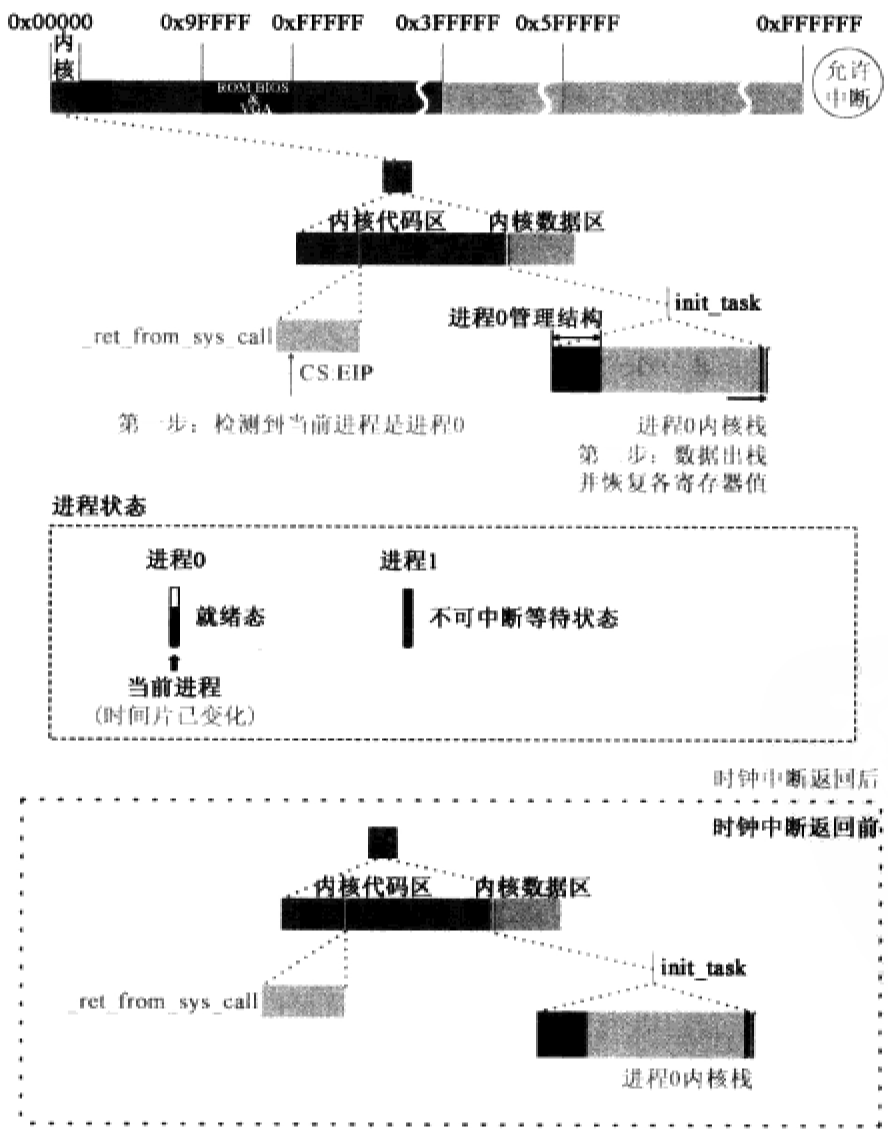


图 2-23 时钟中断返回

执行代码如下：

```
// 代码路径：kernel/system_call.s:
```

```
...
  ret_from_sys_call:
     movl _current,%eax
     cmpl _task,%eax
     je 3f
    ...
     3:   popl %eax
          popl %ebx
          popl %ecx
          popl %edx
          pop %fs
          pop %es
          pop %ds
          iret
```

正如 2.2.5 节中所述，由于现在进程 0 的时间片并没有削减完，这一个条件就决定，当前并不发生进程切换。于是进程 0 将继续执行，它将继续完成创建进程 1 的工作。

2.2.7 调整进程 1 管理结构

时钟中断是在 2.2.4 节中的 p->state = TASK_UNINTERRUPTIBLE; 这一行代码执行完后发生的，所以现在接着这行代码后面执行，进程 0 继续创建进程 1。因为进程 0 已经完成了拷贝任务，所以进程 1 管理结构的雏形此时已经形成了，进程 0 管理结构中的数据信息并不一定全都适用于进程 1，因此还需要针对具体情况进行调整，执行代码如下，对一些重要的数据项的调整，我们直接在代码中进行了注释，以方便读者理解。

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
          long ebx,long ecx,long edx,
          long fs,long es,long ds,
          long eip,long cs,long eflags,long esp,long ss)
{
    ...
   p->pid = last_pid;             // 将本章 2.2.2 节中得到的 last_pid，设为进程 1 的进程号，
                                  // 数值为“1”

   p->father = current->pid;      // 进程 0 就是进程 1 的父进程，所以就将进程 0 的进程号“0”作
                                  // 为进程 1 的父进程号

   p->counter = p->priority;      // 本章 2.2.5 节中已经介绍过，进程 0 的优先级是 15，所以，用优先
                                  // 级的数值来初始化进程 1 的时间片，则进程 1 的时间片也被初始化为 15

   p->signal = 0;                 //2.2.6 节中所介绍的进程的信号，就是记载在 signal 这个成员中的，
                                  // 这个成员总共 4 个字节，折合 32 位，这里相当于把 32 位全部清 0 了

   p->alarm = 0;                  // 用户可以人为设定进程执行的时间，这个 alarm 将用以判断当前
                                  // 进程执行是否超过了这个人为设定的时间

   p->leader = 0;
   p->utime = p->stime = 0;
   p->cutime = p->cstime = 0;
   p->start_time = jiffies;
```

```
// 从这里开始，都是对进程 1 任务状态描述符表 TSS 中的各个成员进行初始化，前面在本章 2.2.1 节和 2.2.3 节
// 中压栈的寄存器值，这时候就都发挥作用了，将来一旦切换到进程 1，系统就用这些值来初始化各个寄存器的
// 值，以此来为进程 1 的执行提供数据依据
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0;         // 这里将 eax 初始化成 0，意义重大，这一点将在本章 2.3.1 节中得以体现
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;
    p->tss.gs = gs & 0xffff;
    p->tss.ldt = _LDT(nr);
    p->tss.trace_bitmap = 0x80000000;
     ...
}
```

调整完成后的进程 1 管理结构如图 2-24 所示。

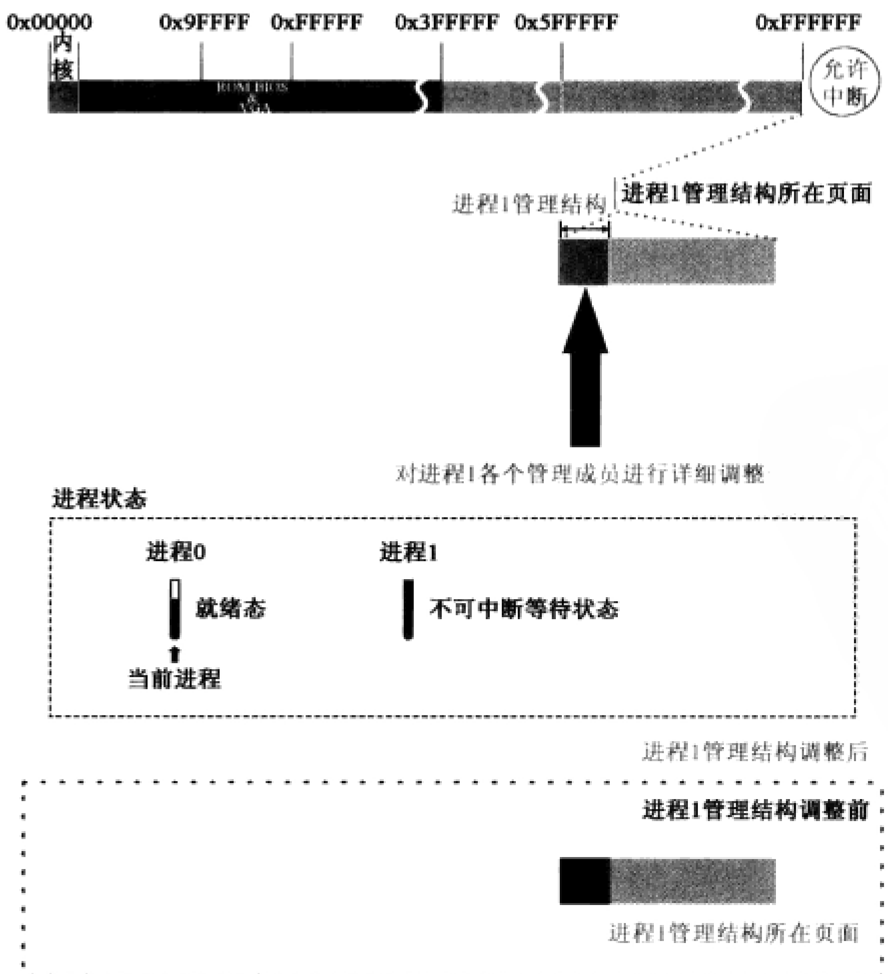


图 2-24　对进程 1 的管理结构进行微调

2.2.8　设置进程 1 的线性地址空间及物理页面

继续创建进程 1，这里将设置进程 1 的页目录表以及复制页表，一般来讲，每个进程将来都有可能加载属于自己的程序代码，这些程序代码是要占用主内存空间的，所以，这些空间要由进程管理。

在 Linux 0.11 中，每个进程所属的程序代码执行时，都要根据其线性地址来进行寻址，并最终映射到物理内存上。计算机会通过硬件将这个线性地址解析成“页目录项”、“页表项”和“页面内偏移”这三级管理数据，最终根据这些数据在物理内存上找到指定的地址。通过图 2-25 我们可以看出，线性地址有 32 位，分别用以表示对应的页目录项、页表项和页内偏移，因为 Linux 0.11 中仅有一个页目录表，所以通过线性地址中提供的“页目录项”数据就可以找到页目录表中对应的页目录项；通过这个页目录项就可以找到唯一的页表；之后，再通过线性地址中提供的“页表项”数据，就可以在该页表中找到唯一的页表项；通过此页表项可以进一步找到唯一的物理页面；最后，通过线性地址中提供的“页内偏移”数据，确定页内偏移值，这个值就是实际的物理地址值。

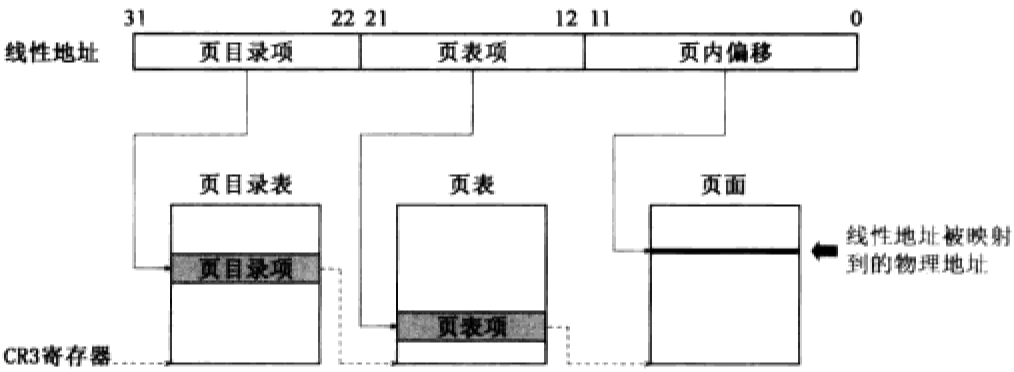


图 2-25　线性地址到物理地址的映射过程示意图

页目录项存在于页目录表中，用以管理页表；页表项存在于页表中，用以管理页面。因此，这里就要设置进程 1 的页目录项，以及为进程 1 创建页表并将进程 0 的页表项复制到进程 1 的页表中，以此为进程 1 将来执行代码创造条件。进程 1 此时还没有对应的程序，它的页表又是从进程 0 对应页表拷贝过来的，所以它们管理的页面完全一致，这就导致它暂时和进程 0 共享一套内存页面管理结构，如图 2-26 所示。

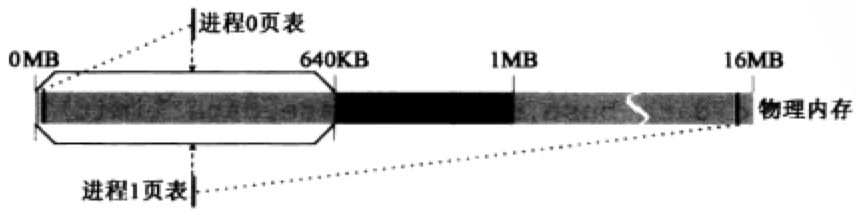


图 2-26　进程 0 和进程 1 共享页表示意图

等将来它有了自己的程序，再把关系解除，并重新组织自己的内存管理结构。

创建页目录项和复制页表，要从调用 copy_mem 函数开始，执行代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
            long ebx,long ecx,long edx,
            long fs,long es,long ds,
            long eip,long cs,long eflags,long esp,long ss)
{
    ...
    if (copy_mem(nr,p))
    ...
}
```

进入 copy_mem 函数后，开始为设置页目录表和复制页表做准备。提取当前进程（进程 0）的代码段、数据段，以及段限长的信息，并设置了进程 1 的代码段和数据段的基地址，这个基地址就是它的进程槽号 nr*64MB，设置新进程局部描述符表中段描述符中的基地址，如图 2-27 中的第一步所示。执行代码如下：

```
// 代码路径：kernel/fork.c:
int copy_mem(int nr,struct task_struct * p)
{
    ...
code_limit=get_limit(0x0f);
    data_limit=get_limit(0x17);
    old_code_base = get_base(current->ldt[1]);
    old_data_base = get_base(current->ldt[2]);
    if (old_data_base != old_code_base)
            panic("We don't support separate I&D");
    if (data_limit < code_limit)
            panic("Bad data_limit");
    new_data_base = new_code_base = nr * 0x4000000;
    p->start_code = new_code_base;
    set_base(p->ldt[1],new_code_base);
    set_base(p->ldt[2],new_data_base);
...
}
```

然后调用 copy_page_tables 函数，开始设置页目录表和复制页表，如图 2-27 中第二步和第三步所示，注意其中页目录项的位置。

执行代码如下：

```
// 代码路径：kernel/fork.c:
int copy_mem(int nr,struct task_struct * p)
{
    ...
if (copy_page_tables(old_data_base,new_data_base,data_limit))
...
}
```

进入 copy_page_tables 函数后，先为新的页表申请一个空闲页面，并把进程 0 中第一个页表里面的前 160 个（控制 640KB 内存空间）页表项复制到这个页面中，这也就导致进程 0 和进程 1 的页表暂时都指向了相同的页面，这就意味着进程 1 也具备了操作这些页面的能力，这是进程 1 与进程 0 拥有同样能力的一个重要体现，之后对进程 1 的页目录表进行设置。最后，刷新页变

换高速缓存，这样，进程 1 的内存页面管理结构就建立起来了，执行代码如下，对其中重要的执行步骤，我们在代码中进行了注释：

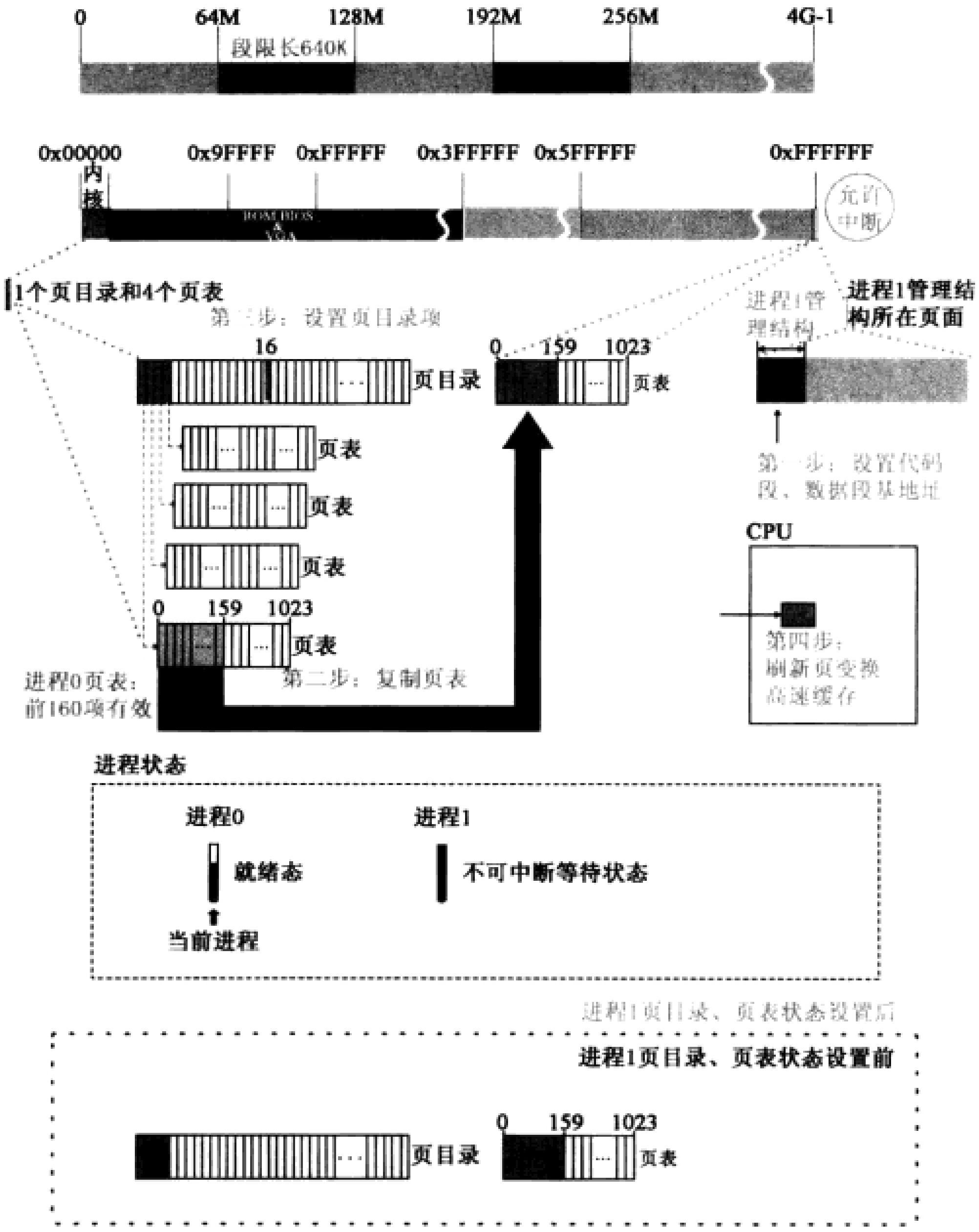


图 2-27　设置进程 1 的线性地址空间

```
// 代码路径：mm/memory.c:
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
    ...
    if ((from&0x3fffff) || (to&0x3fffff))
            panic("copy_page_tables called with wrong alignment");
   from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
   to_dir = (unsigned long *) ((to>>20) & 0xffc);
   size = ((unsigned) (size+0x3fffff)) >> 22;
   for( ; size-->0 ; from_dir++,to_dir++) {
           if (1 & *to_dir)
               panic("copy_page_tables: already exist");
           if (!(1 & *from_dir))
               continue;
           from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
           if (!(to_page_table = (unsigned long *) get_free_page()))
```

```
            return -1; /* Out of memory, see freeing */
        *to_dir = ((unsigned long) to_page_table) | 7;// 这行是对页目录项的设置
        nr = (from==0)?0xA0:1024;       // 这一行就决定了，如果当前进程是进程 0，就把页表中的
                                        // 前 160 个页表项拷贝给进程 1 的页表，如果当前进程
                                        // 不是进程 0，就把页表中全部的页表项，即，1024 项，
                                        // 全部复制给它即将创建的新进程的页表
        for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
            this_page = *from_page_table;
            if (!(1 & this_page))
                continue;
            this_page &= ~2;
            *to_page_table = this_page;
            if (this_page > LOW_MEM) {
                *from_page_table = this_page;
                this_page -= LOW_MEM;
                this_page >>= 12;
                mem_map[this_page]++;
            }
        }
    }
    invalidate();   // 这里刷新"页变换高速缓存"，这个页变换高速缓存在 CPU 中，是用来置换页面的，刷新
                    // 页变换高速缓存，可以提高页面置换的效率，从而提高页面操作的综合效率
    return 0;
}
```

通过 2.2.1 节中的介绍我们得知，进程 1 将来执行时，也和进程 0 一样，都是在用户态通过 LDT 解析线性地址，并最终映射到物理地址的。现在的问题是，进程 0 和进程 1 的线性地址空间分别是 0 ～ 64MB 和 64 ～ 128MB，线性地址值肯定不同，但是进程 1 将来一旦从用户态开始执行，它所对应的代码又和进程 0 的代码一样，同处于内核所在的物理内存空间内。所以，这就需要将进程 0 和进程 1 不同的线性地址映射到相同的物理内存空间内（即 0 ～ 640KB 这个内核空间内），才能保证进程 1 的正常执行。于是这里将进程 0 第一个页面的前 160 项拷贝给进程 1 的页表，并给进程 1 新建一个页目录项，就是为了实现此映射。

从这里我们可以看出，所有进程创建出来的子进程，都需要先从其父进程代码所在页面开始执行，然后才能加载属于它自己的程序（进程加载程序的具体步骤，我们将在本章第 7 节详细介绍），所以，也就都需要先和它的父进程共享相同的物理页面，可见，进程 0 给进程 1 复制页表的这套措施，适用于整个操作系统中全部的进程创建工作。

2.2.9 继续调整进程 1 管理结构

进程 1 的页表和页目录表设置完毕后，将返回到 copy_process 函数中，继 2.2.7 节中对进程 1 管理结构的调整之后，继续对其进行调整。这里先设置一些进程结构中与文件相关的成员，包括：打开了哪些文件 p->filp[20]、进程 0 的"当前工作目录 i 节点结构"、"根目录 i 节点结构"以及"执行文件 i 节点结构"，由于进程 0 中这些数值还都是空的，进程 0 只具备在主机中正常运算的能力，尚不具备与外设以文件形式进行交互的能力，所以，这些设置暂时并不需要，对应

的代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    for (i=0; i<NR_OPEN;i++)
        if (f=p->filp[i])
            f->f_count++;
   if (current->pwd)
       current->pwd->i_count++;
   if (current->root)
       current->root->i_count++;
   if (current->executable)
       current->executable->i_count++;
   ...
}
```

之后把进程 1 的任务状态描述符表和局部描述符表挂接在全局描述符表中，如图 2-28 所示，注意进程 1 在全局描述符表中所占的位置。

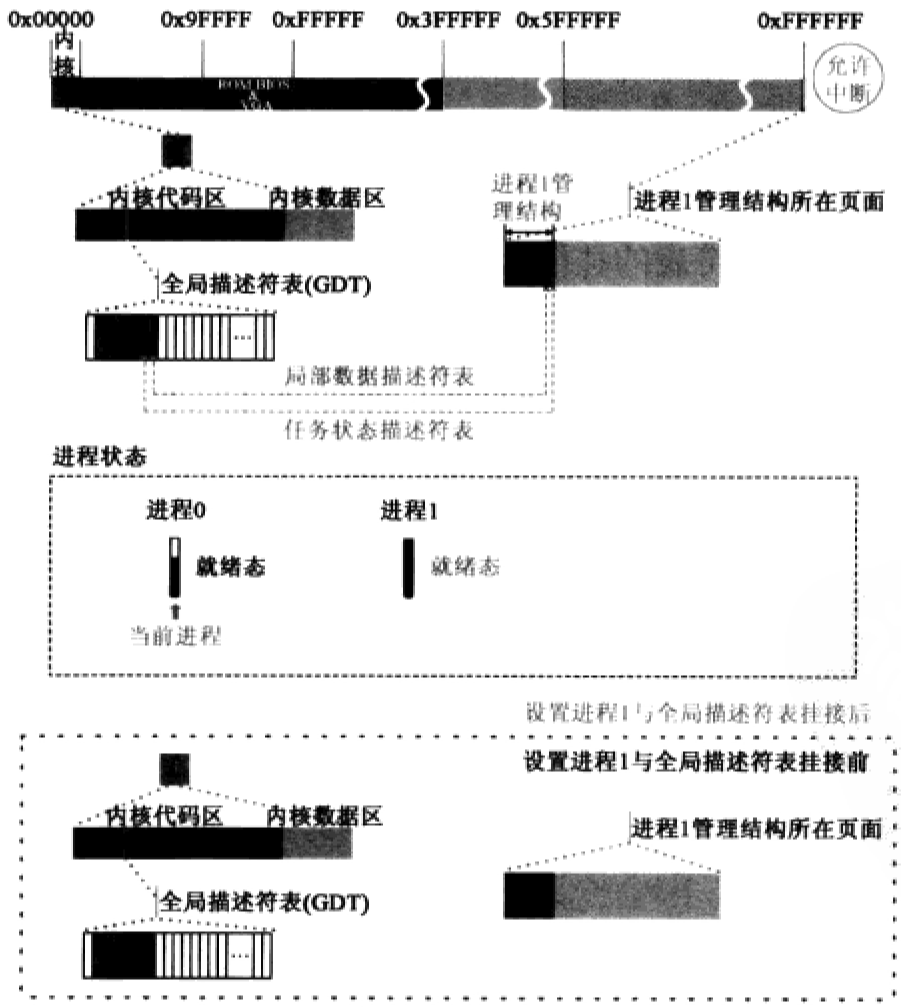


图 2-28　将进程 1 管理结构与全局描述符表挂接

执行代码如下：

```
// 代码路径：kernel/fork.c:
```

```
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
   ...
   set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
   set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
   ...
}
```

任务状态描述符表 TSS 用来记录当前进程执行时所对应的寄存器的数据，这些数据主要在进程切换时发挥作用，比如，现在要由当前进程“进程 A”切换到进程 B，那么系统就要将此时各个寄存器的数值，保存在进程 A 的任务状态描述符表中，以便将来进程 A 再次执行时接着使用而不至于出现混乱；之后，再用进程 B 中 TSS 里面的寄存器值，来设置相应的寄存器，以此支持进程 B 接下来的执行。

局部数据描述符表 LDT 中，记录着当前进程对应程序的代码段和数据段信息，比如代码的基地址等，这些信息将在进程程序执行时提供支持。

系统将来就是通过 GDT 表中挂接的 TSS 描述符和 LDT 描述符，来与当前进程建立关系的，这里将 TSS 和 LDT 挂接在全局描述符表 GDT 中，标志着系统从此具备操作进程 1 的能力。

在 2.1.7 节激活进程 0 过程中对进程槽和全局描述符表的初始化工作在这里仍然发挥着作用，nr 这个数值，就是在进程槽中找到空闲项的标号，这个标号在这里又指引着进程 1 任务状态描述符表和局部描述符表及全局描述符表的挂接。

这样，进程 1 此时已经可以参加进程调度了，所以，将它的状态设置为就绪态，并返回进程号 1。请读者注意图 2-28 中间代表进程的进程条，其中多了一条代表进程 1 处在就绪态的进程条，标志着进程 1 创建已经完成。执行代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    p->state = TASK_RUNNING;
    return last_pid;
}
```

至此，进程 1 的创建工作完毕，这使得进程 1 已经具备了进程 0 的全部能力，它可以在主机中正常的运行了，同时，进程 1 就是进程 0 后续创建的进程，所以也将享有系统目前已经构建出来的与外设沟通的环境。

进程 1 已经能够工作了，所以，接下来进程 0 要切换到进程 1，由进程 1 来完成本章第二阶段的任务。

下面我们开始介绍进程 0 切换到进程 1 的具体步骤。

2.2.10　操作系统如何区分进程 0 和进程 1

进程 1 创建完毕后，copy_process 函数也就执行完毕了，它将返回到 _system_call: 中，执行代码如下：

```
// 代码路径：kernel/system_call.s:
_sys_fork:
...
call _copy_process
    addl $20,%esp
...

_system_call:
...
call _sys_call_table(,%eax,4)
    pushl %eax
    movl _current,%eax
    cmpl $0,state(%eax)             # state
    jne reschedule
    cmpl $0,counter(%eax)           # counter
    je reschedule
...
```

注意看 pushl %eax 这一行代码，这行代码将 2.2.9 节中返回的进程 1 的进程号进行了压栈，之后到 _ret_from_sys_call: 处执行，对应代码如下：

```
// 代码路径：kernel/system_call.s:
ret_from_sys_call:
    movl _current,%eax
    cmpl _task,%eax              // 检测当前进程是不是进程 0
    je 3f
     ...
3:  popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    pop %fs
    pop %es
    pop %ds
    iret
```

由于当前进程是进程 0，所以就跳转到标号 3 处，将压栈的各个寄存器数值还原，图 2-29 中的第一步表示了 init_task 中清栈的这一过程。值得注意的是 popl %eax 这一行代码，这是将本章 2.2.10 节里面刚刚介绍过的 pushl %eax 这一行压栈的进程号，将其回填给 eax，这样此时 eax 的值就是“1”。

之后，中断返回，这行中断返回 iret，意味着，进程 0 从内核态转换成了用户态，这样，本次“创建新进程”这个软中断就结束了。

2.2.1 节中介绍过，在 _syscall0 函数中，if (__res >= 0) 这一行就是进程 0 中断返回后将要执

行的第一行指令，所以接下来进程 0 从这里开始执行，对应代码如下：

```
// 代码路径：include/unistd.h:
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

在执行 if (__res >= 0) 这行代码前，我们先来关注一下："=a" (__res) 这一行代码，这行代码的意思是将 __res 与 eax 寄存器进行绑定，以后只要用到 __res，就用 eax 替换，所以 if (__res >= 0) 这一行代码，实际上就是判断此时 eax 的值是多少，很显然，2.2.10 节中我们刚刚介绍了，这时候 eax 里面装载的是返回的进程 1 的进程号，所以，应该是“1”，于是执行 return (type) __res; 这行代码，将“1”返回。

最后，回到 2.2.1 节中介绍过的 fork 函数的调用点 if (!fork()) 处执行，!1 为“假”，这样就不会执行到 init 函数中，而是进程 0 继续执行，接下来就会执行到 for(;;)。

图 2-29 的第二步形象地表示了上述过程。

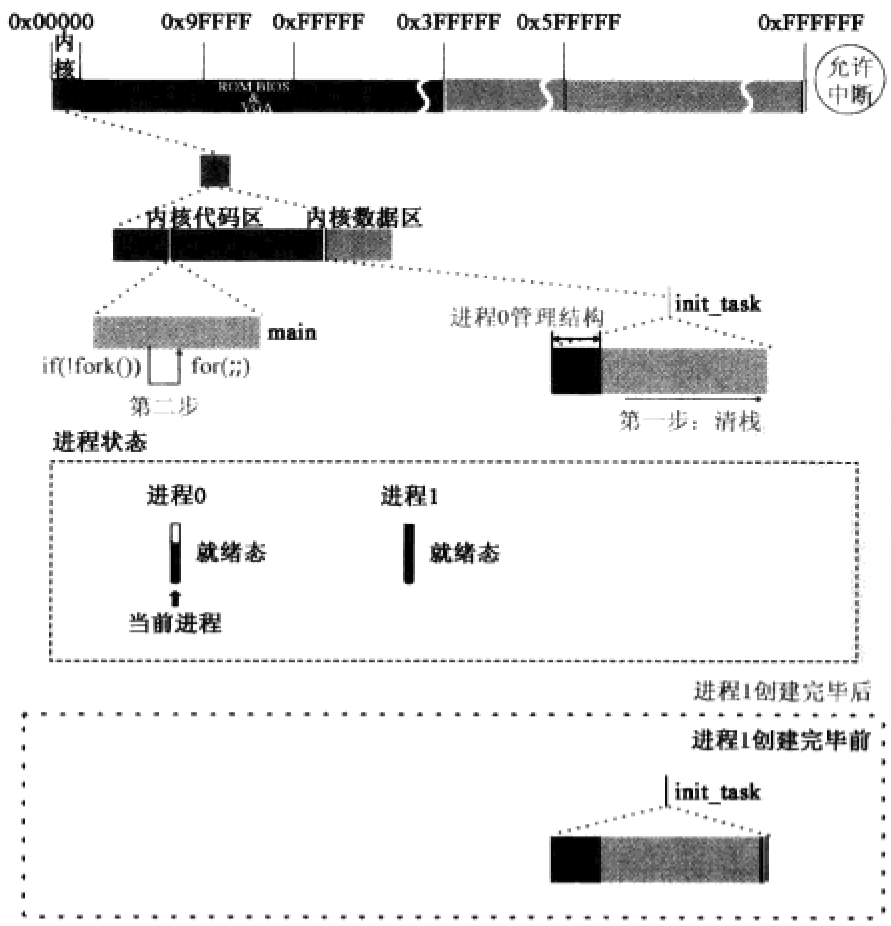


图 2-29 操作系统如何区分进程 0 与进程 1

2.2.11 进程 0 准备切换到进程 1

从这里开始，进程 0 就开始准备切换到进程 1 了。在 Linux 0.11 的进程调度机制中，有两种情况可以产生进程切换。

一种情况是：由于进程运行的时间到了，于是进行切换。每个进程在创建时，都被赋予有限的时间片，以此保证所有进程每次都只执行有限的时间。一旦当前进程的时间片被削减为 0 了，就说明这个进程此次执行的时间用完了，只要它是处于用户态，就立即切换到其他进程去执行，以此来保证多进程能够轮流执行。

另一种情况是：由于逻辑执行需要被打断了，于是进程切换。当一个进程处于内核态时，它下一步确实已经没有事务要处理了，或者它下一步要处理的事务需要外设提供的数据来支持，等等，在这种情况下，该进程就不再具备进一步执行的"逻辑条件"了，如果还等着时钟中断产生后再切换到别的进程去执行，就是在无谓地浪费时间，于是它将被立即切换到其他进程去执行。

这两种情况中任何一种情况出现，都会导致进程切换。进程 0 切换到进程 1，表现为第二种情况。进程 0 的事务现在显然已经处理完了，于是通过下面的 pause 函数的执行，最终会执行到 schedule 函数中去，切换到进程 1，如图 2-30 所示。

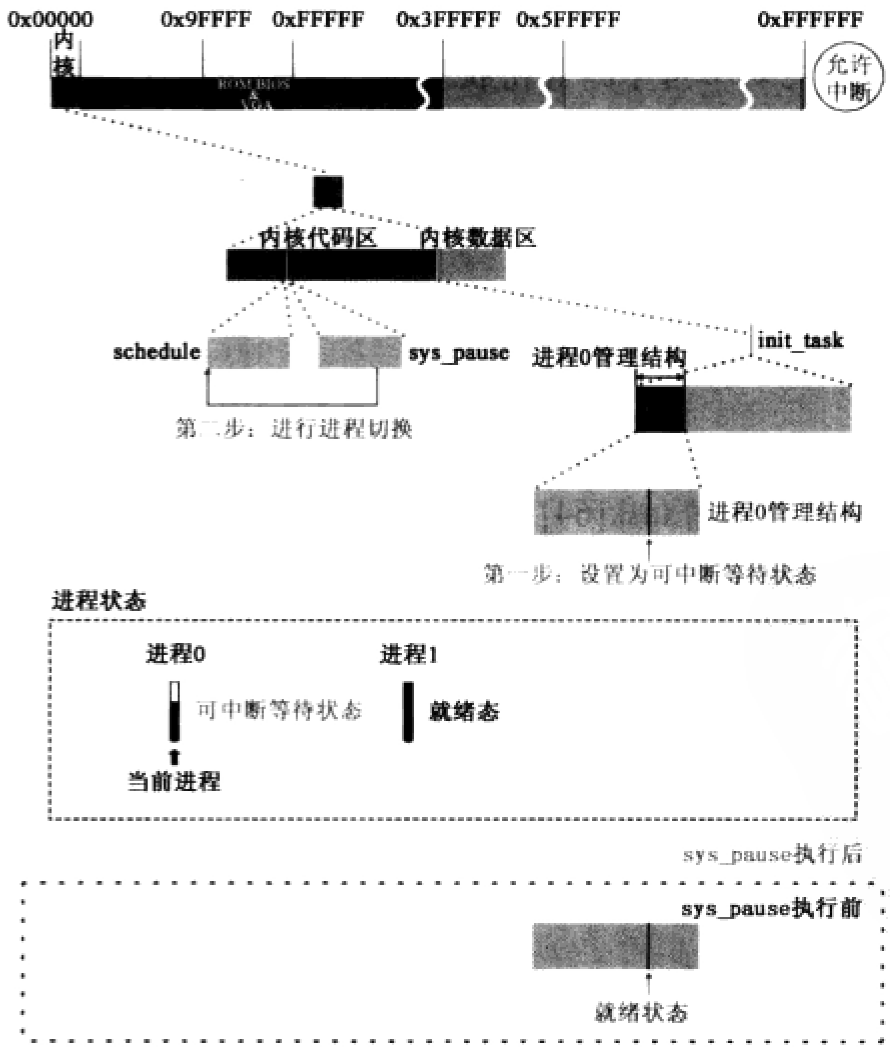


图 2-30 进程 0 挂起并执行调度程序

pause 函数的执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
for(;;) pause();
}
```

通过 main.c 中对 pause 的使用声明 static inline _syscall0(int,pause) 我们得知：

这个函数的调用也会执行到 unistd.h 中的 syscall0 中去，最终也会产生一个软中断，并映射到一个叫做 sys_pause 的系统调用函数去执行，具体映射步骤与 2.2.1 节中 fork 映射步骤大体一致。这里也可以看出，2.1.7 节中为进程 0 进行系统调用所做的准备工作，在这里再次发挥了作用。

进入 sys_pause 函数后，会将进程 0 设置为可中断等待状态，如图 2-30 中第一步所示；并通过调用 schedule 函数，进行进程切换，执行代码如下：

```
// 代码路径：kernel/sched.c:
int sys_pause(void)
{
       current->state = TASK_INTERRUPTIBLE;
       schedule();
       return 0;
}
```

我们在 2.2.4 节中已经对不可中断等待状态进行了讲述，这里所讲的可中断等待状态与它是有所区别的，将进程设置成可中断等待状态意味着，如果产生某种中断，或其他进程给这个进程发送特定信号等，仍然有可能将这个进程的状态改设为就绪状态，使之仍然具备运行的能力。这种意义，适用于 Linux 0.11 中的全部进程。

2.2.12 系统切换到进程 1 执行

接下来进入 schedule 函数，开始分析现在有没有必要进行进程切换，如果有必要，再进行具体的切换操作。首先依据进程槽 task[64] 这个结构，第一次遍历所有进程，执行代码如下：

```
// 代码路径：kernel/sched.c:
void schedule(void)
{
    ...
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
           if (*p) {
                  if ((*p)->alarm && (*p)->alarm < jiffies) {   // 针对报警定时值进行处理
                         (*p)->signal |= (1<<(SIGALRM-1));
                         (*p)->alarm = 0;
                      }
                  if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                  (*p)->state==TASK_INTERRUPTIBLE)   // 针对信号位图进行处理
                     (*p)->state=TASK_RUNNING;
           }
    ...
}
```

只要地址指针不为空，就要针对它们的“报警定时值 alarm”及“信号位图 signal”进行处理（这两个进程管理结构的成员我们在 2.2.7 节中已经介绍），在当前的情况下，这些处理还不会产生具体的效果。尤其是，进程 0 此时并没有收到任何信号，所以它的状态仍然是“可中断等待状态”，不可能转变为“就绪状态”。之后，第二次遍历所有进程，执行代码是：

```
// 代码路径：kernel/sched.c:
void schedule(void)
{
    ...
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
switch_to(next);
}
```

通过分析时间片和进程的状态，if ((*p)->state == TASK_RUNNING && (*p)->counter > c) 来判断此时需要切换到哪个进程去执行，这时候，只有进程 0 和进程 1，且进程 0 是处于可中断等待状态，进程 1 是就绪状态，于是就执行 switch_to(next)，准备切换到进程 1 去执行。如图 2-31 中的第一步所示。

现在开始进程切换具体实施工作，执行代码如下：

```
// 代码路径：include/sched.h:
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \
    "je 1f\n\t" \
    "movw %%dx,%1\n\t" \
    "xchgl %%ecx,_current\n\t" \
    "ljmp %0\n\t" \
    "cmpl %%ecx,_last_task_used_math\n\t" \
    "jne 1f\n\t" \
    "clts\n" \
    "1:" \
    ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
    "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

程序将一直执行到 "ljmp %0\n\t" \ 这一行，这一行将当前各个寄存器值保存在进程 0 的任务状态描述符表 TSS 中，并用进程 1 的任务状态描述符表中的数据来设置各个寄存器的值。用同样的方法处理局部数据描述符表 LDT（在 2.2.9 节中已经对 TSS 和 LDT 进行了介绍）之后，就切换到进程 1 去执行了。如图 2-31 中的第二步所示。

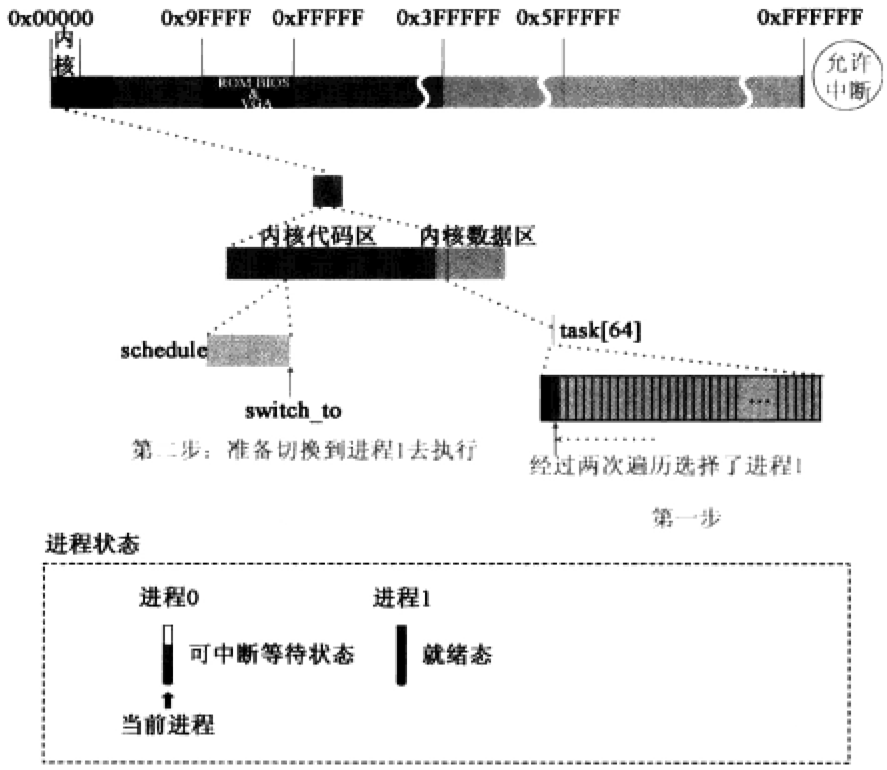


图 2-31　调度进程 1 执行

至此，进程 0 创建进程 1 并切换到进程 1 的工作已经全部完成了，接下来，该进程 1 开始工作了，它将在本章第 3 节中进一步构建环境，使得系统能够以文件的形式与外设交互。

2.3 加载根文件系统

本节将继续介绍本章中第二阶段的内容，这是由进程 1 来完成的，进程 1 将在继承了进程 0 全部能力的基础上，进一步构建环境，使进程能够依托系统与外设“以文件形式”进行数据交互。由于任何对文件的操作，从根本上讲，都要依靠根文件系统的支持，所以进程 1 是通过加载的根文件系统来达到这个目的的。

在本章 2.1.14 节和 2.1.15 节中已经讲解过，系统为进程 0 及其后续创建的进程构建了与硬盘和软盘沟通的环境，且进程 1 也能够享有这种环境，这就为加载根文件系统提供了前提和保障。

本节将分为三个部分来介绍。

第一部分：进程 1 通过对一些与硬盘管理相关的数据结构进行设置，进一步为进程依托系统与硬盘以文件形式进行数据交互创造条件；

第二部分：进程 1 用虚拟盘替代软盘，使之成为根设备；

第三部分：进程 1 以虚拟盘中提供的数据为依据，加载根文件系统。

此三部分的程序结构图如图 2-32 所示。

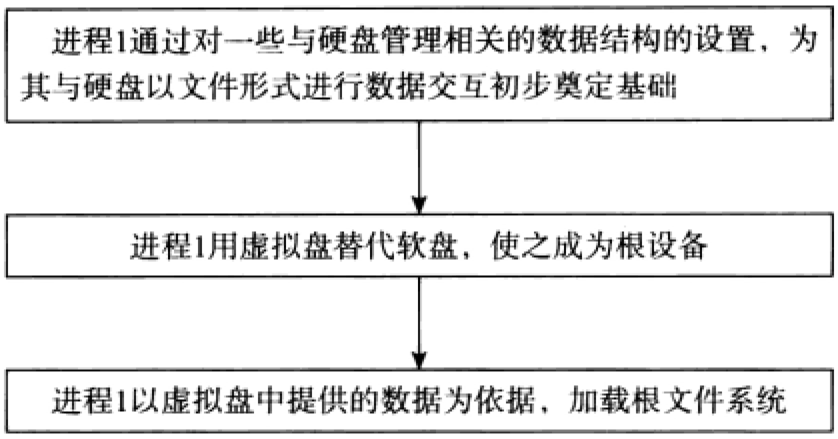


图 2-32　进程 1 加载根文件系统执行过程的程序架构图

第一部分：进程 1 通过对一些与硬盘管理相关的数据结构的设置，为其与硬盘以文件形式进行数据交互初步奠定基础，具体过程如图 2-33 所示。

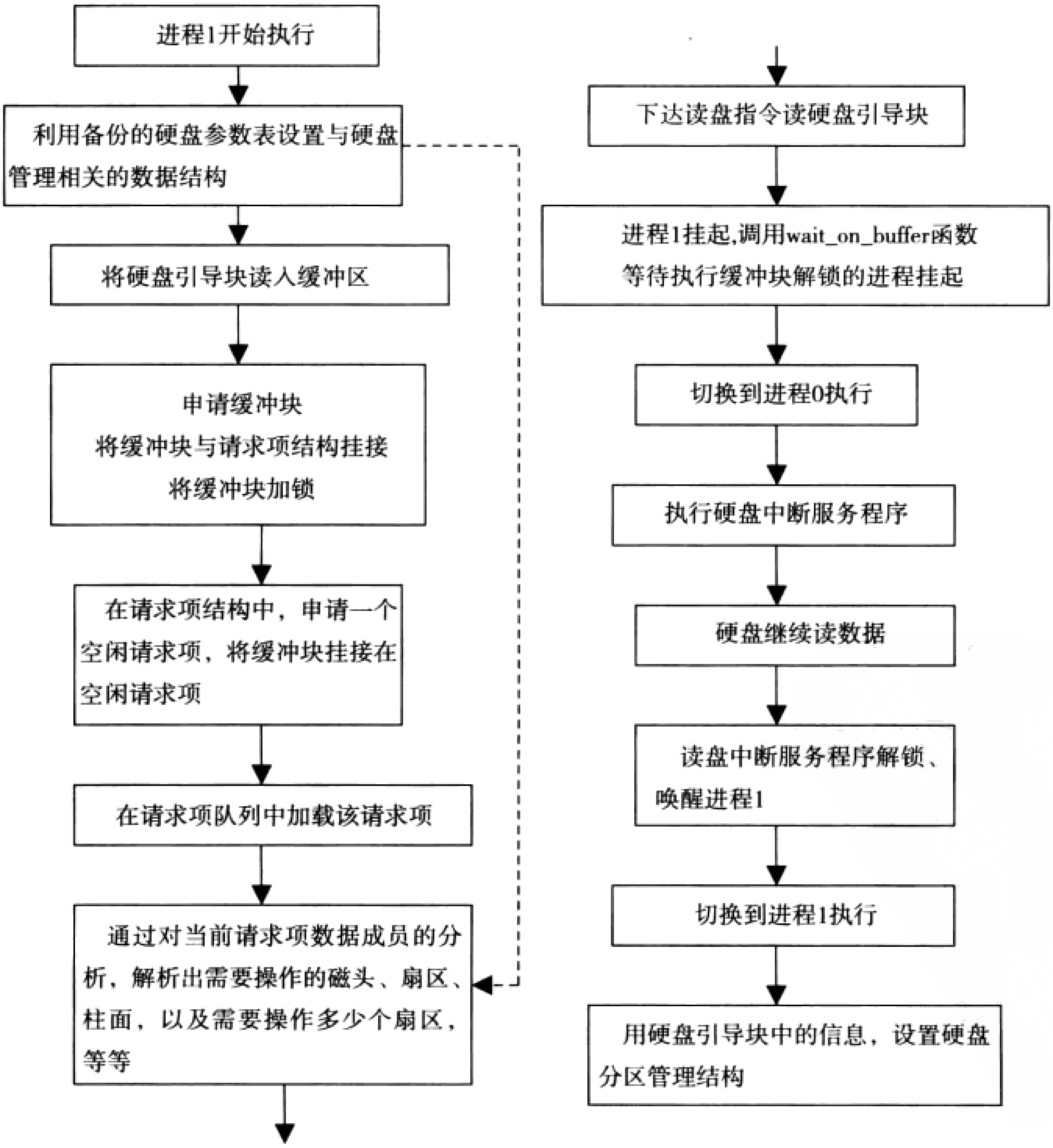


图 2-33　第一部分执行架构图

第二部分：进程 1 用虚拟盘替代软盘，使之成为根设备，具体过程如图 2-34 所示。

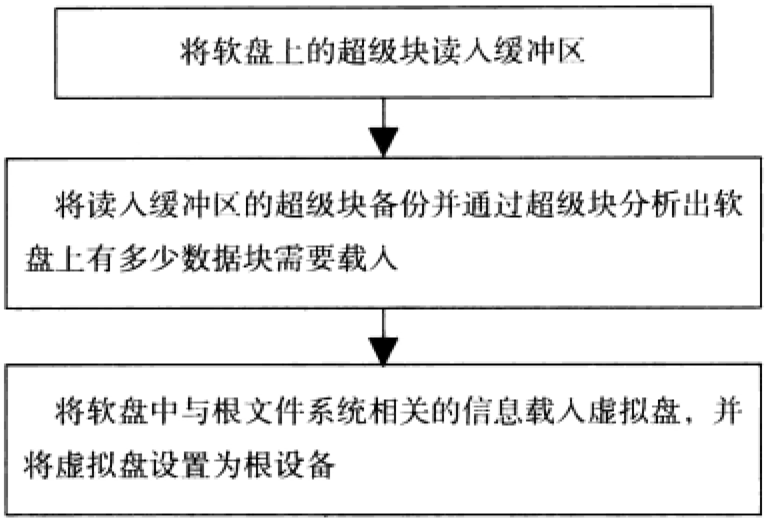


图 2-34　第二部分执行架构图

第三部分：进程 1 以虚拟盘中提供的数据为依据，加载根文件系统，从虚拟盘中读取根文件系统的超级块，并将超级块及其附属信息加载到指定数据结构中，具体过程如图 2-35 所示。

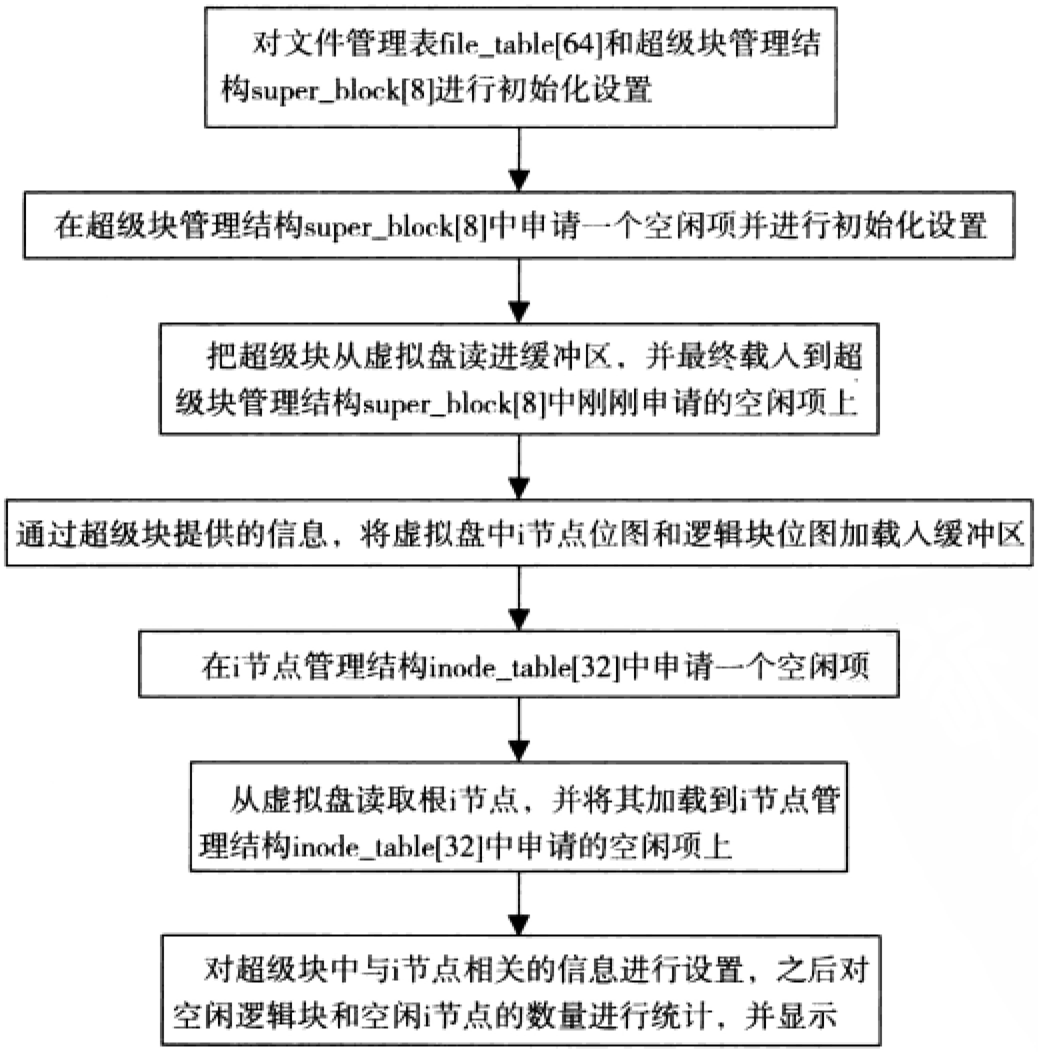


图 2-35　第三部分执行架构图

第一部分中，进程 1、缓冲块、请求项以及硬盘之间的操作关系图如图 2-36 所示。

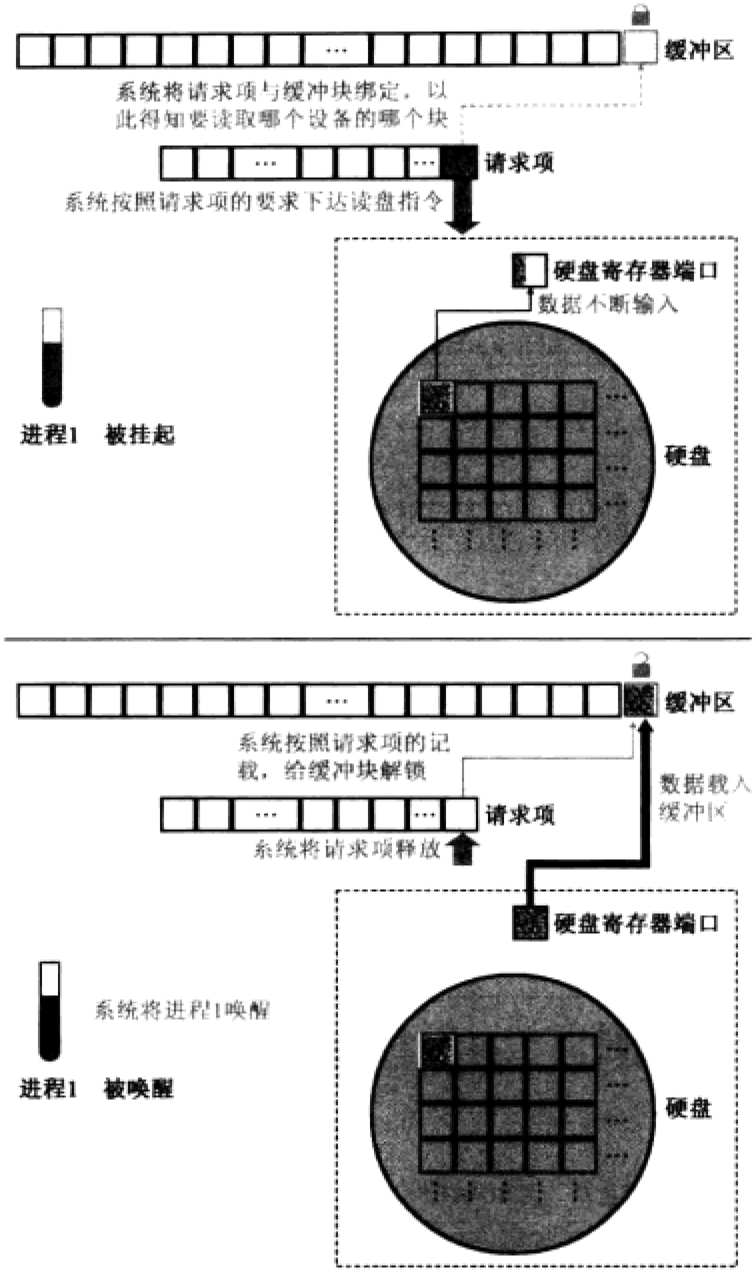


图 2-36　进程、缓冲块、请求项及硬盘的操作关系示意图

第三部分加载根文件系统的效果图如图 2-37、图 2-38、图 2-39 所示。

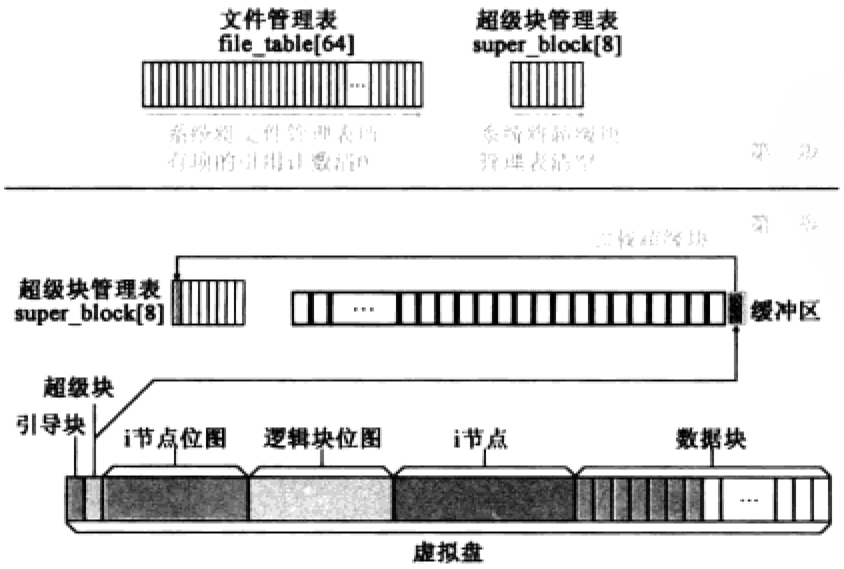


图 2-37　加载根文件系统效果图 1

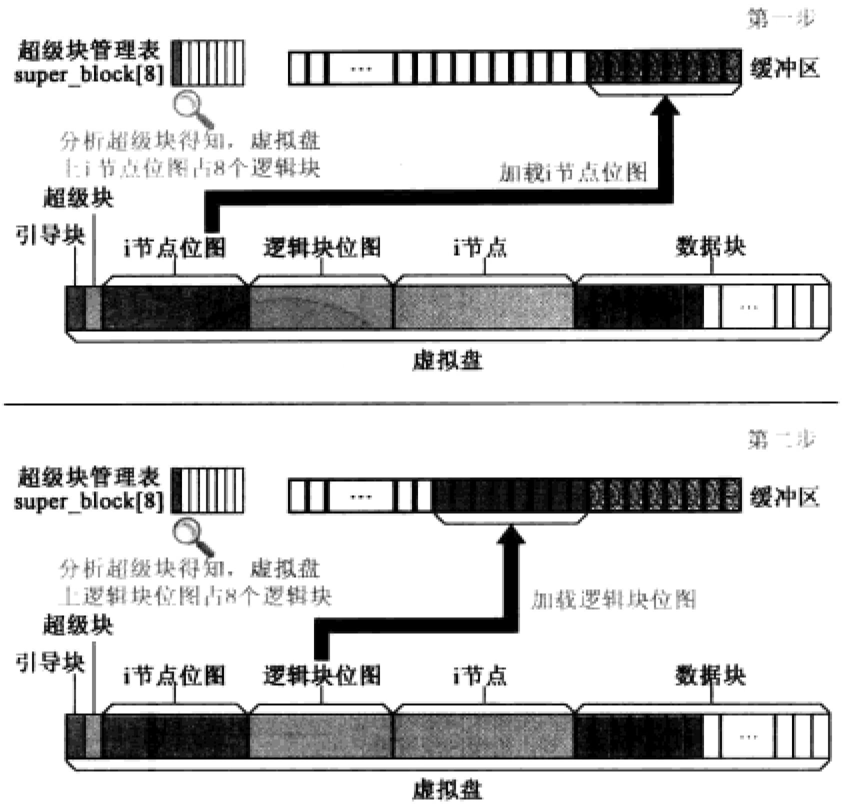


图 2-38　加载根文件系统效果图 2

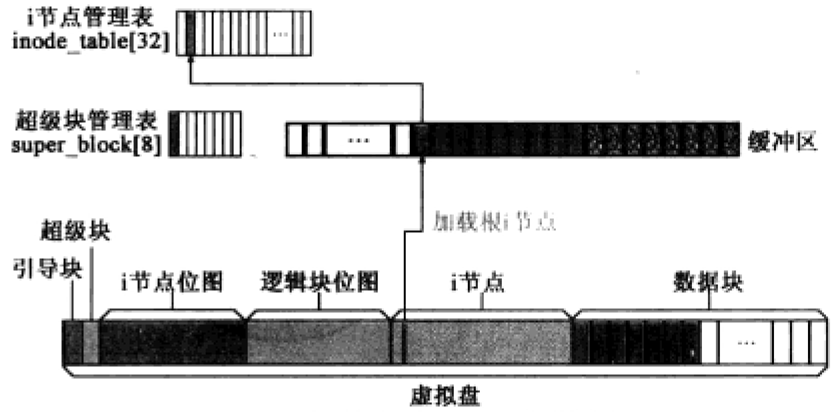


图 2-39　加载根文件系统效果图 3

2.3.1　进程 1 如何开始执行

由于加载根文件系统的工作是由进程 1 完成的，所以我们在正式开始介绍根文件系统的加载过程之前，先来关注一下进程 1 是如何开始执行的。通过 2.2.1 节中的讲述可以得知，进程 1 执行的第一行代码，是从以下函数中的“if (__res >= 0) \”这一行开始执行的：

```
// 代码路径：include/unistd.h:
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
```

```
return -1; \
}
```

这是一个宏函数，展开后表现为：

```
int fork(void)
{
long __res;
__asm__ volatile ("int $0x80"
    : "=a" (__res)
    : "0" (__NR_ fork));
if (__res >= 0)
    return (int) __res;
errno = -__res;
return -1;
}
```

如图 2-40 中的第一步所示。

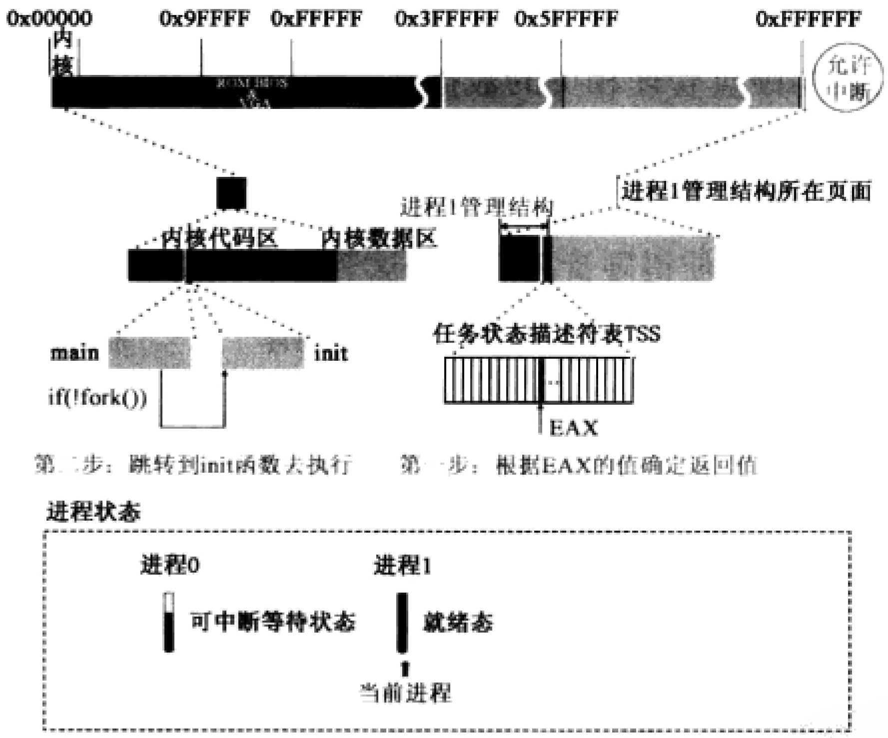


图 2-40　进程 1 开始执行的状态

通过 2.2.10 节中的介绍我们得知，此时 __res 的值，就是进程 1 的任务状态描述符表 TSS 中 eax 的值，这个值在 2.2.7 节中被初始化成 0，即 p->tss.eax = 0; 这行代码，因此，当执行到 return (type) __res; \ 这一行时，返回值就是 0；返回后，同样执行到 main 函数中 if (!fork()) 这一行，这样直接结果将为“真”，于是开始调用 init 函数，执行代码为：

```
// 代码路径：init/main.c:
void main(void)
{
    if (!fork()) {
        init();
    }
}
```

2.3.2 进程 1 开始执行

进程 1 在这里将正式开始完成本节第一部分的内容，即，为进程 1 依托系统与硬盘以文件形式进行数据交互初步创造条件。进入 init 函数后，先要调用 setup 函数，执行代码如下：

```
// 代码路径：init/main.c:
void init(void)
{
    ...
    setup((void *) &drive_info);
    ...
}
```

整个 2.3 节所讲述的内容，都是这个函数实现的。

这个函数最终也会映射到一个系统调用函数 sys_setup 中，具体的映射方式，与 2.2.1 节中 fork 函数映射到 sys_fork 函数的方式大体类似，这是由于进程 1 共享了进程 0 的工作环境，所以就包括可以进行系统调用了。

进入 sys_setup 函数后，本章 2.1.1 节中备份的硬盘参数表开始发挥作用了，系统将用这些备份的硬盘参数，设置一些与硬盘管理相关的数据结构，包括 hd_info 和 hd，对这些数据结构的设置，将使系统以后对数据交互操作的控制更加方便，其中 hd_info 这个结构将在逻辑块对应到硬盘具体位置的解析过程中发挥作用，这一点我们将在 2.3.6 节中介绍；hd 这个结构，将用硬盘引导块中的信息进行设置，这将使得硬盘可以被分区管理。图 2-41 形象地表示了这一过程。

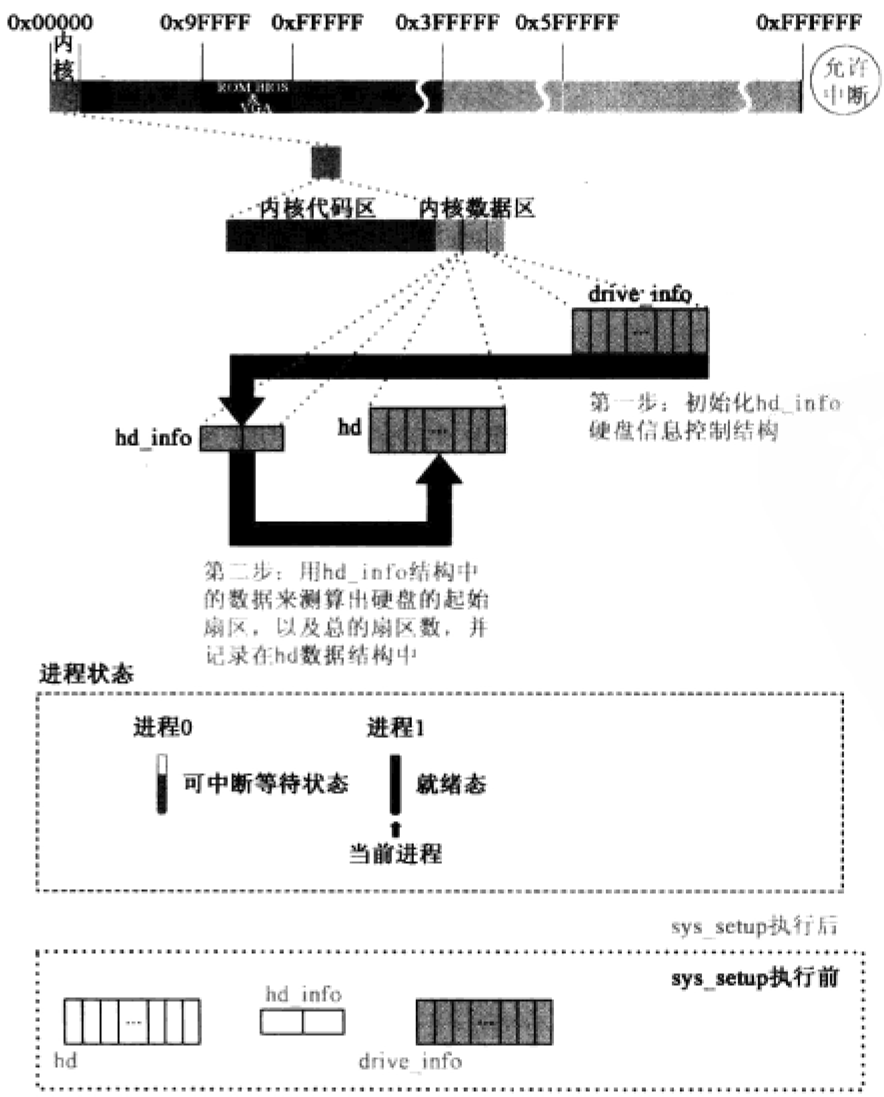


图 2-41 初始化硬盘控制数据结构

具体的执行代码是：

```
// 代码路径：kernel/blk_dev/hd.c:
int sys_setup(void * BIOS)
{
    ...
    for (drive=0 ; drive<2 ; drive++) {
        hd_info[drive].cyl = *(unsigned short *) BIOS;
        hd_info[drive].head = *(unsigned char *) (2+BIOS);
        hd_info[drive].wpcom = *(unsigned short *) (5+BIOS);
        hd_info[drive].ctl = *(unsigned char *) (8+BIOS);
        hd_info[drive].lzone = *(unsigned short *) (12+BIOS);
        hd_info[drive].sect = *(unsigned char *) (14+BIOS);
        BIOS += 16;
    }
    ...
    for (i=0 ; i<NR_HD ; i++) {
        hd[i*5].start_sect = 0;
        hd[i*5].nr_sects = hd_info[i].head*
                hd_info[i].sect*hd_info[i].cyl;
    }
    if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
        if (cmos_disks & 0x0f)
            NR_HD = 2;
        else
            NR_HD = 1;
    else
        NR_HD = 0;
    for (i = NR_HD ; i < 2 ; i++) {
        hd[i*5].start_sect = 0;
        hd[i*5].nr_sects = 0;
    }
    ...
}
```

2.3.3　进程 1 开始以数据块的形式操作硬盘

进程 1 开始完成本节第一部分任务。我们假设系统只有一块硬盘，从这里开始，就要把这个硬盘的引导块读入缓冲区了，在本章的 2.1.13 节中对缓冲区的设置就要发挥作用了。这是进程 1 首次对硬盘自身的特征数据进行读取，将来在以文件的形式与硬盘进行数据交互时，这些数据都将发挥作用。这个采集工作将通过调用 bread 函数逐步实现，执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
    int sys_setup(void * BIOS)
    {
        ...
        if (!(bh = bread(0x300 + drive*5,0)))
        ...
    }
```

这两个实参的意义是这样的：0x300 + drive*5 代表设备号，此时它代表硬盘的设备号；0 代表将要从硬盘上读取第 0 号逻辑块，即最开始的两个扇区（1KB），那里存储着硬盘引导块的内容。

进入 bread 函数后，进程 1 先要调用 getblk 函数，因为从硬盘载入的数据都要先进入缓冲

区，理由在本章 2.1.13 节中已经说明，所以要先在缓冲区中申请一个缓冲块，执行代码如下：

```
// 代码路径：fs/buffer.c:
struct buffer_head * bread(int dev,int block)
{
      ...
      if (!(bh=getblk(dev,block)))
      ...
}
```

查找缓冲块的主要步骤在图 2-42 中已有形象的说明，以下将结合代码来深入分析这一过程。

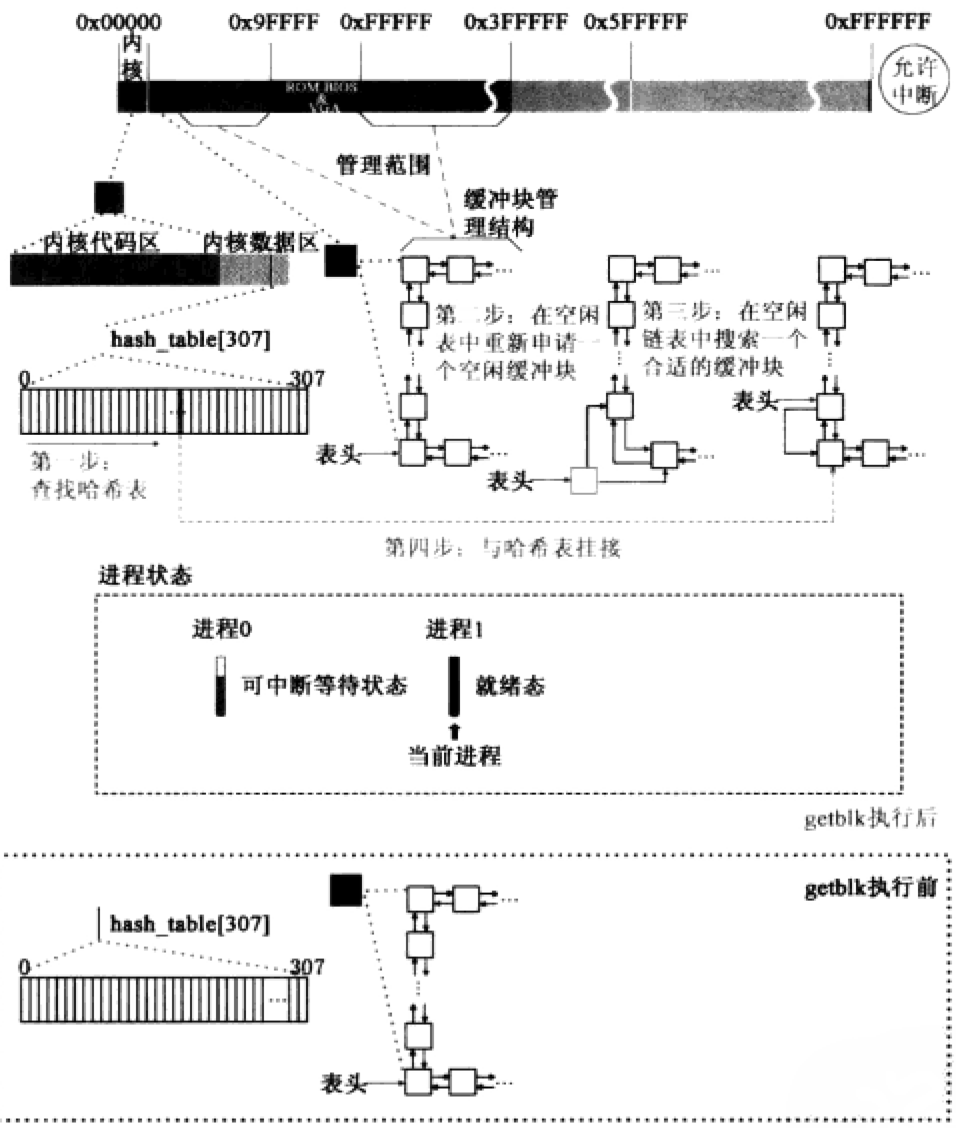


图 2-42　查找缓冲块

进入 getblk 函数后，申请的步骤是这样的：系统要先在缓冲区里面，通过哈希表结构找一下，看看此时有没有哪个缓冲块中所对应的设备号和块号，与当前想要找的那个逻辑块的块号和设备号相匹配，这是因为，缓冲块将来要大量频繁的被用于主机与外设的数据交互，而存储在缓冲块的数据不会立即消失，这样，本次要读取的缓冲块有可能已经在本次读取之前就存在于缓冲区中了，那么本次就没必要再重新从外设上读取一遍了，直接使用就可以。如图 2-42 中的第一步所示。

使用哈希表进行查询，可以充分利用哈希结构的特点提高查询速度，进而提高系统执行的综合效率。执行代码如下：

```
// 代码路径：fs/buffer.c:
struct buffer_head * getblk(int dev,int block)
{
    ...
    if (bh = get_hash_table(dev,block))
    ...
}
```

进程 1 继续执行，进入 get_hash_table 函数后，如果能找到指定缓冲块，直接用就可以了。但由于现在是第一次使用缓冲区，所以不可能存在匹配的缓冲块，因此从 get_hash_table 函数退出后，只能在空闲链表中新申请一个合适的缓冲块了（因为所有缓冲块都是绑定在空闲表中的，所以要在空闲表中申请新的缓冲块），如图 2-42 中的第二步所示，执行代码如下：

```
// 代码路径：fs/buffer.c:
struct buffer_head * getblk(int dev,int block)
{
   ...
   tmp = free_list;
   do {
       if (tmp->b_count)
           continue;
       if (!bh || BADNESS(tmp)<BADNESS(bh)) {
           bh = tmp;
           if (!BADNESS(tmp))
               break;
       }
   } while ((tmp = tmp->b_next_free) != free_list);
   ...
}
```

申请到缓冲块后，对它进行初始化设置，由于空闲表是一个“双环链表”，为了提高“申请到空闲缓冲块的概率”，就要调整它在空闲表中的位置，并重组空闲表，这一点在图 2-42 的第三步中可以得到证实，之后，让设备号和块号与该缓冲块相对应，并把这个缓冲块挂接在哈希表结构中，之后返回这个缓冲块的指针，申请工作到此完毕，执行代码如下：

```
// 代码路径：fs/buffer.c:
struct buffer_head * getblk(int dev,int block)
{
    ...
    bh->b_count=1;
    bh->b_dirt=0;
    bh->b_uptodate=0;
    remove_from_queues(bh);
    bh->b_dev=dev;
    bh->b_blocknr=block;
    insert_into_queues(bh);
    return bh;
}
```

2.3.4　将找到的缓冲块与请求项挂接

进程 1 继续完成本节的第一部分任务。缓冲块申请完毕后，返回到 bread 函数中，调用 ll_ rw_block 这个函数，将缓冲块与请求项结构挂接，如图 2-43 所示。

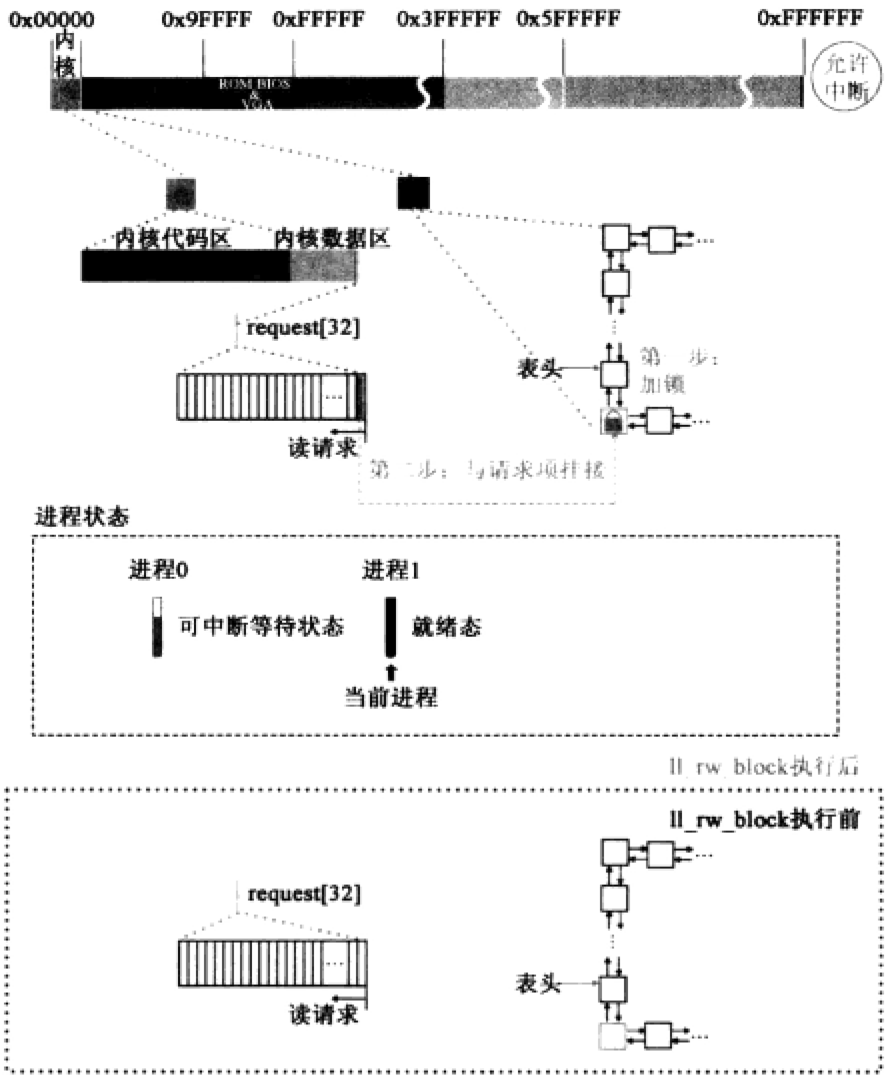


图 2-43　缓冲块与请求项挂接

2.1.6 节中对请求项管理结构的设置在这里就要开始发挥作用了，那一节中对请求项的作用已经做了说明，执行代码如下：

```
// 代码路径：fs/buffer.c:
struct buffer_head * bread(int dev,int block)
{
      ...
      ll_rw_block(READ,bh);
      ...
}
```

进入 ll_rw_block 函数后，先判断缓冲块对应的设备是否存在，以及这个设备的请求项函数是否挂接正常，如果正常，说明可以操作这个缓冲块。之后调用 make_request 这个函数，准备将缓冲块与请求项建立关系，执行代码如下：

```
// 代码路径：kernel/blk_dev/ll_rw_block.c:
void ll_rw_block(int rw, struct buffer_head * bh)
{
     ...
     if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
    !(blk_dev[major].request_fn)) {
```

```
        printk("Trying to read nonexistent block-device\n\r");
        return;
    }
    make_request(major,rw,bh);
}
```

进程 1 继续执行，进入 make_request 函数后，先要将这个缓冲块加锁，目的是让这个缓冲块在解锁之前将不再被操作，这是因为这个缓冲块现在已经被使用了，如果此时它再被用于它途，这里面的数据就会发生混乱，如图 2-43 右边的缓冲块管理结构所示，其中选中的那个缓冲块对应的管理结构已加锁。执行代码如下：

```
//代码路径：kernel/blk_dev/ll_rw_block.c:
static void make_request(int major,int rw, struct buffer_head * bh)
{
    ...
    lock_buffer(bh);
    ...
}
```

之后，在请求项结构中，申请一个空闲请求项，准备与这个缓冲块相挂接。值得注意的是，如果是读请求，则从整个请求项结构的最末端开始寻找空闲请求项，如果是写请求，则从整个结构的三分之二处申请空闲请求项，这是因为从用户使用系统的心理角度来讲，用户更希望读取的数据能更快地显现出来，所以给读取操作以更大的空间，在这时候，请求项结构是第一次被使用，而且是读请求，所以在请求项结构的末端找到一个空闲的请求项，如图 2-43 中的 request[32] 结构所示，其中的最后一项已被选中。执行代码如下：

```
//代码路径：kernel/blk_dev/ll_rw_block.c:
static void make_request(int major,int rw, struct buffer_head * bh)
{
    ...
    if (rw == READ)
        req = request+NR_REQUEST;
    else
        req = request+((NR_REQUEST*2)/3);
    while (--req >= request)
        if (req->dev<0)
            break;
    ...
}
```

之后，缓冲块与请求项正式挂接，并对这个请求项各个成员进行初始化，执行代码是：

```
//代码路径：kernel/blk_dev/ll_rw_block.c:
static void make_request(int major,int rw, struct buffer_head * bh)
{
    ...
    req->dev = bh->b_dev;
   req->cmd = rw;
   req->errors=0;
   req->sector = bh->b_blocknr<<1;
   req->nr_sectors = 2;
```

```
    req->buffer = bh->b_data;
    req->waiting = NULL;
    req->bh = bh;
    req->next = NULL;
    ...
}
```

2.3.5 将请求项与硬盘处理函数挂接

进程 1 继续完成本节第一部分任务。接下来，调用 add_request 这个函数，往请求项队列中加载该请求项，执行代码如下：

```
// 代码路径：kernel/blk_dev/ll_rw_block.c:
static void make_request(int major,int rw, struct buffer_head * bh)
{
    ...
    add_request(major+blk_dev,req);
}
```

在 2.1.14 节中讲过，系统为进程 0 以及后续进程构建了与硬盘沟通的环境，进程 1 可以与硬盘进行沟通，进入 add_request 后，先对当前硬盘设备的工作情况进行分析，此时硬盘正好空闲，所以设置该请求项为当前请求项，并调用硬盘请求项处理函数去给硬盘发送具体的读盘命令 (dev->request_fn)();，即调用 do_hd_request 函数，图 2-44 中给出了请求项管理结构与 do_hd_request 函数的对应关系。理由在本章 2.1.14 节中已经说明。

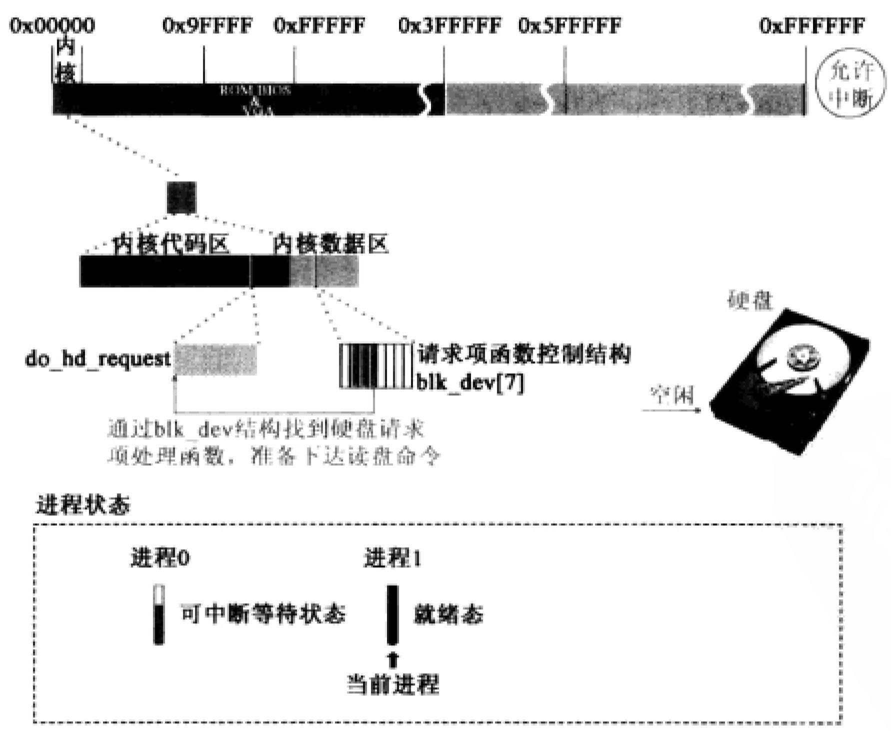


图 2-44 将请求项与硬盘处理函数挂接

执行代码如下：

```
// 代码路径：kernel/blk_dev/ll_rw_block.c:
static void add_request(struct blk_dev_struct * dev, struct request * req)
{
    ...
```

```
    if (!(tmp = dev->current_request)) {
        dev->current_request = req;
        sti();
        (dev->request_fn)();
        return;
    }
    ...
}
```

2.3.6　进行硬盘读盘前的准备工作

进程 1 继续完成本节第一部分任务。接下来要进入 do_hd_request 函数去执行，为读盘做最后的准备工作，具体的准备过程如图 2-45 所示。

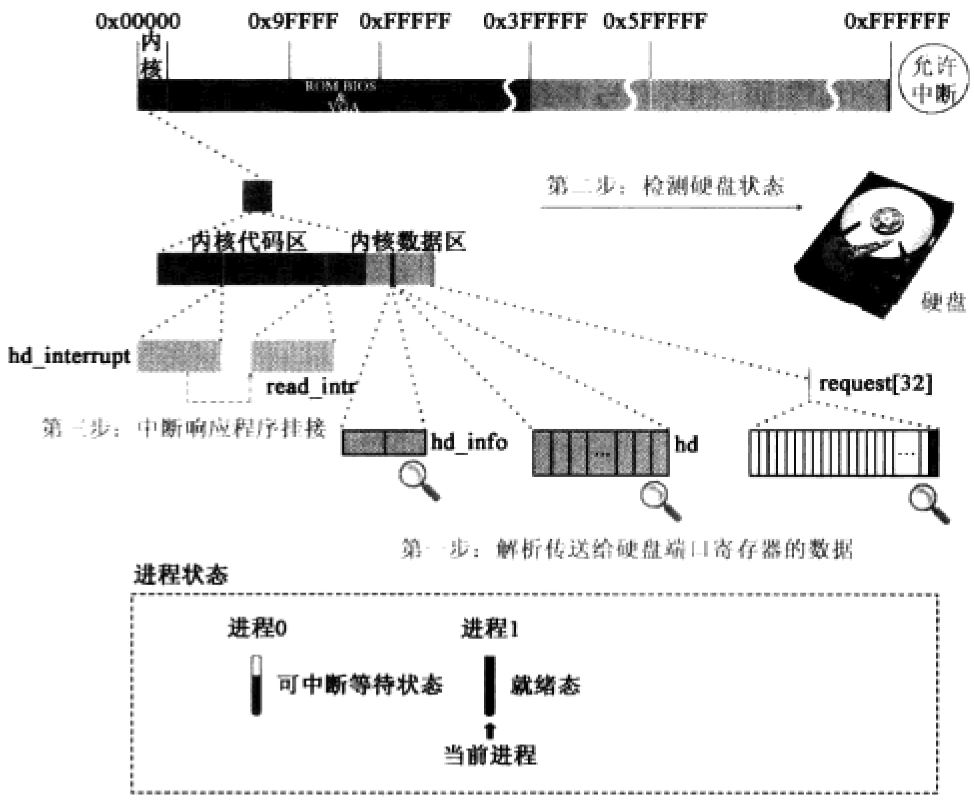


图 2-45　读盘操作前的主要准备工作

先通过对当前请求项数据成员的分析，解析出需要操作的磁头、扇区、柱面，以及需要操作多少个扇区，等等，如图 2-45 的第一步所示。此时，2.3.2 节中介绍的 hd_info 和 hd 结构都在解析时发挥作用了，执行代码如下：

```
//代码路径：kernel/blk_dev/hd.c:
void do_hd_request(void)
{
    ...
    dev = MINOR(CURRENT->dev);
    block = CURRENT->sector;
    ...
    block += hd[dev].start_sect;
    dev /= 5;
    __asm__("divl %4":"=a" (block),"=d" (sec):"0" (block),"1" (0),
        "r" (hd_info[dev].sect));
    __asm__("divl %4":"=a" (cyl),"=d" (head):"0" (block),"1" (0),
        "r" (hd_info[dev].head));
```

```
    sec++;
    nsect = CURRENT->nr_sectors;
     ...
}
```

之后，检测硬盘此时的状态是否正常，如图 2-45 中第二步所示，执行代码如下：

```
//代码路径：kernel/blk_dev/hd.c:
void do_hd_request(void)
{
     ...
     if (reset)
     ...
     if (recalibrate)
     ...
}
```

如果正常，则针对命令的性质（读 / 写）给硬盘发送操作命令，很显然，现在是读操作（因为要读硬盘的引导块），所以接下来要调用 hd_out 函数来下达最后的硬盘操作指令，执行代码如下：

```
//代码路径：kernel/blk_dev/hd.c:
void do_hd_request(void)
{
     ...
     else if (CURRENT->cmd == READ) {
          hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
     ...
}
```

注意看最后两个实参，WIN_READ 表示接下来要进行读操作了，read_intr 是读盘操作对应的中断服务程序，所以要提取它的函数地址，准备挂接，这一动作反映在图 2-45 中的第三步，请读者注意，这是通过 hd_out 函数实现的，读盘请求就挂接 read_intr；如果是写盘，那就不是 read_intr，而是 write_intr 了，下一节还会详细讲解这个问题。

2.3.7 给硬盘下达读盘指令

进程 1 继续完成本节第一部分任务。接下来进入 hd_out 函数中去执行读盘的最后一步，即下达读盘指令。如图 2-46 中的第一步所示。

执行代码如下：

```
//代码路径：kernel/blk_dev/hd.c:
static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
        unsigned int head,unsigned int cyl,unsigned int cmd,
        void (*intr_addr)(void))
{
     ...
     do_hd = intr_addr;
     outb_p(hd_info[drive].ctl,HD_CMD);
     port=HD_DATA;
     outb_p(hd_info[drive].wpcom>>2,++port);
     outb_p(nsect,++port);
     outb_p(sect,++port);
```

```
        outb_p(cyl,++port);
        outb_p(cyl>>8,++port);
        outb_p(0xA0|(drive<<4)|head,++port);
        outb(cmd,++port);
    }
```

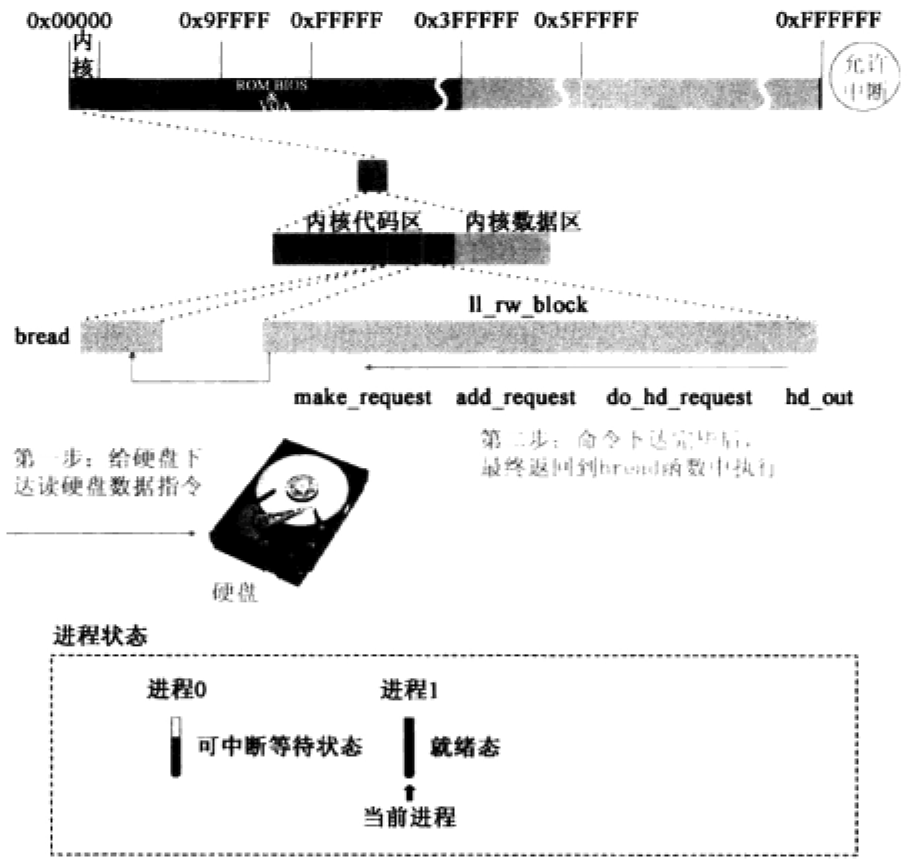


图 2-46　给硬盘端口寄存器传递参数和当前进程挂起

其中 do_hd = intr_addr; 这一行代码是把读盘服务程序与硬盘中断操作程序相挂接，这里面的 do_hd 是 _hd_interrupt: 处 xchgl _do_hd,%edx 这一行代码所描述的内容，见代码：

```
// 代码路径：kernel/system_call.s:
_hd_interrupt:
    ...
    xchgl  _do_hd,%edx
    ...
```

现在要处理读盘操作，所以挂接的就是实参 read_intr 了，如果是写盘，挂接的就应该是 write_intr 函数了，可见，_do_hd 这个挂接点，可以灵活挂接；之后，下达读盘指令，硬盘就开始将盘区中的数据不断地读入到它对应的数据寄存器端口中，此时，内核的程序也该返回了，将会沿着 hd_out--- do_hd_request--- add_request--- make_request... 这条路线，一直返回到 bread 中的 ll_rw_block 函数中，之后，继续在 bread 中执行，准备执行 wait_on_buffer 函数，这些函数在内存中的具体位置见图 2-46 的第二步所示。

2.3.8　进程 1 由于等待读盘操作挂起

进程 1 继续完成本节第一部分任务。此时，进程 1 该做的工作已经做完了，现在硬盘正在把引导块的内容读出，进程 1 如果继续执行，则需要对引导块中的数据进行操作，但现在这些数据还没有从

硬盘上读出来，所以进程 1 接下来将准备挂起。于是调用 wait_on_buffer 函数，执行代码如下：

```
// 代码路径：fs/buffer.c:
struct buffer_head * bread(int dev,int block)
{
    ...
    wait_on_buffer(bh);
    ...
}
```

这个函数的功能是：将等待执行缓冲块解锁的进程挂起。

进入 wait_on_buffer 函数后，首先要关中断，如图 2-47 中第一步所示。然后，判断刚才申请到的缓冲块是否被加锁了，执行代码如下：

```
// 代码路径：fs/buffer.c:
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
    ...
}
```

此时这个缓冲块确实加锁了，这一点在本章 2.3.4 节中已经说明。之后调用 sleep_on 函数，准备将进程 1 挂起，因为进程 1 下面的执行需要该缓冲块中数据的支持，所以在数据载入之前，进程 1 不能执行，必须挂起。对应图 2-47 中的第二步操作。

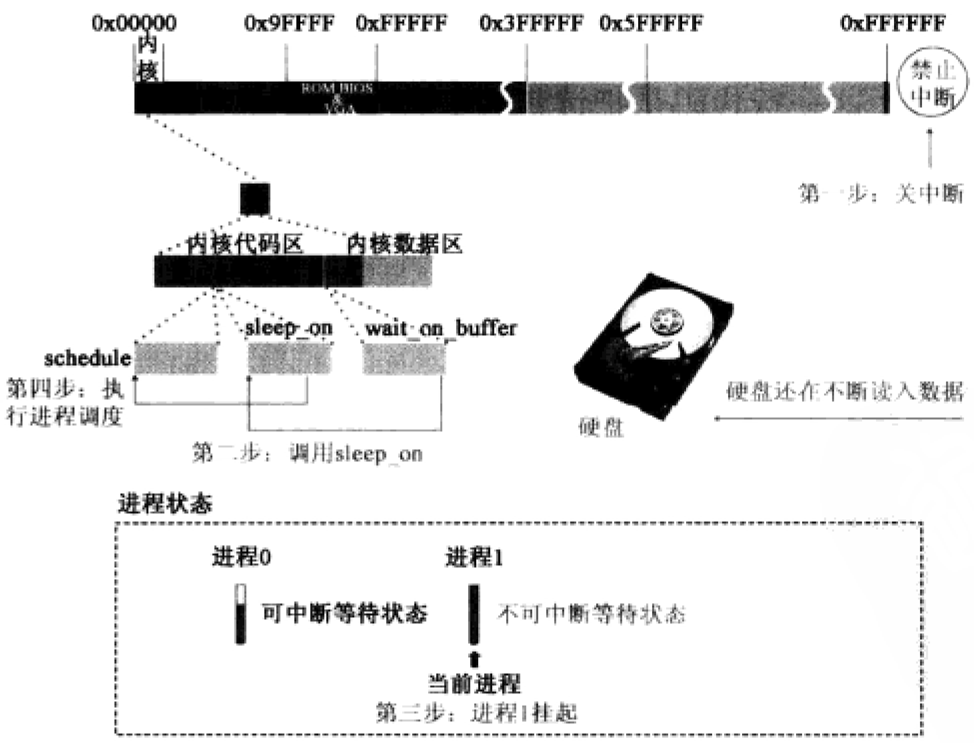


图 2-47　进程 1 挂起并执行调度

执行代码如下：

```
// 代码路径：fs/buffer.c:
static inline void wait_on_buffer(struct buffer_head * bh)
{
    ...
```

```
        sleep_on(&bh->b_wait);
        ...
    }
```

进入 sleep_on 函数后，将进程 1 设置为不可中断等待状态，如图 2-47 中第三步所示，进程 1 挂起；然后调用 schedule 函数，准备进程切换，执行代码如下：

```
//代码路径：kernel/sched.c:
void sleep_on(struct task_struct **p)
{
    ...
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    ...
}
```

2.3.9　系统切换到进程 0 执行

进程 1 继续完成本节第一部分任务。此时将准备进程切换，进入 schedule 函数后，经过一系列执行，将切换到进程 0 去执行。图 2-48 给出了切换过程的主要步骤。

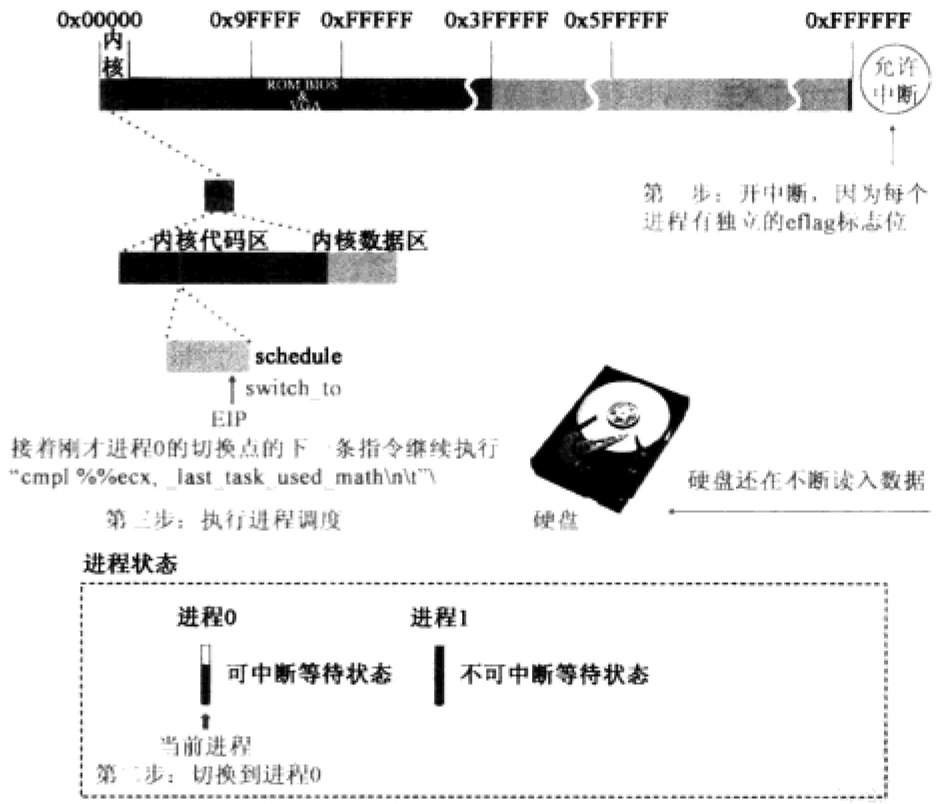


图 2-48　切换到进程 0 去执行

具体执行步骤 2.2.12 节中已经说明，但第二次遍历 task[64] 的时候，与 2.2.12 节中执行的结果不一样。这时候一共只有两个进程，进程 0 的状态是可中断等待状态，进程 1 的状态也已经刚刚被设置成了不可中断等待状态，而常规的进程切换的条件是，时间片要有剩余且必须是就绪状态，两者要同时满足，即代码 "if ((*p)->state == TASK_RUNNING && (*p)->counter > c)" 给出的条件，现在两个进程都不是就绪状态了，无法满足进程切换的常规条件。于是，接下来就要执行默认方案了，值得关注的代码如下：

```
// 代码路径：kernel/sched.c:
void schedule(void)
{
    ...
    while (1) {
     c = -1;
       next = 0;
         while (--i) {
             if (!*--p)
                 continue;
             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                 c = (*p)->counter, next = i;
           }
         if (c) break;
    ...
           }
switch_to(next);
}
```

遍历结束后，c 的值将仍然是 -1，所以 next 仍然是 0，这个 next 就是要切换到进程的进程号。可见，如果没有合适的进程，next 的数值将永远是 0，那么就会默认切换到进程 0 中去执行了，具体切换过程在 2.2.12 节中已经说明。

2.3.10 进程 0 的执行过程

现在已经切换到进程 0 去执行，将要执行的代码所在的函数如下：

```
// 代码路径：kernel/sched.h:
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \
   "je 1f\n\t" \
   "movw %%dx,%1\n\t" \
   "xchgl %%ecx,_current\n\t" \
   "ljmp %0\n\t" \
   "cmpl %%ecx,_last_task_used_math\n\t" \
   "jne 1f\n\t" \
   "clts\n" \
   "1:" \
   ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
   "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

在 2.2.12 节中已经说明，当时进程 0 切换到进程 1 时，是从 "ljmp %0\n\t" \ 这一行代码切换的，所以，现在进程 0 就要从这行代码的下一行开始执行，即 "cmpl %%ecx,_last_task_used_math\n\t" \ 这行代码，如图 2-48 中的第三步所示。

switch_to 执行完毕后，进程 0 返回到 sys_pause 中执行了，这是因为当时是在 sys_pause 这个函数中开始实施的进程切换，这一点在 2.2.11 节中已经说明。sys_pause 这个函数将在 for(;;) 这个循环里面被反复调用，理由在 2.2.11 节中也已经说明。所以，又会继续调用 schedule 进行进程切换，而再次切换的时候，由于两个进程还都不是就绪状态，所以仍然切换到进程 0，理

由在本章 2.3.9 节中已经说明，继续执行 switch_to，但这回执行步骤不一样了，"cmpl %%ecx,_current\n\t" 这一行的意思是：如果切换到的进程就是当前进程，就直接跳转到标号 1 处，直接返回。此时当前进程正是进程 0，正好符合这个条件，所以，又回到进程 0，继续循环执行这个动作。这部分内容如图 2-49 所示，请读者注意其中的进程 0 的切换点，及其与图 2-48 中所示的切换点的区别。这也是操作系统提高执行效率的一个具体体现。当需要切换到的进程就是当前进程时，在 switch_to 中就少执行一段代码，提高系统效率。

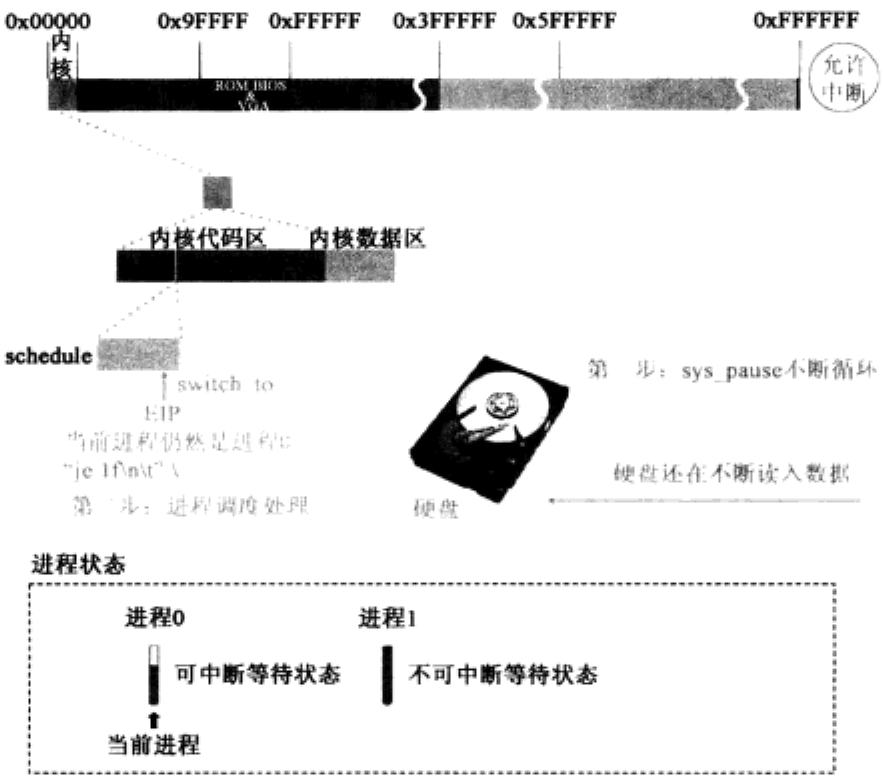


图 2-49　进程 0 的循环执行过程

由此可见，在系统中所有进程的状态都不是就绪态时，将由进程 0 来接管这种情况下的运算，即便进程 0 并不做什么实质性的工作，也要由它来接管，这是因为如果此时没有进程 0 接管执行，而是由系统接管执行，将来再切换的时候，系统自身就要与进程一起参与切换，就要满足进程的全部可切换条件，这与既定的设计方案不符，将会导致大麻烦，比如，进程可以被设置为不可中断等待状态，如果系统也被设置为这种状态，那整个系统就瘫痪了。

所以，系统中进程的切换，说到底还是进程间的切换，这是系统既定的设计方案，将来如果有其他进程被唤醒了，可以执行了，再从进程 0 切换到那个唤醒的进程，这样才能符合这个既定的方案。

与此同时，硬盘也正忙着把刚才缓冲区指定的数据载入至它的数据寄存器中。

2.3.11　进程 0 执行过程中发生硬盘中断

进程 0 执行了一段时间后，硬盘在某一时刻已经把一个扇区的数据读出来了，产生硬盘中断。CPU 接到中断指令后，终止正在执行的程序，即进程 0，终止的位置肯定在进程 0 中 for 循环里面的 pause 某行指令处，如图 2-50 中的第一步所示。

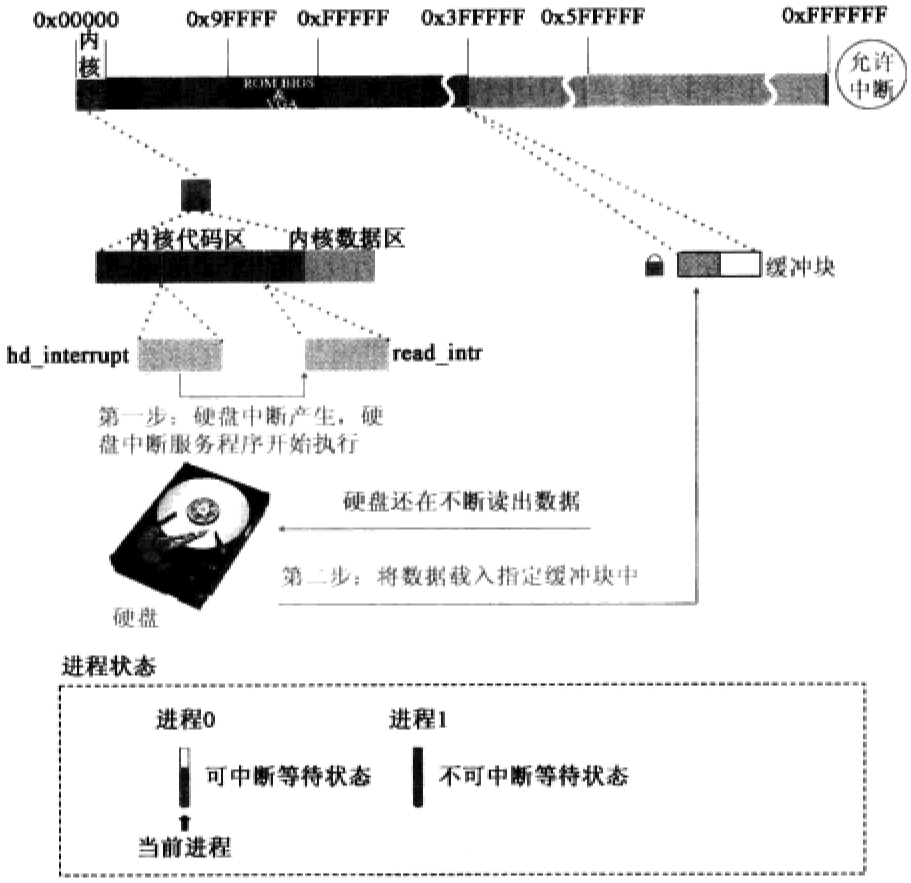


图 2-50　硬盘中断处理过程

然后转而去执行硬盘中断服务程序。执行代码如下：

```
// 代码路径：kernel/system_call.s:
_hd_interrupt:
    pushl %eax
    pushl %ecx
    pushl %edx
    push %ds
    push %es
    push %fs
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    movl $0x17,%eax
    mov %ax,%fs
    movb $0x20,%al
    outb %al,$0xA0
    jmp 1f
1:  jmp 1f
1:  xorl %edx,%edx
    xchgl _do_hd,%edx
    testl %edx,%edx
    jne 1f
    movl $_unexpected_hd_interrupt,%edx
1:  outb %al,$0x20
    call *%edx
...
```

先将一些寄存器的数据压栈以保存进程 0 终止处的现场，之后，执行 _do_hd 处的读盘中断处理程序，对应的代码应该是 call *%edx 这一行，这个 edx 里面，承载着读盘中断处理程序

read_intr 的地址。

然后，read_intr 函数会将硬盘已经导入到数据寄存器端口中的数据，复制到刚才被锁定的那个缓冲块中，这时候应该是 1 个扇区的数据，256 字，即 512 字节，如图 2-50 中的第二步所示，注意其中的缓冲块中已有 512 字节的数据了。执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
static void read_intr(void)
{
    ...
    port_read(HD_DATA,CURRENT->buffer,256);
    CURRENT->errors = 0;
    CURRENT->buffer += 512;
    CURRENT->sector++;
    ...
}
```

但是，每个缓冲块的数据是 1024 字节，请求项要求的也是 1024 字节，所以这次仅仅读出了一半，因此硬盘在把刚才这 512 字节数据读进寄存器并发中断后，会一直不停地继续读盘。

与此同时，在得知请求项对应的缓冲块数据没有读完的情况下，系统将再次把读盘服务程序绑定在硬盘中断服务程序上，以待下次使用，之后中断服务程序便返回了。

执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
static void read_intr(void)
{
    ...
    if (--CURRENT->nr_sectors) {
        do_hd = &read_intr;
        return;
    }
    ...
}
```

2.3.12 硬盘中断服务程序响应后，进程 0 继续执行

进程 1 仍然被挂起着，进程 0 继续执行，与此同时，缓冲块指定的数据也正在从硬盘读出。

硬盘中断服务程序返回后，仍然在进程 0 中 pause 代码某处执行，因为刚才就是在这个函数里面打断的，这一点在 2.3.11 节中已经说明。由于这时候这两个进程还都处于非就绪状态，所以，pause 继续循环执行的时候，每次都回到进程 0 执行。整个过程如图 2-51 所示。

注意此处每次切换到进程 0 时，进程 0 仍然是从“"je 1f\n\t"”这行语句开始执行的。执行代码如下：

```
// 代码路径：init/main.c:
void main(void)
{
   ...
   for(;;) pause();
}
```

```
// 代码路径：kernel/sched.h:
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \
    "je 1f\n\t" \
    ......
    "1:" \
    ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
    "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

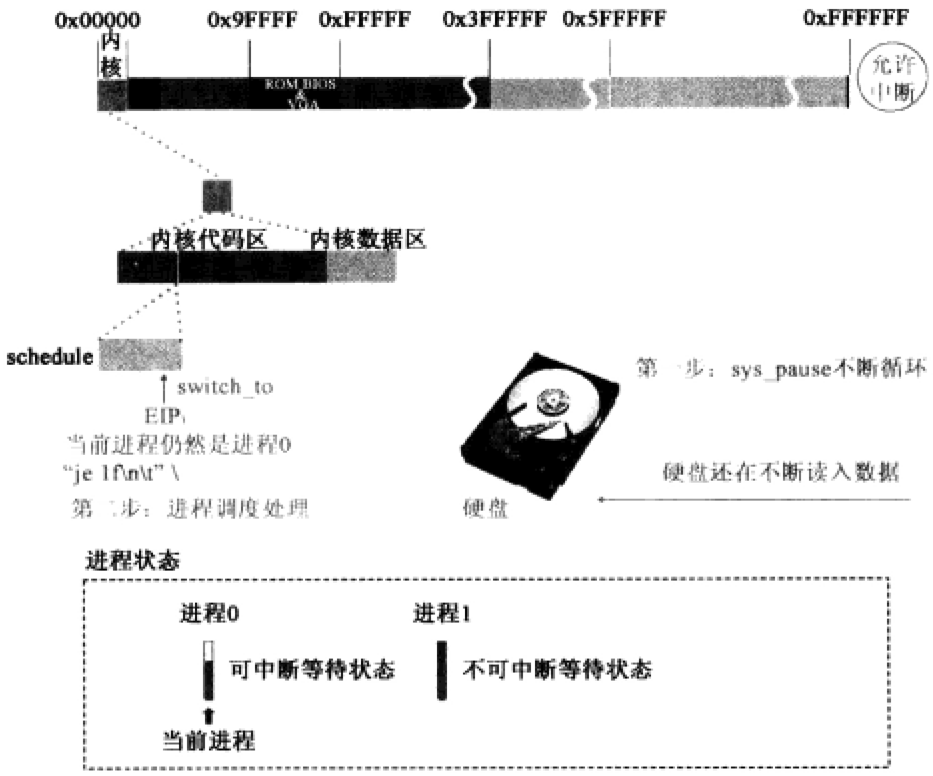


图 2-51　进程 0 继续循环执行

与此同时，硬盘继续读数据。

2.3.13　再次响应硬盘中断并唤醒进程 1

又过了一段时间后，剩下的那一半数据也读完了（要求是读 1KB 的数据，相当于两个扇区，现在都读完了），硬盘产生中断，读盘中断服务程序再次响应这个中断，进入 read_intr 函数后，仍然会判断请求项对应的缓冲块的数据是否读完了，对应代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
static void read_intr(void)
{
    ...
if (--CURRENT->nr_sectors)
    ...
}
```

这次已经将请求项要求的数据量全部读完了，经检验完成后，不执行 if 里面的内容了，跳到 end_request 函数去执行，如图 2-52 中 read_intr 这个函数所示，由于 end_request 的声明是一个内联函数，因此在编译时会编译到 read_intr 函数中。

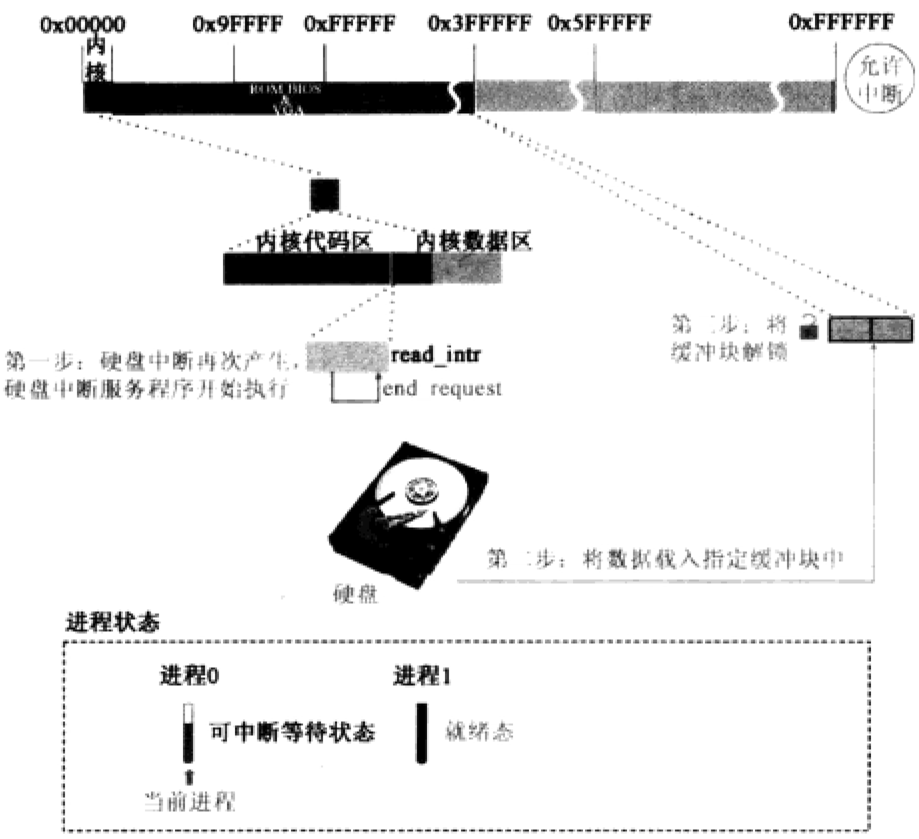


图 2-52　再次响应硬盘中断并唤醒进程 1

执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
static void read_intr(void)
{
    ...
end_request(1);
    ...
}
```

进入 end_request 后，由于此时缓冲块的内容已经全部读进来了，将这个缓冲块的更新标志置 1，说明它可用了，执行代码如下：

```
// 代码路径：kernel/blk_dev/blk.h:
extern inline void end_request(int uptodate)
{
    ...
CURRENT->bh->b_uptodate = uptodate;
    ...
}
```

之后调用 unlock_buffer 函数为缓冲块解锁，执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
extern inline void end_request(int uptodate)
{
    ...
unlock_buffer(CURRENT->bh);
    ...
}
```

最后，调用 wake_up 函数，将等待这个缓冲块解锁的进程唤醒，即，将进程 1 唤醒，并对

刚刚使用过的请求项进行处理，如将它对应的这个请求项仍旧设置为空闲请求项等，执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
extern inline void end_request(int uptodate)
{
    ...
wake_up(&CURRENT->waiting);
    ...
    CURRENT->dev = -1;
        CURRENT = CURRENT->next;
}
```

本次硬盘中断处理结束，至此，本节第一部分任务中，载入硬盘引导块的实质性工作就全部结束了，从下面开始，将介绍硬盘引导块载入后的具体应用工作。当然，硬盘中断响应结束后，仍然还是会回到进程 0 中继续执行，如图 2-52 中第二步所示。

2.3.14 读盘操作完成后，进程 1 继续执行

引导块已经载入，进程 1 已经处于就绪态，所以，进程 0 再继续循环执行 pause 时，就会切换进程 1 去执行。该过程如图 2-53 所示。

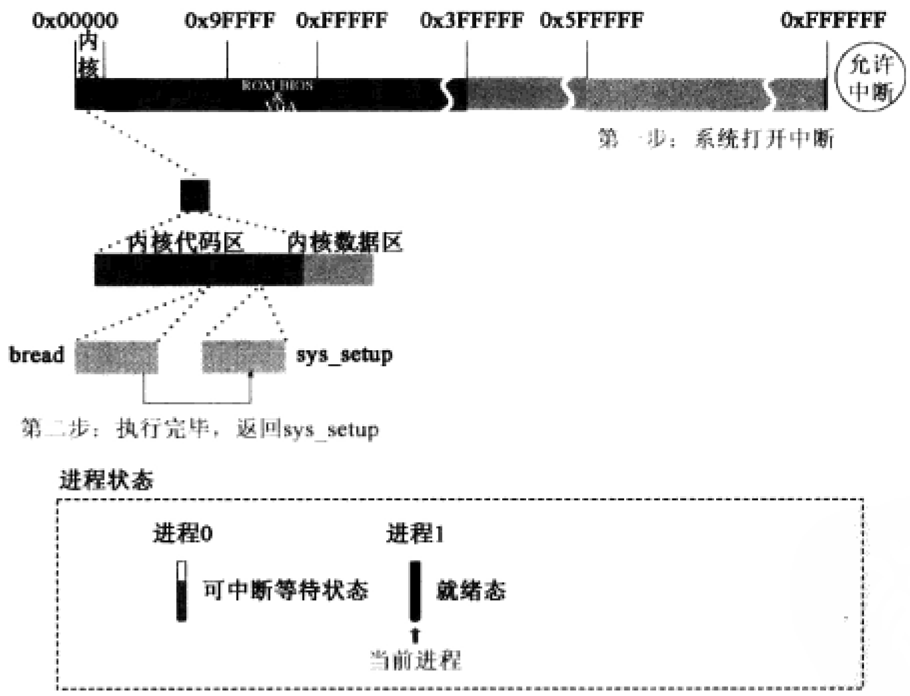


图 2-53　切换到进程 1，并返回 sys_setup

切换到进程 1 后，进程 1 从下面这个函数中的代码继续执行：

```
// 代码路径：kernel/sched.h:
#define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \
    "je 1f\n\t" \
    "movw %%dx,%1\n\t" \
    "xchgl %%ecx,_current\n\t" \
    "ljmp %0\n\t" \
```

```
        "cmpl %%ecx,_last_task_used_math\n\t" \
        "jne 1f\n\t" \
        "clts\n" \
        "1:" \
        ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
        "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

在 2.3.9 节中，进程 1 是从 "ljmp %0\n\t" \ 切换走的，所以执行下一行 "cmpl %%ecx,_last_task_used_math\n\t" \，之后返回到进程 1 切换的发起者 sleep_on 函数中，并最终返回到 bread 函数中，这一点在 2.3.8 节中已经说明了。之后，判断得知缓冲块的“已更新”标志被设置为 1，直接返回，这样，bread 函数执行完毕。执行代码如下：

```
// 代码路径：fs/buffer.c:
struct buffer_head * bread(int dev,int block)
{
    ...
    if (bh->b_uptodate)
        return bh;
    ...
}
```

进程 1 继续执行，处理硬盘引导块载入缓冲区后的一些事务。此时系统将返回到 bread 函数的调用点 sys_setup 函数中继续执行，理由在 2.3.3 节中已经说明。如图 2-53 所示。

2.3.15　进程 1 继续设置硬盘管理结构

这个缓冲块里面装载着硬盘的引导块的内容，现在开始使用它。先来判断硬盘信息有效标志‘55AA’，如果第一个扇区的最后两个字节不是‘55AA’，就说明这个扇区中的数据是无效的。在本章中，引导块的数据肯定是没有问题的，执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
int sys_setup(void * BIOS)
{
    ...
    if (bh->b_data[510] != 0x55 || (unsigned char)
            bh->b_data[511] != 0xAA)
    ...
}
```

之后，利用从引导块中采集到的分区表信息来设置硬盘分区管理结构，如图 2-54 中所示。执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
int sys_setup(void * BIOS)
{
    ...
    p = 0x1BE + (void *)bh->b_data;
        for(i=1;i<5;i++,p++) {
            hd[i+5*drive].start_sect = p->start_sect;
            hd[i+5*drive].nr_sects = p->nr_sects;
```

```
        }
        ...
}
```

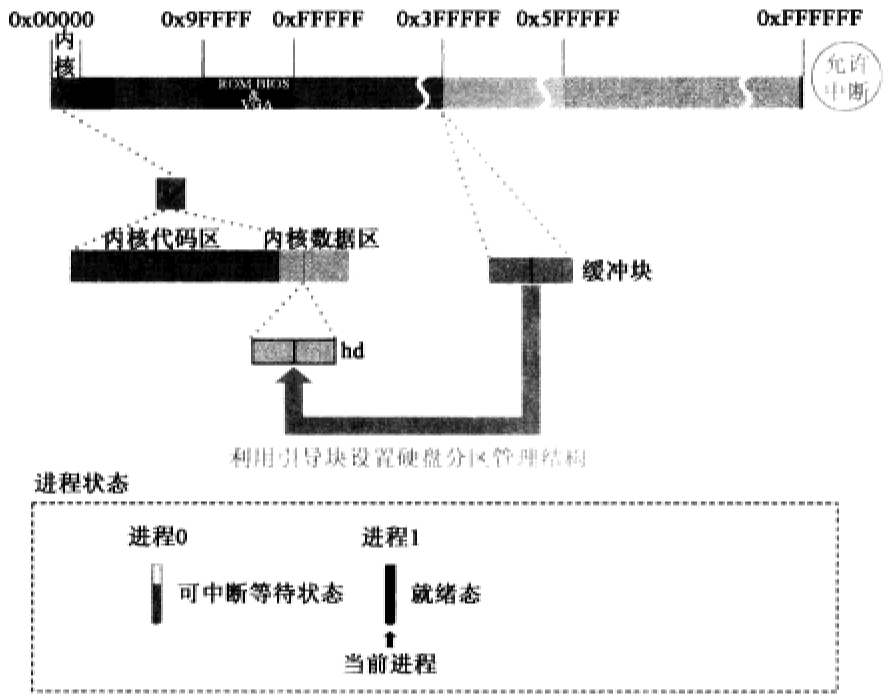


图 2-54　利用引导块设置硬盘分区管理结构

这样引导块的使命就完成了，调用 brelse 函数，立即释放其所在的缓冲块，以便以后继续使用，执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
int sys_setup(void * BIOS)
{
    ...
    brelse(bh);
    ...
}
```

到此为止，本节第一部分的内容就全部介绍完了，这部分工作的完成意味着进一步建立了进程 1 与硬盘以文件形式进行数据交互的基础。从下面开始，我们来介绍本节的第二部分内容，即进程 1 用虚拟盘替代软盘，使之成为根设备，这部分内容就是为了进程 1 最终加载根文件系统做准备的。

2.3.16　进程 1 获取软盘超级块，为加载根文件系统做准备

这里开始介绍本节第二部分的内容，进程 1 通过调用 rd_load 函数来实现，执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
int sys_setup(void * BIOS)
{
    ...
    rd_load();
    ...
}
```

进入 rd_load 函数后，通过调用 breada 函数从软盘预读取一些数据块，由于在 2.1.15 节中系

统为进程 0 以及后续进程构建了与软盘沟通的环境，因此进程 1 可以与软盘进行沟通，此函数和 bread 函数的基本功能一样，在本章 2.3.3 节至 2.3.8 节中已经说明，它们都是从块设备上读取数据块，执行代码如下：

```
// 代码路径：kernel/blk_dev/ramdisk.c:
void rd_load(void)
{
    ...
    bh = breada(ROOT_DEV,block+1,block,block+2,-1);
    ...
}
```

不同点在于可以把一些连续的数据块都读进来，一共三块，分别是：257、256 和 258，其中超级块就在 257 中，而且现在根设备是软盘，所以从软盘上读取，在读取的过程中，所发生的情况与 bread 读硬盘时基本一致，进程 1 也会被挂起，具体和本章 2.3.8 节至 2.3.13 节所描述的情况大体一致。读取完成后的状态如图 2-55 所示，可以看出 3 个连续的数据块被读入到了高速缓冲区的缓冲块中。其中，超级块用绿色框标注。

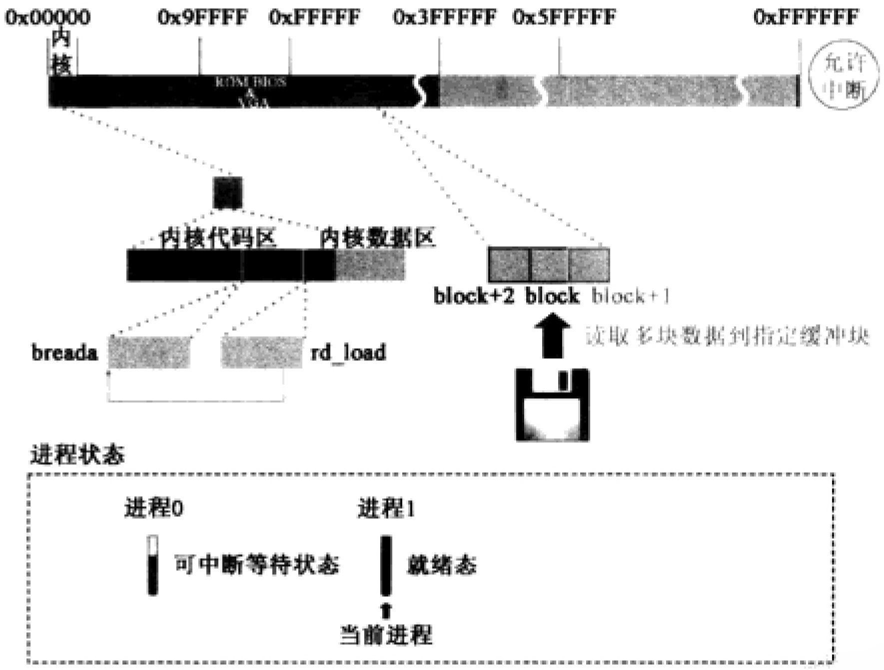


图 2-55 读取根文件系统超级块

2.3.17 进程 1 备份超级块数据

进程 1 继续完成本节第二部分的内容。此时超级块中的内容已经被载入缓冲区中，于是系统将超级块的内容进行备份，并释放掉该缓冲块，如图 2-56 中的第一步和第二步所示，图中可以看出，超级块的内容备份到了位于进程 1 管理结构所在页面的进程 1 内核栈中。执行代码如下：

```
// 代码路径：kernel/blk_dev/ramdisk.c:
void rd_load(void)
{
    ...
    *((struct d_super_block *) &s) = *((struct d_super_block *) bh->b_data);
    brelse(bh);
```

```
    ...
}
```

之后，分析超级块信息，包括判断这个文件系统是不是 minix 文件系统；包括接下来要载入的根文件系统的数据块数，会不会比整个虚拟盘区都大等，这些条件都通过，才能继续加载根文件系统，否则返回。如图 2-56 中的第三步所示。

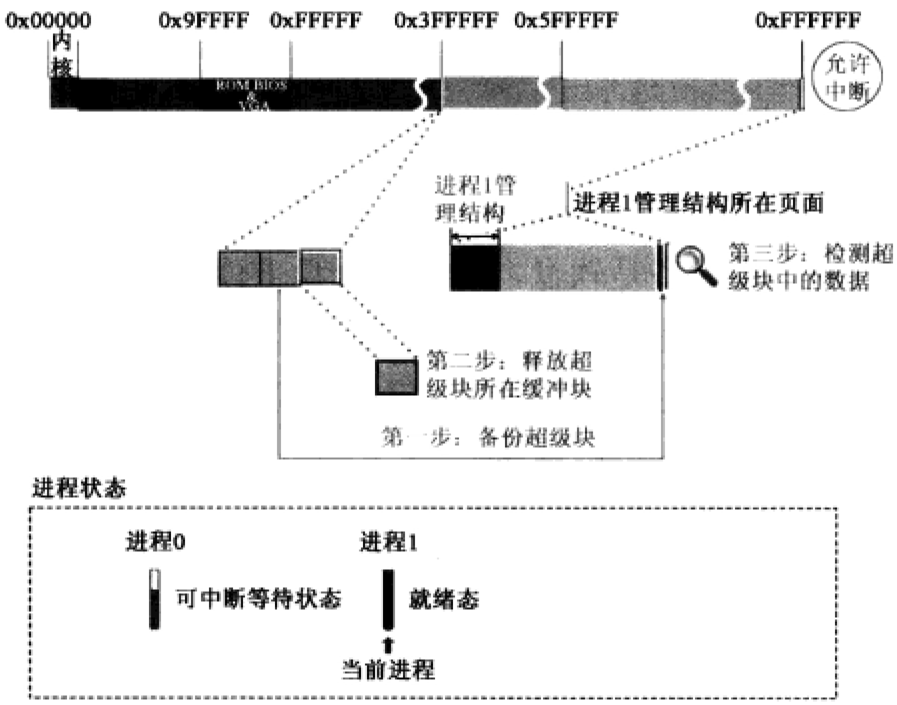


图 2-56　备份超级块并检测数据

分析超级块信息的代码如下，从代码中可以看出，在得到代表根文件系统中数据块总数的变量 nblocks 的过程中，超级块发挥了作用：

```
//代码路径：kernel/blk_dev/ramdisk.c:
void rd_load(void)
{
    ...
    if (s.s_magic != SUPER_MAGIC)
        /* No ram disk image present, assume normal floppy boot */
        return;
   nblocks = s.s_nzones << s.s_log_zone_size;
   if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
        printk("Ram disk image too big!  (%d blocks, %d avail)\n",
             nblocks, rd_length >> BLOCK_SIZE_BITS);
        return;
    }
    ...
}
```

2.3.18　进程 1 将根文件系统从软盘拷贝到虚拟盘

进程 1 继续完成本节第二部分的内容。调用 breada 函数，把与根文件系统相关的内容从软盘上拷贝到虚拟盘中。由于在 2.1.3 节中，系统为进程 0 以及后续进程构建了与虚拟盘沟通的环境，因此进程 1 可以与虚拟盘进行沟通，每次拷贝三个数据块到缓冲块，直到把规定的数据全部

拷贝进来为止，并将其备份到虚拟盘空间，然后及时释放缓冲块。这个过程如图 2-57 中的第一步和第二步所示。

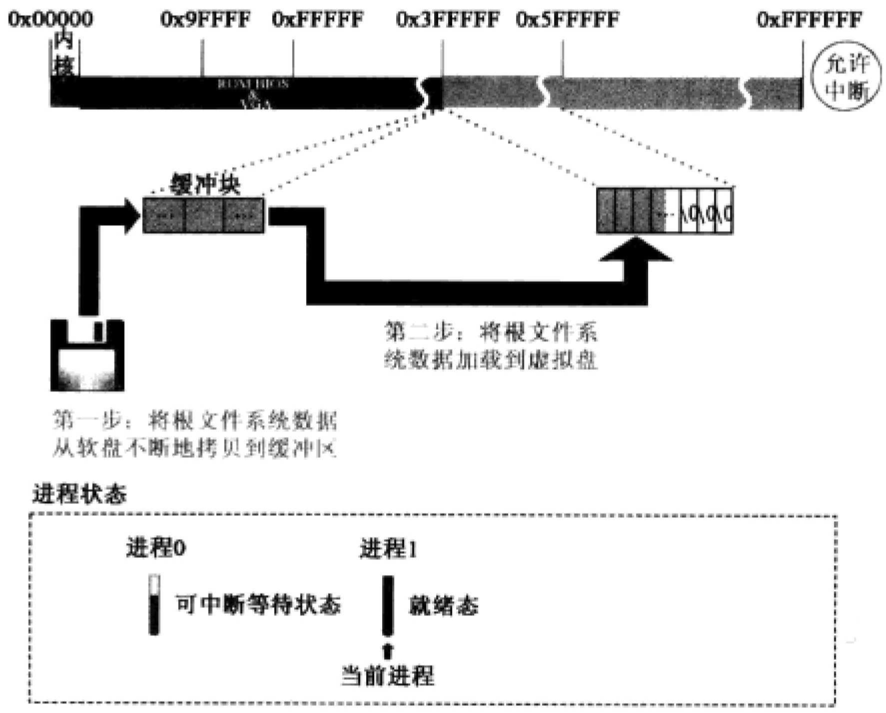


图 2-57　将根文件系统从软盘拷贝到虚拟盘

对应的代码如下：

```
// 代码路径：kernel/blk_dev/ramdisk.c:
void rd_load(void)
{
    ...
    cp = rd_start;
    while (nblocks) {
        if (nblocks > 2)
            bh = breada(ROOT_DEV, block, block+1, block+2, -1);
        else
            bh = bread(ROOT_DEV, block);
        if (!bh) {
            printk("I/O error on block %d, aborting load\n",
                block);
            return;
        }
        (void) memcpy(cp, bh->b_data, BLOCK_SIZE);
        brelse(bh);
        printk("\010\010\010\010\010%4dk",i);
        cp += BLOCK_SIZE;
        block++;
        nblocks--;
        i++;
    }
    ...
}
```

拷贝结束后，将虚拟盘设置为根设备，执行代码如下：

```
// 代码路径：kernel/blk_dev/ramdisk.c:
void rd_load(void)
```

```
{
    ...
    ROOT_DEV=0x0101;
}
```

至此，本节的第二部分内容就介绍完了，进程 1 用虚拟盘取代了软盘，使之成为根设备，下面将要介绍本节第三部分内容，即进程 1 正式加载根文件系统。

2.3.19 进程 1 开始加载根文件系统

这里开始介绍本节第三部分的内容，进程 1 要通过调用 mount_root 函数得以实现，这部分中涉及的超级块、逻辑块位图、i 节点位图、根 i 节点等概念将在本书以后的文件操作章节中详细介绍，执行代码如下：

```
// 代码路径：kernel/blk_dev/hd.c:
   int sys_setup(void * BIOS)
{
    ...
    mount_root();
    ...
}
```

进入 mount_root 函数后，先对文件管理表 file_table 进行初始化，将来系统中所有进程操作的文件都要在这个表中登记，因此要将该表中所有表项的"被引用次数"设置为 0，使之全部成为空闲项以便以后使用，如图 2-58 中的第一步所示。

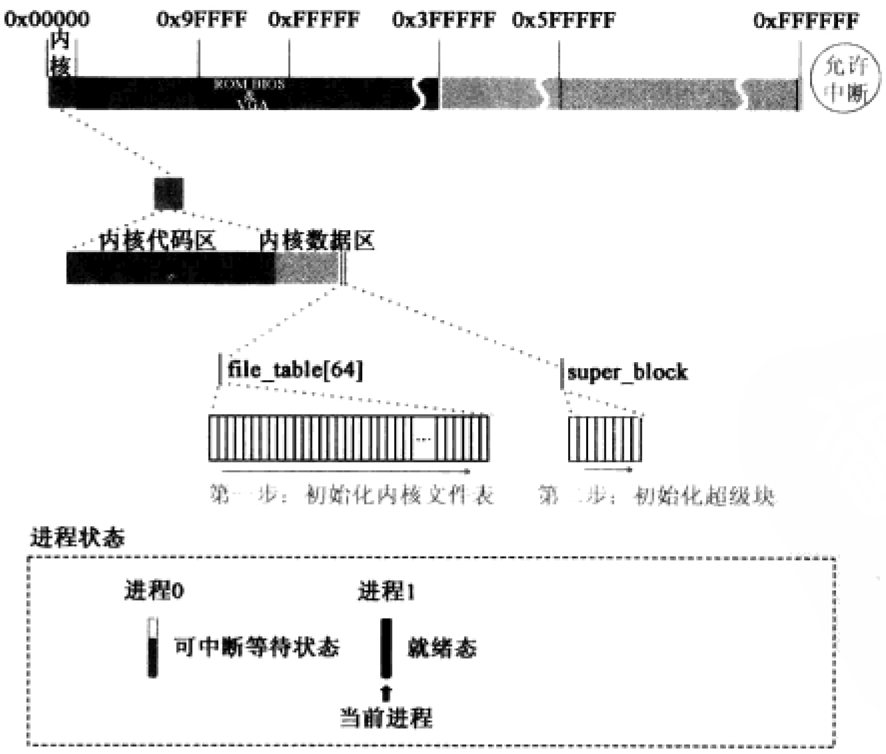


图 2-58 初始化内核文件表和超级块

执行代码如下：

```
// 代码路径：fs/super.c:
void mount_root(void)
{
```

```
    ...
    for(i=0;i<NR_FILE;i++)
        file_table[i].f_count=0;
    ...
}
```

之后，初始化内存中的超级块管理结构 super_block，这个结构一共有 8 项，这里要将每一项所对应的设备号加锁标志和等待该设备解锁的进程标志，全部设置为 0，如图 2-58 中的第二步所示。系统只要想和任何一个设备以文件的形式进行数据交互，都要将这个设备的超级块存储在 super_block 中，这样才能获取这个设备中文件系统的最基本信息，根设备中的超级块也不例外，执行代码如下：

```
// 代码路径：fs/super.c:
 void mount_root(void)
{
    ...
    for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {
            p->s_dev = 0;
            p->s_lock = 0;
            p->s_wait = NULL;
        }
    ...
}
```

2.3.20　进程 1 准备加载根文件系统超级块

进程 1 继续完成本节第三部分的内容，接下来调用 read_super 函数，从虚拟盘中读取根文件系统的超级块并加载，执行代码如下：

```
// 代码路径：fs/super.c:
void mount_root(void)
{
     ...
     if (!(p=read_super(ROOT_DEV)))
     ...
}
```

进入 read_super 函数后，先检测这个超级块是不是已经被读进 super_block 中了，如果已经被读进来了，则直接使用，这是因为，由于内存超级块管理结构一共有 8 项（这一点在本章 2.3.19 节中已经说明），系统中最多可以加载 8 个设备的文件系统，如果事先已经针对某个设备的文件系统加载过了，就不用再加载一次了，这与本章 2.3.3 节中先通过哈希表来检测缓冲块是否已经存在的道理是一样的，执行代码如下：

```
// 代码路径：fs/super.c:
static struct super_block * read_super(int dev)
{
     ...
     if (s = get_super(dev)) return s;
     ...
}
```

当然，此时加载的是根文件系统，而之前还没有过任何加载，所以，要在 super_block 这个超级块控制结构中申请一个新的空间，从图 2-59 中可以看出，此时找到的是 super_block 结构中的第一项。然后进行初始化并加锁，准备把超级块读出来。

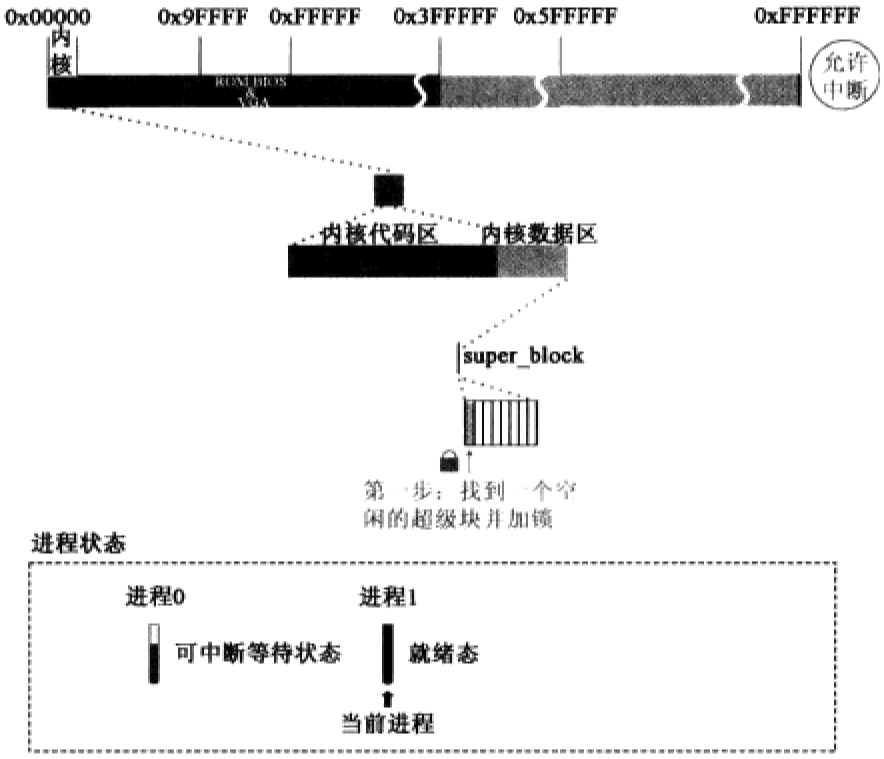


图 2-59　加载根文件系统超级块

对应的代码如下：

```
// 代码路径：fs/super.c:
static struct super_block * read_super(int dev)
{
    ...
    for (s = 0+super_block ;; s++) {
        if (s >= NR_SUPER+super_block)
            return NULL;
        if (!s->s_dev)
            break;
        }
        s->s_dev = dev;
        s->s_isup = NULL;
        s->s_imount = NULL;
    s->s_time = 0;
    s->s_rd_only = 0;
    s->s_dirt = 0;
        lock_super(s);
    ...
}
```

2.3.21　进程 1 加载根文件系统超级块

进程 1 继续完成本节第三部分的内容，仍旧执行 read_super 函数。接下来开始调用 bread 函数，把超级块从虚拟盘上读进缓冲区，系统在本章的 2.1.3 节中对虚拟盘的设置以及在 2.1.13 节中对缓冲区的设置将在这里继续发挥作用，执行代码如下：

```
// 代码路径：fs/super.c:
static struct super_block * read_super(int dev)
{
      ...
      if (!(bh = bread(dev,1)))
      ...
}
```

bread 函数在本章的 2.3.3 节至 2.3.8 节中已经说明，但此处也有一点区别，在 2.1.14 节中提到，如果给硬盘发送操作命令，则调用 do_hd_request 函数，但此次操作的是虚拟盘，所以要调用 do_rd_request 函数，理由在 2.1.3 节中已经说明。值得注意的是：虚拟盘虽然被视为外设，但它毕竟就是内存里面一段空间，并不是实际的外设，所以，调用 do_rd_request 函数从虚拟盘上读取超级块，并不会发生类似硬盘中断的情况。以上动作的效果如图 2-60 的第一步所示。

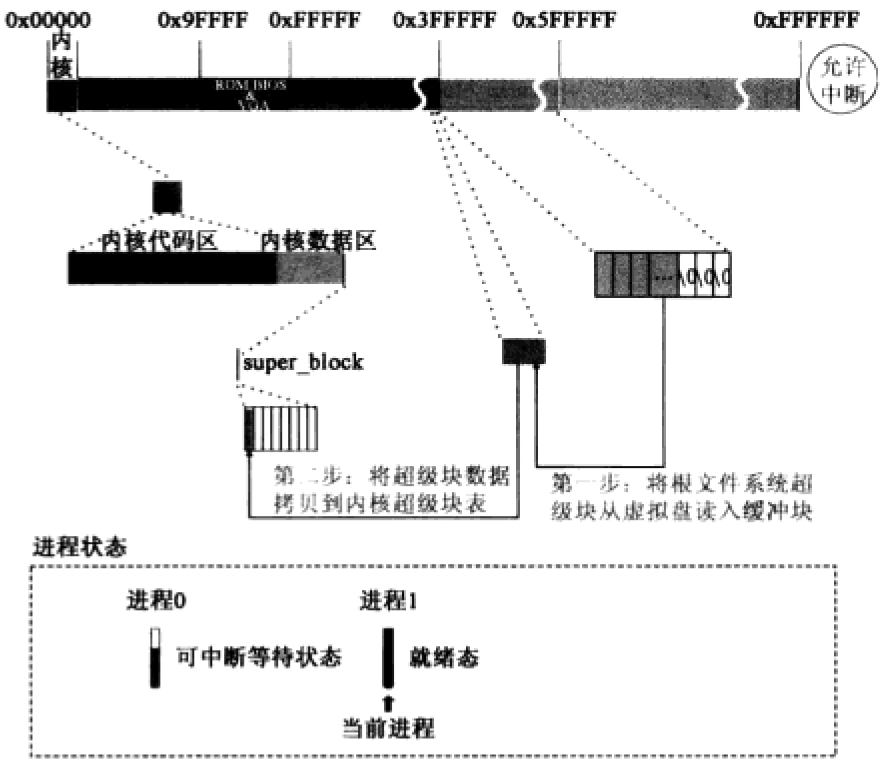


图 2-60 从虚拟盘读取超级块并拷贝到内核超级块表

超级块拷贝进缓冲块以后，将这个超级块的数据加载进 super_block 这个超级块结构中，如图 2-60 中的第二步所示。并调用 brelse 函数释放缓冲块，之后，判断读进来的这个超级块中魔数 (SUPER_MAGIC) 信息是否正确，执行代码如下：

```
// 代码路径：fs/super.c:
static struct super_block * read_super(int dev)
{
     ...
     *((struct d_super_block *) s) =
         *((struct d_super_block *) bh->b_data);
         brelse(bh);
         if (s->s_magic != SUPER_MAGIC)
...
}
```

2.3.22 进程 1 继续加载根文件系统

进程 1 继续完成本节第三部分的内容，仍旧执行 read_super 函数。备份的超级块信息开始发挥作用了。系统通过它提供的信息，设置 i 节点位图管理结构 s_imap 和逻辑位图管理结构 s_zmap，这两个成员都是将来管理设备上文件的重要数据信息，它们都是绑定在 super_block 这个结构上面的成员，因此，也要调用 bread 函数到虚拟盘上去读取。由于对它们的操作会比较频繁，于是这些占用的缓冲块并不被释放，以后它们将常驻在缓冲区内。如图 2-61 所示，超级块通过指针与 s_imap 和 s_zmap 实现挂接。

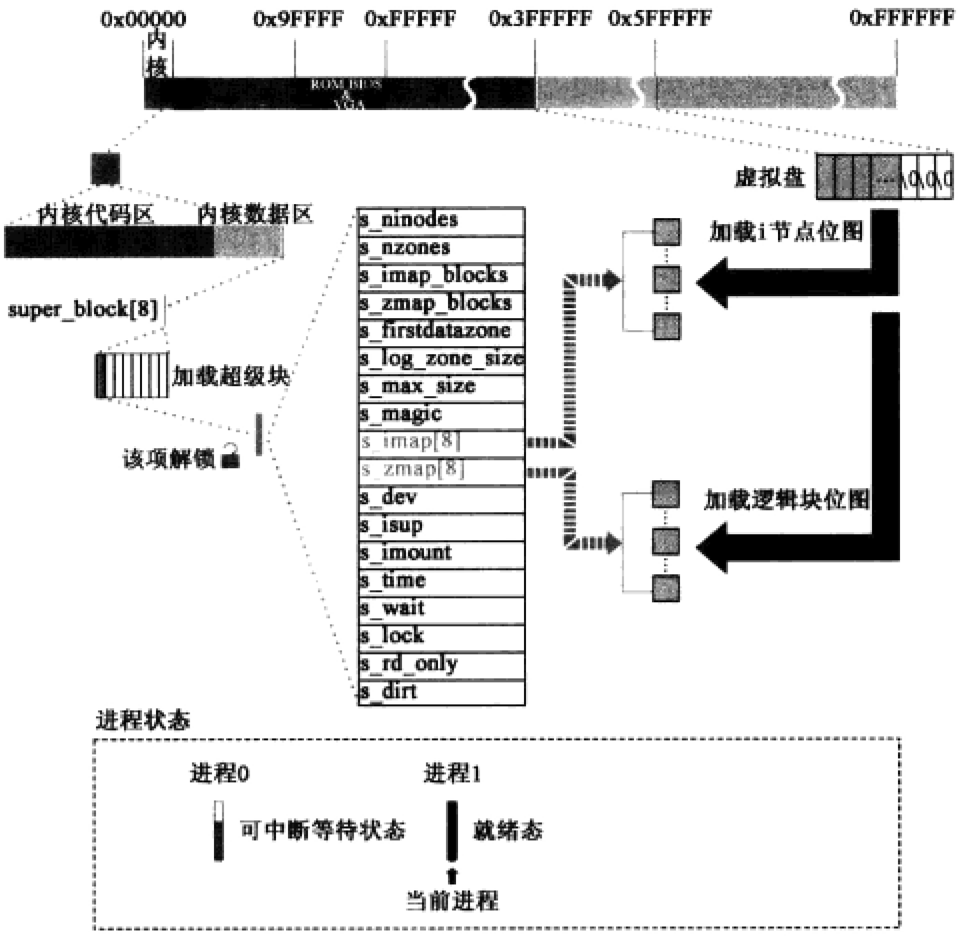


图 2-61 读取逻辑块为图和 i 节点位图

到此，超级块的操作完毕，于是将超级块解锁，函数返回，执行代码如下：

```
// 代码路径：fs/super.c:
static struct super_block * read_super(int dev)
{
    ...
    for (i=0;i<I_MAP_SLOTS;i++)
        s->s_imap[i] = NULL;
    for (i=0;i<Z_MAP_SLOTS;i++)
        s->s_zmap[i] = NULL;
    block=2;
    for (i=0 ; i < s->s_imap_blocks ; i++)
        if (s->s_imap[i]=bread(dev,block))
            block++;
        else
            break;
```

```
    for (i=0 ; i < s->s_zmap_blocks ; i++)
        if (s->s_zmap[i]=bread(dev,block))
            block++;
        else
            break;
...
s->s_imap[0]->b_data[0] |= 1;
    s->s_zmap[0]->b_data[0] |= 1;
    free_super(s);
    return s;
}
```

2.3.23　进程 1 准备读取根目录 i 节点

进程 1 继续完成本节第三部分的内容，回到 mount_root 函数继续执行，接下来调用 iget 函数，从虚拟盘上读取根 i 节点，它的意义在于：只要将来设备与系统建立了文件交互关系，都可以通过这个根 i 节点来找到指定的文件。

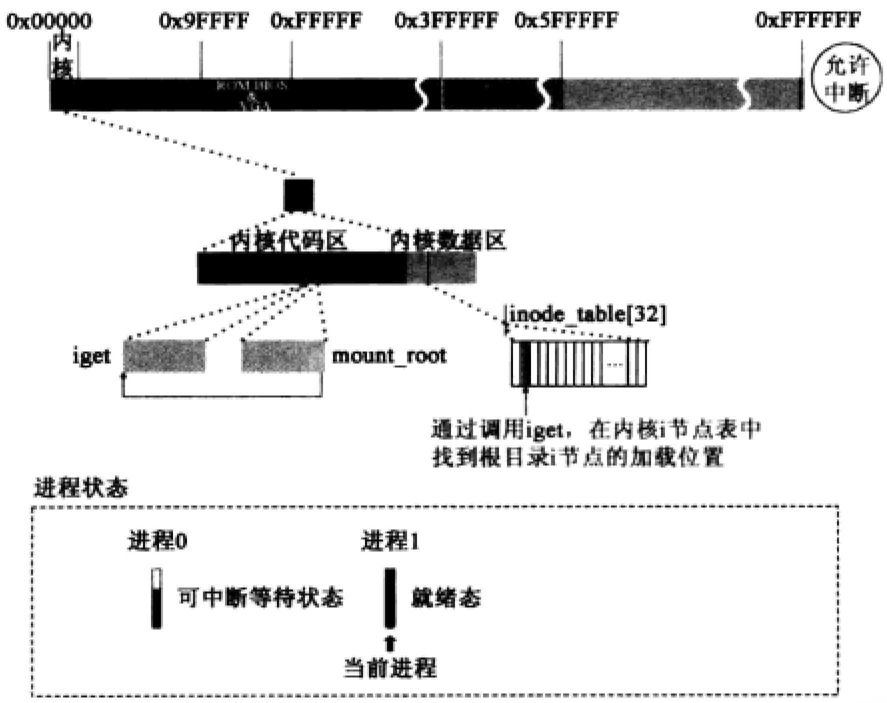


图 2-62　读取根目录 i 节点

执行代码如下：

```
//代码路径：fs/super.c:
void mount_root(void)
{
    ...
    if (!(mi=iget(ROOT_DEV,ROOT_INO)))
    ...
}
```

进入 iget 函数后，系统从 i 节点表 inode_table 中申请一个空闲的 i 节点位置，此时应该是首个 i 节点，图 2-62 中给出了根目录 i 节点在内核 i 节点表中的位置。

对应代码如下：

```
// 代码路径：fs/inode.c:
struct m_inode * iget(int dev,int nr)
{
    ...
    empty = get_empty_inode();
    ...
}
```

之后对这个 i 节点进行初始化设置，其中包括该 i 节点对应的设备号、该 i 节点的节点号等，执行代码如下：

```
// 代码路径：fs/inode.c:
struct m_inode * iget(int dev,int nr)
{
    ...
    inode=empty;
      inode->i_dev = dev;
      inode->i_num = nr;
    ...
}
```

2.3.24 进程 1 加载根目录 i 节点

进程 1 继续完成本节第三部分的内容，仍旧在 iget 函数中执行，接下来调用 read_inode 这个函数，把 i 节点从虚拟盘上读出来。

执行代码如下：

```
// 代码路径：fs/inode.c:
struct m_inode * iget(int dev,int nr)
{
    ...
    read_inode(inode);
    ...
}
```

进入 read_inode 函数后，先给这个 i 节点加锁，这样在解锁之前，这个 i 节点就不能被另行占用了，之后，通过该 i 节点所在的超级块，来间接地计算出 i 节点所在的逻辑块号，并将 i 节点所在的逻辑块，整个地读出来，从中提取这个 i 节点的信息，将之载入到 i 节点管理表 m_inode 中由本章 2.3.23 节申请到的 i 节点位置上，如图 2-63 所示，注意 inode_table 中的变化。最后，释放缓冲块并将锁定的 i 节点解锁，iget 函数到此执行完毕了，执行的代码如下：

```
// 代码路径：fs/inode.c:
static void read_inode(struct m_inode * inode)/* 不是接口 */
{
...
    lock_inode(inode);
```

```
    if (!(sb=get_super(inode->i_dev)))
        ...
    block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
        (inode->i_num-1)/INODES_PER_BLOCK;
    if (!(bh=bread(inode->i_dev,block)))
        ...
    *(struct d_inode *)inode =
        ((struct d_inode *)bh->b_data)
            [(inode->i_num-1)%INODES_PER_BLOCK];
    brelse(bh);
    unlock_inode(inode);
}
```

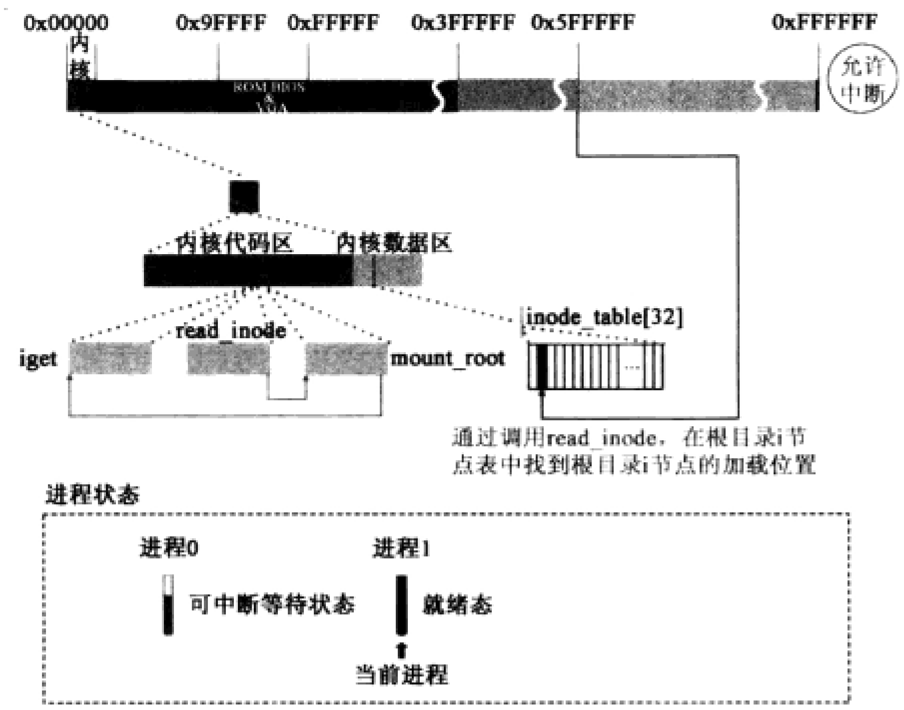


图 2-63　读取 i 节点

2.3.25　进程 1 结束加载根文件系统的过程

进程 1 继续完成本节第三部分的内容，现在针对根 i 节点的操作已经完成了，进程 1 将回到 mount_root 函数中继续执行。接下来，将根文件系统的超级块中“被安装文件系统 i 节点 s_isup”和“被安装到的 i 节点 s_imount”都设置为这个根 i 节点，这样，将来系统在根设备上寻找文件时，就可以通过这里建立的关系，一步步的把文件找到。然后，再对进程 1 管理结构中与文件系统 i 节点有关的字段进行设置，将来对根文件系统中存储的文件操作，就需要以此为起点。如图 2-64 所示。

执行代码如下：

```
// 代码路径：fs/super.c:
void mount_root(void)
{
    ...
    p->s_isup = p->s_imount = mi;
    current->pwd = mi;
```

```
    current->root = mi;
    ...
}
```

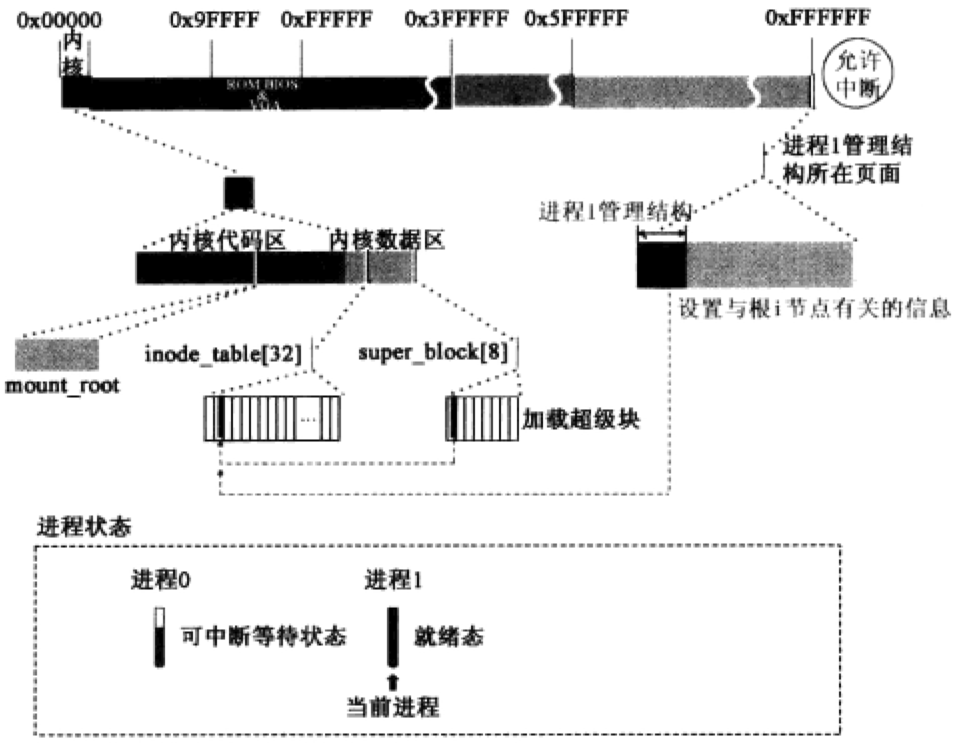


图 2-64　加载根文件系统完成并返回

得到了根文件系统的超级块，就可以根据超级块中“逻辑位图”里记载的信息，计算出虚拟盘上数据块的占用与空闲情况，并将此信息记录在本章 2.3.22 节中提到的“装载逻辑位图信息的缓冲块中”，执行代码如下：

```
// 代码路径：fs/super.c:
void mount_root(void)
{
    ...
    while (-- i >= 0)
        if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))
    ...
}
```

然后，再根据超级块中“i 节点位图”里记载的信息，计算出虚拟盘上 i 节点的占用与空闲的情况，并将此信息记录在本章 2.3.22 节中提到的“装载 i 节点位图信息的缓冲块中”，执行代码如下：

```
// 代码路径：fs/super.c:
void mount_root(void)
{
    ...
    while (-- i >= 0)
        if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))
    ...
}
```

到此为止，sys_setup 函数就全都执行完毕了。由于这个函数也是由于产生软中断才被调用的，所以返回到 system_call 中执行，之后会执行 ret_from_sys_call:，这时候的当前进程是进程 1

了，所以下面将调用 do_signal 函数，（只要当前进程不是进程 0，都要执行到这里），来对当前进程的信号位图进行检测，执行代码如下：

```
// 代码路径：kernel/system_call.s:
ret_from_sys_call:
    movl _current,%eax
    cmpl _task,%eax
    je 3f
    cmpw $0x0f,CS(%esp)
    jne 3f
    cmpw $0x17,OLDSS(%esp)
    jne 3f
    movl signal(%eax),%ebx
    movl blocked(%eax),%ecx
    notl %ecx
    andl %ebx,%ecx
    bsfl %ecx,%ecx
    je 3f
    btrl %ecx,%ebx
    movl %ebx,signal(%eax)
    incl %ecx
    pushl %ecx
    call _do_signal
```

当然，当前进程（进程 1）并没有接收到信号，所以调用 do_signal 函数并没有实际的意义，这样，此次关于 sys_setup 的系统调用就结束了，进程 1 又将返回到本章 2.3.2 节中讲到的代码的调用点，准备下面的执行。

```
// 代码路径：init/main.c:
void init(void)
{
    ...
    setup((void *) &drive_info);
    ...
}
```

到此为止，本节第三部分的内容就介绍完了，这标志着：本章第二阶段的核心任务，即，“进程 1 与外设能够以文件的形式进行沟通”这个任务，已经彻底完成了，在下面的章节中，进程 1 将继续执行，在具备了以文件形式与外设沟通的基础后，打开一个终端设备文件并复制该文件句柄，以此继续完成第二阶段中其他的一些任务。

2.4　打开终端设备文件及复制文件句柄

在 2.3 节中，进程 1 成功加载了根文件系统，这就为它继续完成本节所介绍的任务奠定了基础。本节将在此基础上，通过进程 1 打开一个指定的终端设备文件，并复制此文件的句柄，这些工作将为以后终端控制台的正常运行做准备。

打开一个文件及复制文件句柄的本质都是让当前进程与指定的文件建立可操作的关系。在当

前进程的管理结构中，*filp[20] 这个结构将记录这个关系。如果进程 1 记录了这个关系，将来它创建进程 2 的时候，这个关系也会被复制过去，就像 2.2 节中将进程 0 这个母本中的信息复制给进程 1 一样。这就导致进程 2 也将能够与这个终端设备文件保持可操作关系，于是进程 2 也能够与终端控制台建立关系了。

可见，进程 1 除了具备与外设以文件形式进行沟通的能力之外，还为进程 2“预定”了一个必然要用到的文件，即终端设备文件，这就减轻了进程 2 将来的工作压力。

本节首次涉及了文件打开操作，这里介绍的文件打开步骤与操作系统中所有文件打开的步骤基本一致，因此，掌握本节的内容将会对以后系统地理解文件操作打下基础。

本节执行的程序之架构图如图 2-65 所示。

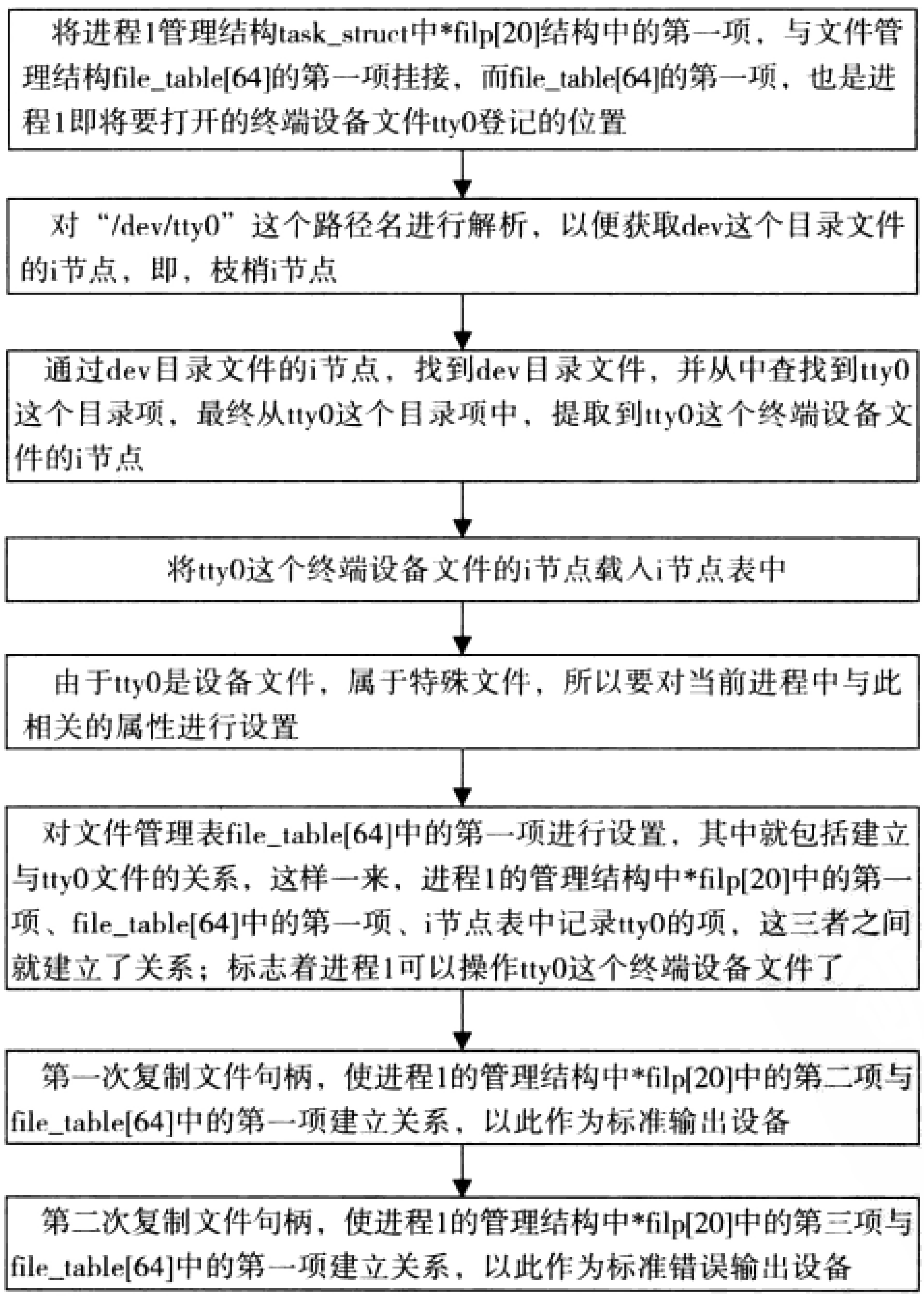


图 2-65　进程 1 打开“/dev/tty0”文件并复制文件句柄的架构图

tty0 文件加载后，就形成了如图 2-66 所示的效果图，将来就可以通过图中指示的路线找到 tty0 文件并对它执行操作。

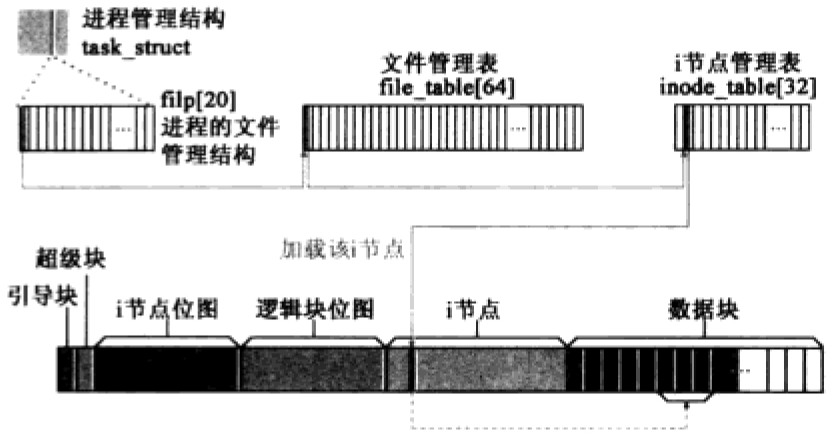


图 2-66 打开 tty0 文件后，相关信息在内存和进程中的分布图

2.4.1 进程 1 与内核文件表挂接，为打开文件做准备

进程 1 开始打开终端设备文件。它将通过调用 open 函数来实现此目的，执行代码如下：

```
//代码路径：init/main.c:
void init(void)
{
    ...
    (void) open("/dev/tty0",O_RDWR,0);
    ...
}
```

其中实参 "/dev/tty0" 表明了这个终端设备文件在根设备上的位置。

这个函数执行后，也会产生软中断，并最终映射到 sys_open 这个函数中去执行，具体的映射过程与本章 2.2.1 节中介绍的 fork 函数映射到 sys_fork 的方式大体一致。

进入 sys_open 函数中执行后，先在进程 1 的管理结构中的"文件打开控制结构 *filp[20]"里申请一个空闲项，执行代码如下：

```
//代码路径：fs/open.c:
int sys_open(const char * filename,int flag,int mode)
{
    ...
    for(fd=0 ; fd<NR_OPEN ; fd++)
        if (!current->filp[fd])
            break;
    ...
}
```

之后，在系统中的文件管理结构 file_table 中也为该文件申请一个空闲项，执行代码如下：

```
//代码路径：fs/open.c:
int sys_open(const char * filename,int flag,int mode)
{
    ...
    f=0+file_table;
    for (i=0 ; i<NR_FILE ; i++,f++)
        if (!f->f_count) break;
    ...
}
```

最后，把两者进行挂接，并将文件管理结构 file_table 中的文件引用次数加 1，表明这个位置所对应的文件已经被使用一次了，执行代码如下：

```
// 代码路径：fs/open.c:
int sys_open(const char * filename,int flag,int mode)
{
    ...
    (current->filp[fd]=f)->f_count++;
    ...
}
```

这样进程 1 就与文件管理结构 file_table 建立了关系，将来 tty0 文件成功打开以后，将登记在这个在文件管理结构 file_table 中与进程挂接的空闲项上，这样进程 1 就具备了操作 tty0 文件的能力了。进程 1 与内核 file_table 结构中空闲项的挂接状态如图 2-67 所示。

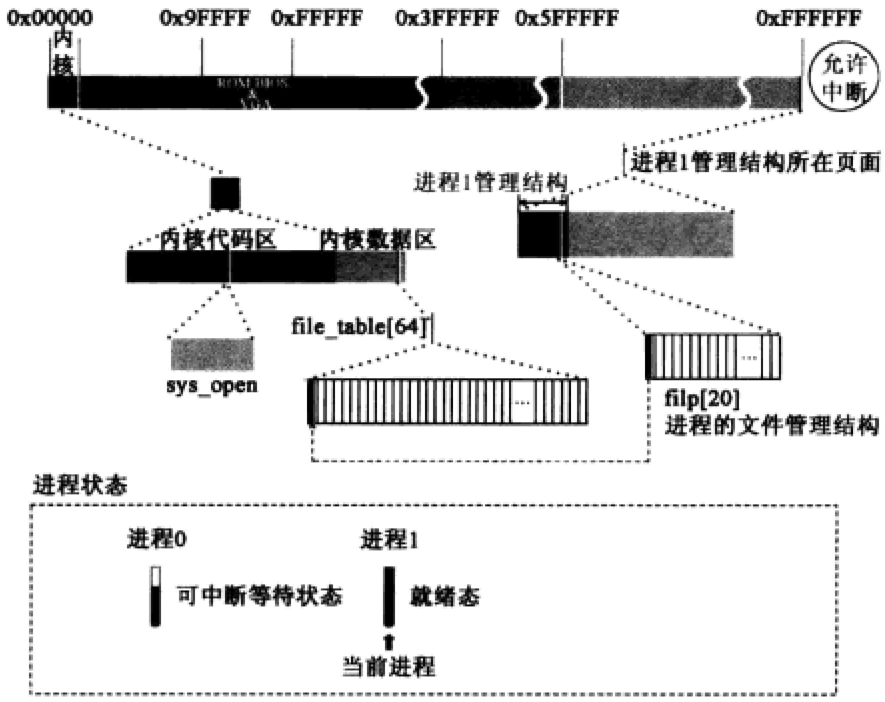


图 2-67　打开终端设备文件的准备工作

接下来就要调用 sys_open 中的主体 open_namei 函数了，开始实质性的打开文件操作，执行代码如下：

```
// 代码路径：fs/open.c:
int sys_open(const char * filename,int flag,int mode)
{
    ...
    if ((i=open_namei(filename,flag,mode,&inode))<0)
    ...
}
```

这个函数的主要目的就是找到这个终端设备文件的 i 节点，因为找到了这个文件的 i 节点，就可以找到这个文件的全部内容，关于 i 节点的具体内容，将在本书后续与文件操作相关的章节中详细介绍。

2.4.2　确定打开操作的起点

进程 1 执行本节打开终端设备文件的工作。在进入 open_namei 函数后，先调用 dir_namei 函数，对路径名进行解析，找到“枝梢 i 节点”。对于“/dev/tty0”这个路径名而言，找到枝梢 i 节点就是要找到 dev 这个目录文件的 i 节点，因为 tty0 这个目录项就在 dev 这个目录文件里面（目录文件与普通文件有所区别，它里面装载的信息都是一个一个的目录项，这些目录项，都将作为查找指定文件的索引信息），所以这一步是为下面进而找到 tty0 这个目录项做准备的，执行代码如下：

```
//代码路径：fs/namei.c:
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    if (!(dir = dir_namei(pathname,&namelen,&basename)))
    ...
}
```

进入 dir_namei 函数后，调用 get_dir 函数，准备针对 "/dev/tty0" 这个路径名进行具体的解析操作，并最终获得 dev 目录文件的 i 节点，执行代码如下：

```
//代码路径：fs/namei.c:
static struct m_inode * dir_namei(const char * pathname,
    int * namelen, const char ** name)
{
    ...
    if (!(dir = get_dir(pathname)))
    ...
}
```

进入 get_dir 函数后，先检测这个进程在根 i 节点和当前目录指针方面的设置，看看有没有问题，这些确定以后，才能解析，执行代码如下：

```
//代码路径：fs/namei.c:
static struct m_inode * get_dir(const char * pathname)
{
    ...
    if (!current->root || !current->root->i_count)
    ...
    if (!current->pwd || !current->pwd->i_count)
    ...
}
```

之后，通过分析得知，“/dev/tty0”这个路径名的第一个字符是不是 '/'，当前情况下，肯定是 '/'，这就说明，这个路径名是绝对路径名，所以从根 i 节点开始查找操作，然后获取当前进程的根 i 节点，这个根 i 节点，就是我们在本章 2.3.23 节至 2.3.24 节中从虚拟盘上获取的那个根 i 节点，现在它开始发挥作用了，该过程如图 2-68 所示。

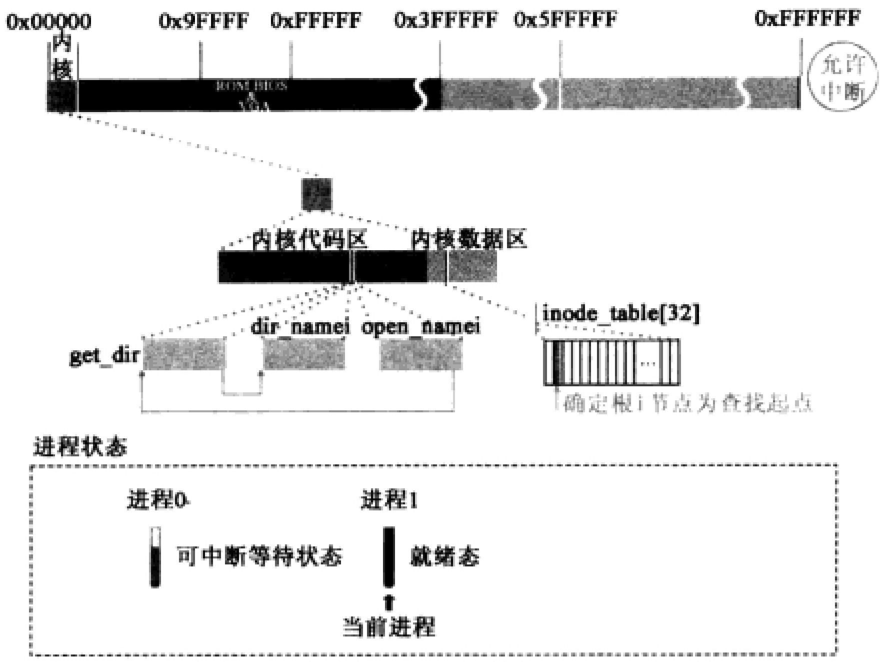


图 2-68　文件名解析准备工作

执行代码如下：

```
// 代码路径：fs/namei.c:
static struct m_inode * get_dir(const char * pathname)
{
    ...
    if ((c=get_fs_byte(pathname))=='/') {
        inode = current->root;
        pathname++;
    }
    ...
}
```

2.4.3 获得枝梢 i 节点——dev 目录文件的 i 节点

进程 1 执行本节打开终端设备文件的工作。在获取了根 i 节点后，将开始解析 "/dev/tty0" 这个路径名，具体步骤是这样的：首先对 dev 这个目录项进行分析，要在虚拟盘上找到这个目录项所在的逻辑块，并读进缓冲区中，如图 2-69 的第一步所示，执行代码如下：

```
// 代码路径：fs/namei.c:
static struct m_inode * get_dir(const char * pathname)
{
    ...
    for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
    ...
    if (!(bh = find_entry(&inode,thisname,namelen,&de)))
    ...
}
```

现在的实际情况是，dev 这个目录项确实存在于根 i 节点所对应的目录文件中，所以一定能被找到，值得注意的是，目录项结构是这样声明的，代码如下：

```
struct dir_entry {
    unsigned short inode;
```

```
    char name[NAME_LEN];
};
```

struct dir_entry 这个结构总共 16 个字节，其中 inode 代表这个目录项所对应的目录文件在硬盘上的 i 节点号，它占 2 个字节；name[NAME_LEN] 是这个目录项的名字，占 14 个字节。

所以，通过被载入缓冲区的 dev 目录项，就可以找到它所对应目录文件的 i 节点号，另外，再根据此时已知的 i 节点（现在是根 i 节点）又能提取虚拟盘设备的设备号，这样，就可以把 dev 目录中记录的关于 dev 目录文件的 i 节点，从虚拟盘中提取出来，并用这个提取出来的 i 节点来覆盖已经获取的根 i 节点，这个 i 节点，将作为下一次路径解析的指引。如图 2-69 中的第二步所示。

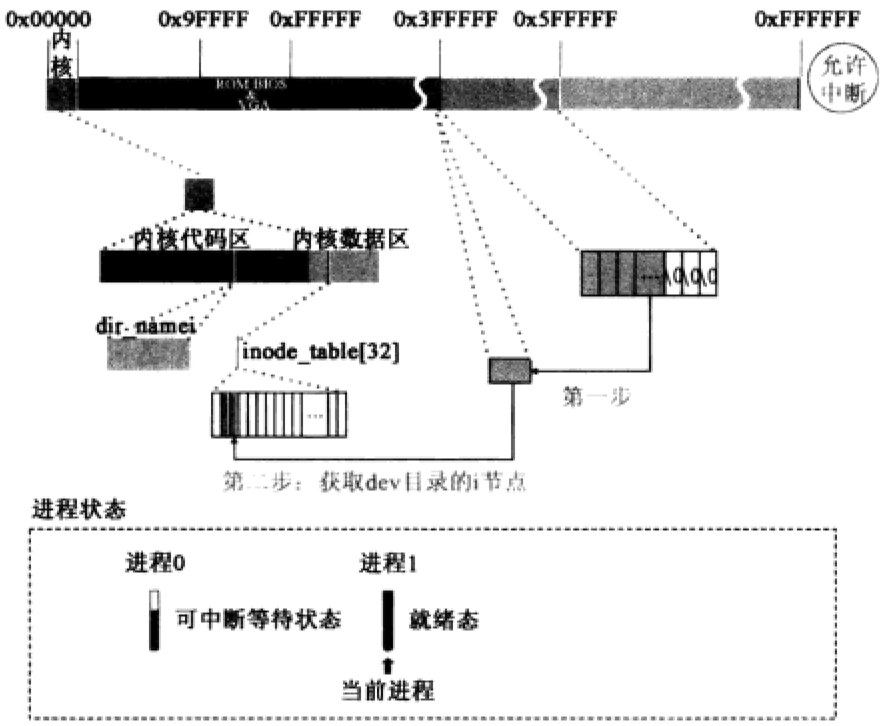


图 2-69 获得 dev 的 i 节点

执行代码如下：

```
// 代码路径：fs/namei.c:
static struct m_inode * get_dir(const char * pathname)
{
    ...
    inr = de->inode;
      idev = inode->i_dev;
    ...
    if (!(inode = iget(idev,inr)))
    ...
}
```

2.4.4 确定 dev 目录文件 i 节点为枝梢 i 节点

进程 1 继续执行本节打开终端设备文件的工作。再次分析路径名，这一次是轮到 tty0 了。解析后发现，它的最后一个字符是空的，所以 if (!c) 的执行结果为真，这就证明 dev 目录文件的 i 节点就是枝梢 i 节点（枝梢 i 节点就是准备查找的目标 i 节点前的最后一个 i 节点），已经找到了

在 2.4.3 节中提到的 dev 目录文件的 i 节点，它也是用来覆盖根 i 节点的那个 i 节点，就是本函数要找的枝梢 i 节点，于是系统将这个 i 节点返回，返回的方式是通过进程 1 的内核栈。

执行代码如下：

```
// 代码路径：fs/namei.c:
static struct m_inode * get_dir(const char * pathname)
{
    ...
    for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
    if (!c)
            return inode;
    ...
}
```

get_dir 这个函数就执行完毕，并最终返回到 dir_namei 中去继续执行。

2.4.5 继续返回枝梢 i 节点

进程 1 执行本节打开终端设备文件的工作。返回到 dir_namei 函数后，针对“/dev/tty0”这个路径名测算出 dev 之后所有路径的字符串长度，以及 dev 之后的剩余路径名。在现在的情况下，剩余的肯定是关于 tty0 的路径名了，并将获取的枝梢 i 节点返回到 open_namei 函数中。

执行代码如下：

```
// 代码路径：fs/namei.c:
static struct m_inode * dir_namei(const char * pathname,
    int * namelen, const char ** name)
{
    basename = pathname;
    while (c=get_fs_byte(pathname++))
        if (c=='/')
            basename=pathname;
    *namelen = pathname-basename-1;
    *name = basename;
    return dir;
}
```

2.4.6 查找 tty0 文件的 i 节点

进程 1 执行本节打开终端设备文件的工作。返回到 open_namei 函数后，通过在本章 2.4.5 节中获取到的枝梢 i 节点、剩余路径名，以及剩余路径名字符串的长度，从虚拟盘中把 dev 这个目录文件中的 tty0 这个目录项所在的逻辑块载入缓冲区，如图 2-70 中的第一步所示。执行代码如下：

```
// 代码路径：fs/namei.c:
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
bh = find_entry(&dir,basename,namelen,&de);
    ...
}
```

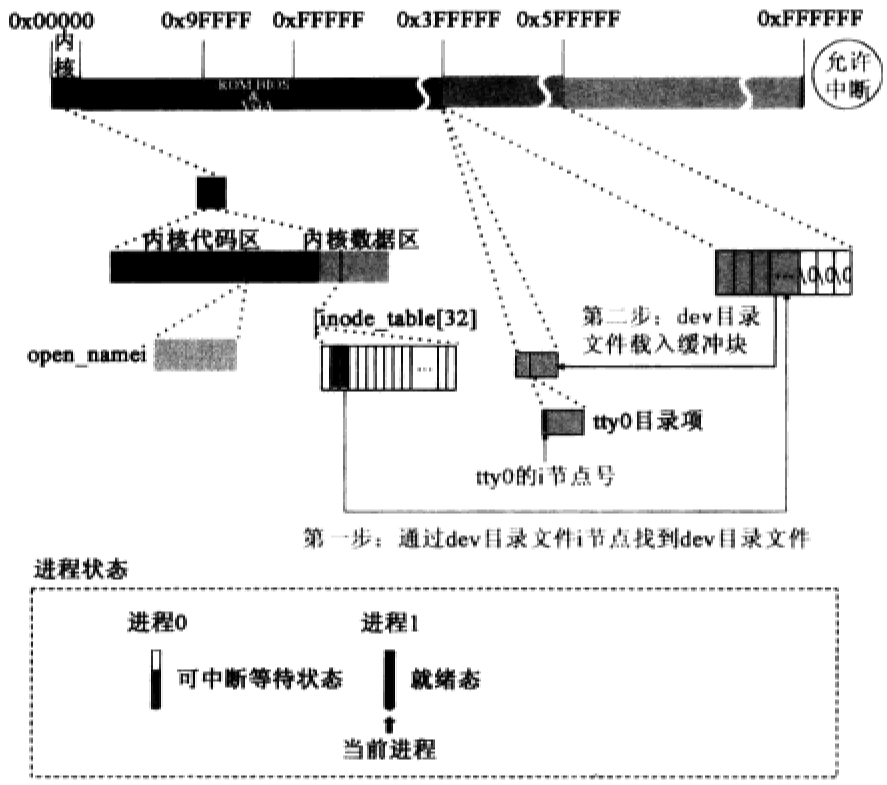


图 2-70　查找 tty0 的 i 节点

现在的情况是，肯定能把目录文件载入进来，载入后，就可以通过 tty0 这个目录项获取 tty0 这个文件的 i 节点号，通过 dir 所记录的 i 节点也可以获取虚拟盘的设备号，这样，最终通过调用 iget 函数就可以得到 tty0 这个设备文件的 i 节点了，从图 2-71 中可见，执行完 iget 函数后，tty0 文件的 i 节点已经被加载到了 inode_table。执行代码如下：

```
// 代码路径：fs/namei.c:
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
inr = de->inode;
    dev = dir->i_dev;
    ...
    if (!(inode=iget(dev,inr)))
    ...
}
```

2.4.7　将 tty0 设备文件的 i 节点返回给 sys_open 系统调用

进程 1 继续执行本节打开终端设备文件的工作。对获取的 tty0 设备文件的 i 节点属性进行检测，看它是否可用，并设置当前时间为 i 节点访问时间，执行代码如下：

```
// 代码路径：fs/namei.c:
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
        !permission(inode,ACC_MODE(flag)))
...
```

```
    inode->i_atime = CURRENT_TIME;
    ...
}
```

在 2.1.6 节中，系统设置了开机启动时间，现在就发挥作用了。由于进程 0 具备了掌控时间的能力，所以现在进程 1 也继承了此能力，这里所指的当前时间是这样定义的：

```
#define CURRENT_TIME (startup_time+jiffies/HZ)
```

可见这个当前时间也是依据开机启动时间 startup_time 计算出来的，这已经在 2.1.6 节中讲解。

最后，就把这个 i 节点与实参的指针相挂接，通过这种方式将 tty0 的 i 节点返回给主调函数 sys_open，返回的方式如图 2-71 所示。

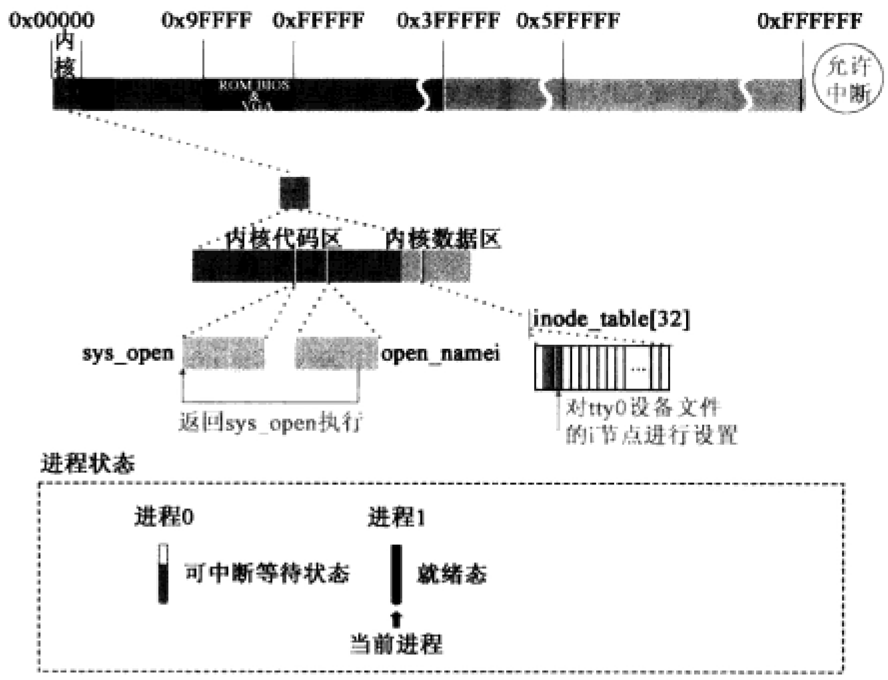


图 2-71 对 tty0 的 i 节点进行处理

执行代码如下：

```
// 代码路径：fs/namei.c:
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
*res_inode = inode;
    ...
}
```

到这里 open_namei 函数执行完毕，程序将返回到 sys_open 函数里面去执行。

2.4.8 分析 tty0 文件 i 节点

进程 1 继续执行本节打开终端设备文件的工作。返回到 sys_open 函数后，对 tty0 的 i 节点属性进行分析，判断出该文件是不是字符设备文件，即根据图 2-72 中的 i_mode 字段进行分析。当然现在的情况是 tty0 文件确实是字符设备文件，那么，通过采集这个 i 节点中的设备号得知，设备号是“4”，具体方式是通过读取图 2-72 中所示 i 节点中的 i_zone[0] 字段。于是就设置当前进

程的 tty 号为该 i 节点的次设备号，并设置进程 1 tty 对应的 tty_table 表项的父进程组号等于进程 1 的父进程组号，这个设置是为多用户使用操作系统做准备的。

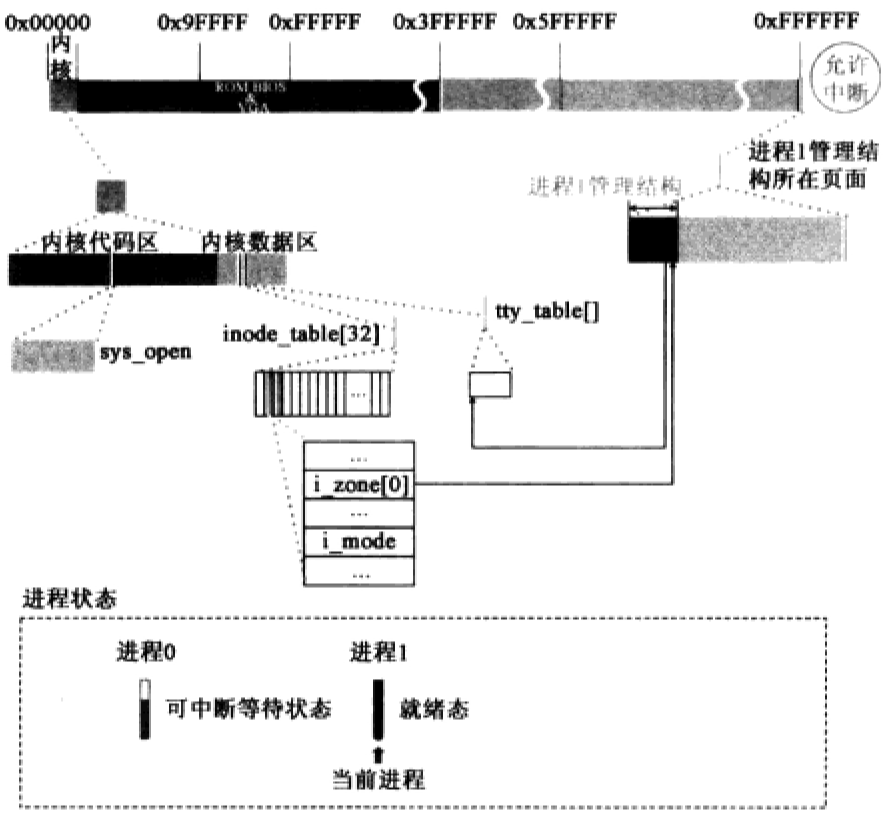


图 2-72　当前进程的 tty 管理结构处理

对应的代码如下：

```
// 代码路径：fs/open.c:
int sys_open(const char * filename,int flag,int mode)
{
    ...
    if (S_ISCHR(inode->i_mode))
        if (MAJOR(inode->i_zone[0])==4) {
            if (current->leader && current->tty<0) {
                current->tty = MINOR(inode->i_zone[0]);
                tty_table[current->tty].pgrp = current->pgrp;
            }
    ...
}
```

2.4.9　设置文件管理结构并返回给用户进程

进程 1 继续执行本节打开终端设备文件的工作。这里将执行 sys_open 函数的最后一步，即让 tty0 这个设备文件的 i 节点与本章 2.4.1 节中在文件管理结构 file_table 里面新申请的空闲项相挂接，这样，当前进程就可以操作这个设备文件了。另外，对于这个空闲项，还要初始化一些属性，包括文件属性、标志、引用次数、i 节点字段，以及文件读写指针等，并最终返回文件句柄，对空闲项的具体设置如图 2-73 右下角所示。

执行代码为：

```
// 代码路径：fs/open.c:
int sys_open(const char * filename,int flag,int mode)
```

```
{
    ...
    f->f_mode = inode->i_mode;     // 这是文件属性，文件属性本质上就是 i 节点的属性
    f->f_flags = flag;
    f->f_count = 1;                // 这是文件的引用次数，这里引用成功了，所以设置为 1
    f->f_inode = inode;            // 这个文件的 i 节点指针，将直接指向到 i 节点表中 tty0i 节点所
                                   // 在的表项上
    f->f_pos = 0;
    return (fd);
}
```

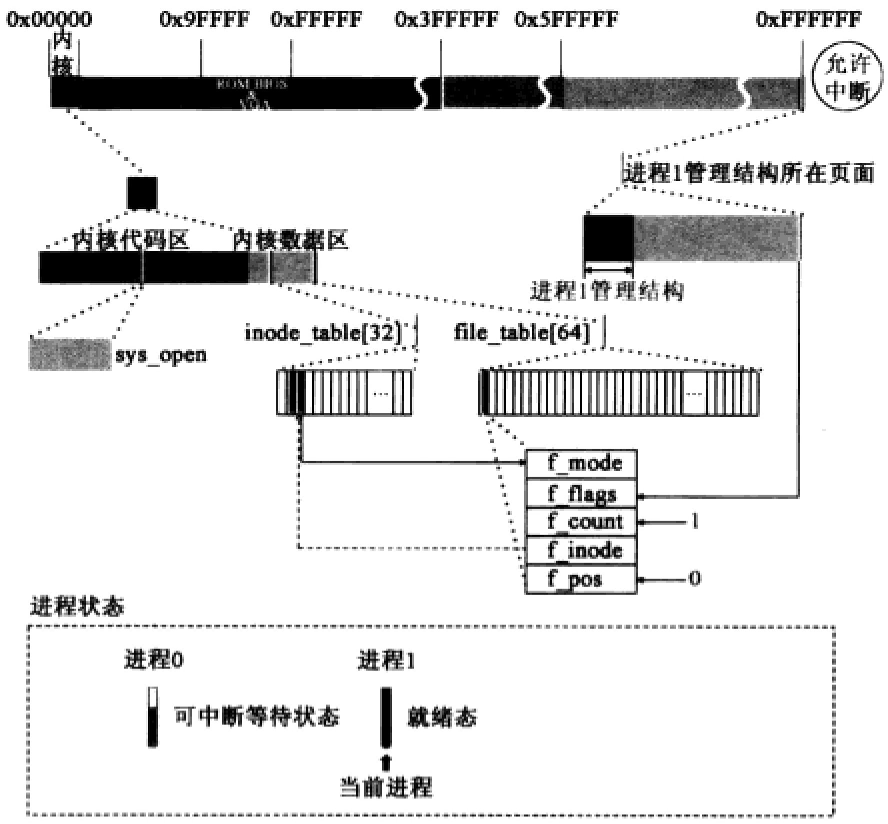


图 2-73 文件槽的设置及返回文件句柄

到此为止，本节进程 1 打开终端设备文件的工作就全部完成了，标志着进程 1 已经具备了对设备文件 tty0 的操作能力，接下来，系统开始执行本章中另一个任务：复制文件句柄。

这个文件打开的动作使进程 1 管理结构中的 *filp[20] 的第一项与系统的文件管理表建立了关系，这个关系将作为标准输入设备，但现在还缺少标准输出设备和标准错误输出设备，所以接下来要对这此文件句柄进行复制，以产生这两个输出设备。

2.4.10 进程 1 复制 tty0 文件句柄

进程 1 在 tty0 设备文件已经被打开的基础上复制文件句柄，一共进行了两次。第一次，系统先调用 dup 函数复制文件句柄，执行代码如下：

```
// 代码路径：init/main.c:
void init(void)
{
    ...
    (void) dup(0);
    ...
}
```

这个函数最终也会对应到 sys_dup 这个系统调用函数中，并进一步调用到 dupfd 函数中，先进行检测，如果没有非正常情况出现，才开始准备复制句柄，执行代码如下：

```
// 代码路径：fs/fcntl.c:
static int dupfd(unsigned int fd, unsigned int arg)
{
    if (fd >= NR_OPEN || !current->filp[fd])
    ...
    if (arg >= NR_OPEN)
    ...
}
```

接下来开始复制句柄，复制句柄的情况如图 2-74 所示，可见，复制文件句柄并不改变内核中 file_table 结构的情况，只是改变当前进程的 *filp[20] 中的相关结构。执行代码如下：

```
// 代码路径：fs/fcntl.c:
static int dupfd(unsigned int fd, unsigned int arg)
{
    ...
    while (arg < NR_OPEN)
        if (current->filp[arg])
            arg++;
        else
            break;
    ...
    (current->filp[arg] = current->filp[fd])->f_count++;
    ...
}
```

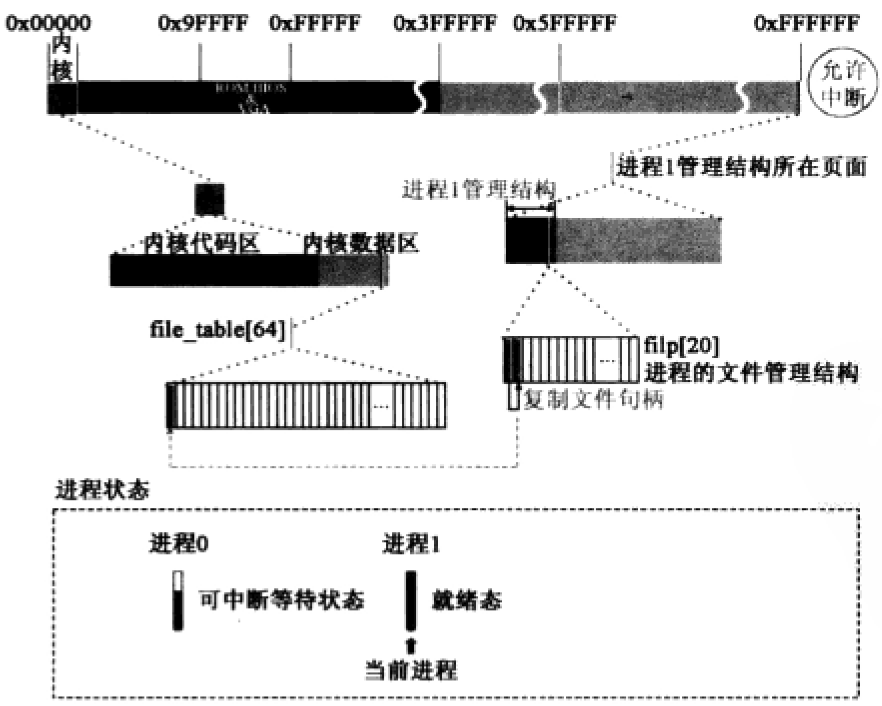


图 2-74　dup(0) 复制文件句柄

在现在的情况下，循环中只会执行一次 arg++;，所以，执行后结果为 1，于是就把当前进程中文件管理结构的第一项的数据全部复制到第二项的位置处，并让这个被复制的文件句柄的引用次数加 1，dupfd 函数执行完毕，并最终返回到了 init 函数中去执行，此次句柄的复制，使进程

1 管理结构中的 *filp[20] 这个文件管理结构，又增添了一组与 tty0 对应的关系，它将作为“标准输出设备”，为终端控制台提供支持。

2.4.11 进程 1 继续复制 tty0 文件句柄

进入 init 函数后，进程 1 继续执行，第二次复制文件句柄，最终还会执行到 dupfd 函数中去操作，同样的执行代码，这次情况有所不同了。注意看代码：

```
// 代码路径：fs/fcntl.c:
static int dupfd(unsigned int fd, unsigned int arg)
{
    ...
    while (arg < NR_OPEN)
        if (current->filp[arg])
            arg++;
        else
            break;
    ...
    (current->filp[arg] = current->filp[fd])->f_count++;
    ...
}
```

current->filp 结构中已经有两个句柄符合条件了（这是上一次复制句柄导致的结果），所以在这个循环中 while (arg < NR_OPEN) 和 arg++; 被执行了两次，这就会把第一项文件句柄的全部信息再复制给第三项所在的位置，导致进程 1 管理结构的文件结构中有了三个相同的文件句柄了，它们都与刚才打开的 tty0 这个设备文件在 file_table 中的对应项相绑定。dup 函数执行完毕，又返回到 init 函数中去执行。此次复制文件句柄的效果如图 2-75 所示，请读者注意进程文件管理结构 *filp[20] 中的相应变化。

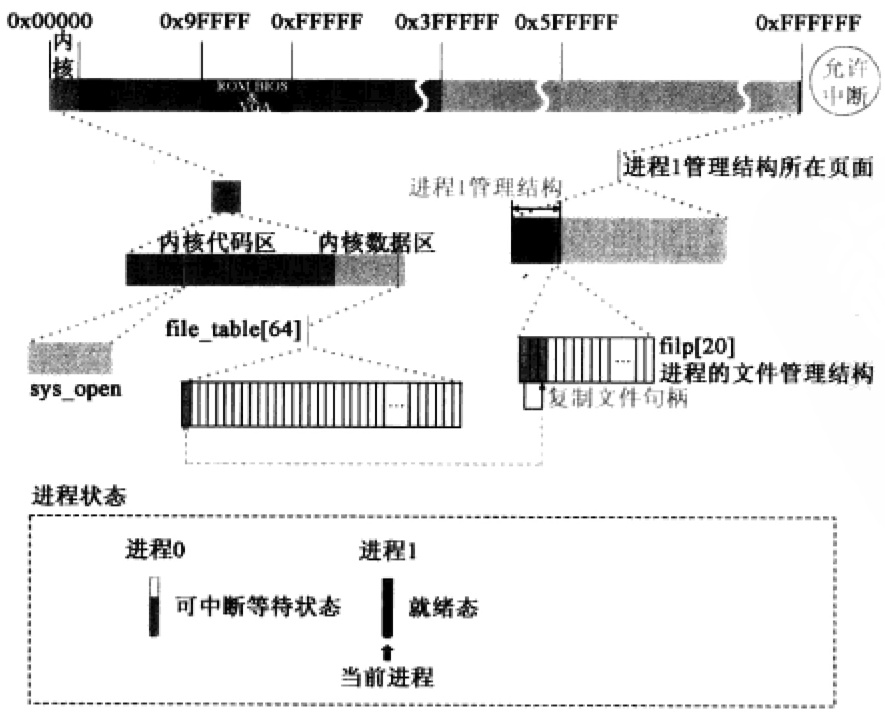


图 2-75 dup(0) 再次复制文件句柄

此次句柄的复制，使进程 1 管理结构中的 *filp[20] 这个文件管理结构在第一次复制的基础

上，再次增添了一组与 tty0 对应的关系，它将作为“标准错误输出设备”，为终端控制台提供支持。本节内容到这里已经全部结束了。这标志着本章所述的第二阶段任务已经完成，进程 1 具备了与外设以文件方式沟通的能力，并使用这种能力，与终端设备文件建立了 3 套关系，接下来，就要完成本章第三阶段的任务了，这个阶段任务的核心就是：由进程 2 为用户构建人机交互平台。

下一节我们将从进程 2 的创建开始讲起。

2.5　创建进程 2

本节开始完成本章第三阶段的内容。这里将以进程 1 为母本创建进程 2，具体的创建步骤与进程 0 创建进程 1 的步骤完全一致。不同之处在于，进程 1 已经在进程 0 的基础上拓展了与外设的交互能力，这种能力将在进程 1 创建进程 2 的过程中全部遗传给进程 2，以此为进程 2 构建人机交互平台奠定基础。

另外，进程 0 和进程 1 现在都具备了参与多进程轮询的能力，所以在此过程中，我们假设又发生了时钟中断，看看轮询会不会产生。

2.5.1　进程 1 准备创建进程 2

进程 1 开始创建进程 2。创建的方式和本章第 2 节中进程 0 创建进程 1 的方式完全一样，只是最终创建的结果不一样。通过系统调用端口，最终找到 sys_fork 这个系统调用函数，在此过程中，也会不断地把一些寄存器的数值压栈，进入 sys_fork 后，调用 find_empty_process 函数，在 task[64] 中，为进程 2 寻找一个空闲位置，此时会是第三个空闲项，并得到新进程的进程号，之后，再陆续地压入一些数据，并最终调用 _copy_process 函数以进程 1 为蓝本复制进程。图 2-76 中给出了系统为进程 2 新申请的进程槽。

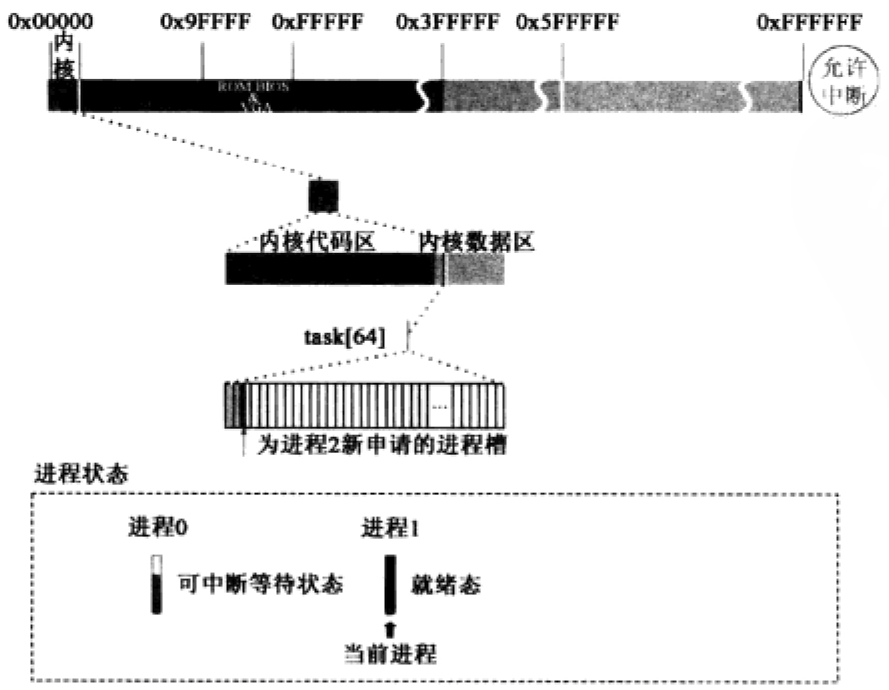


图 2-76　进程 1 创建进程 2

2.5.2 复制进程 2 管理结构并进行调整

为进程 2 的管理结构和内核栈申请空闲页面，图 2-77 中给出了新申请的页面在内存中的位置，请读者注意右上角内存条中的黑色线条，它代表了进程 2 管理结构所在页面的位置。并将指针挂接在 task[64] 中，从图 2-77 可见，进程 2 将占据 task[64] 结构中的第三项。之后将进程 1 管理结构的信息复制给进程 2，作为进程 2 管理结构的雏形。此时以主内存区中进程 1 的管理结构为蓝本进行复制，复制完毕后，将进程 2 的状态设置为不可中断等待状态，以免被轮询到，然后对进程 2 的管理结构信息进行调整。

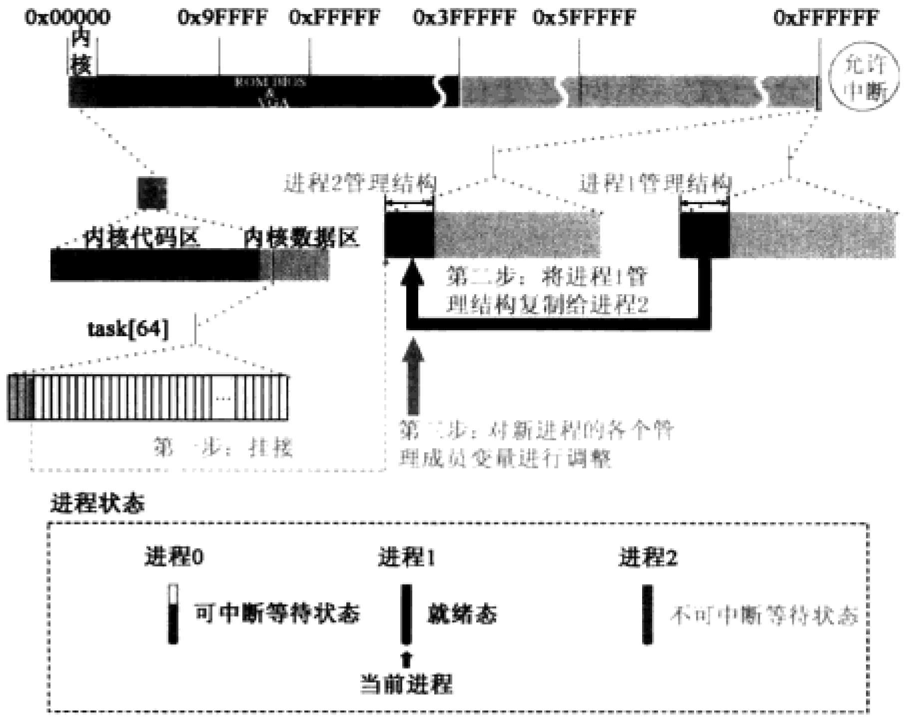


图 2-77 复制进程及调整

2.5.3 设置进程 2 的页目录项并复制进程 2 的页表

为进程 2 复制页表和设置页目录表。同样通过调用 copy_mem 函数来做准备工作，采集进程 1 的代码段和数据段信息，并根据进程 2 的进程号设置进程 2 的代码段和数据段的段基地址，然后设置进程 2 的局部描述符表中的代码段和数据段信息。这些操作都完成后，调用 copy_page_tables 函数来复制进程 2 的页表，要在内存中申请新的页面，然后把进程 1 的页表中的页表项拷贝到进程 2 中，但这回不是 160 项了，而是 1024 项，如图 2-78 右半部分所示。然后，设置进程 2 的页目录表，图 2-78 中给出了设置的具体位置。并刷新页变换高速缓存。这样，进程 2 和进程 1 就具备了操作相同内存页面的能力了。

2.5.4 调整进程 2 管理结构中与文件有关的内容

进程 1 继续创建进程 2，现在要对进程 2 的管理结构中与文件管理相关的结构进行调整，即将引用次数加 1。通过 2.4 节中的内容我们得知，进程 1 中的 *filp[20] 结构里面现在有 3 套与 tty0 对应的关系。所以，进程管理结构的复制致使进程 2 的 *filp[20] 结构里面也自然而然地存

在着同样三套与 tty0 对应的关系，每一次复制都会导致 tty0 文件的引用次数 f_count 加 1，如图 2-79 所示，因此 tty0 文件的引用次数将在循环中被连续三次加 1。

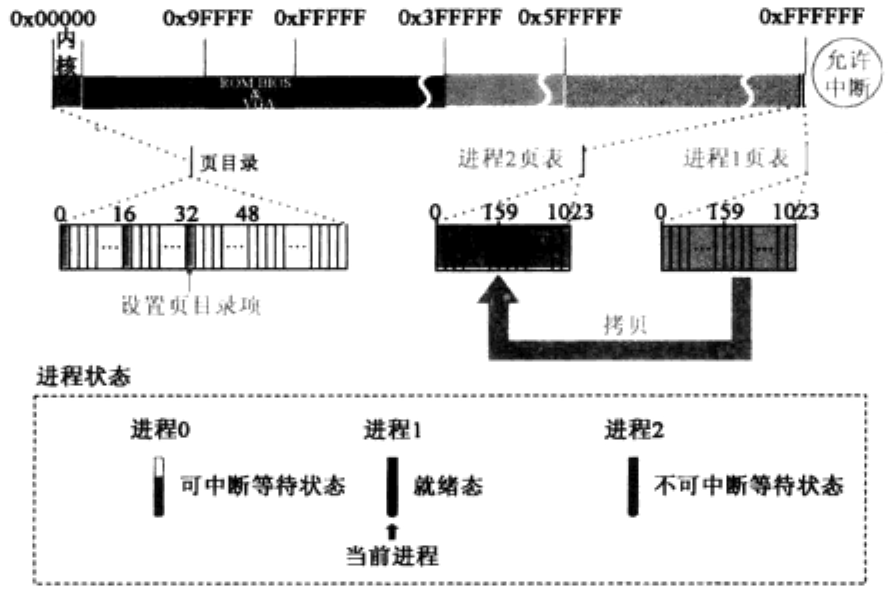


图 2-78　为进程 2 复制页表和设置页目录项

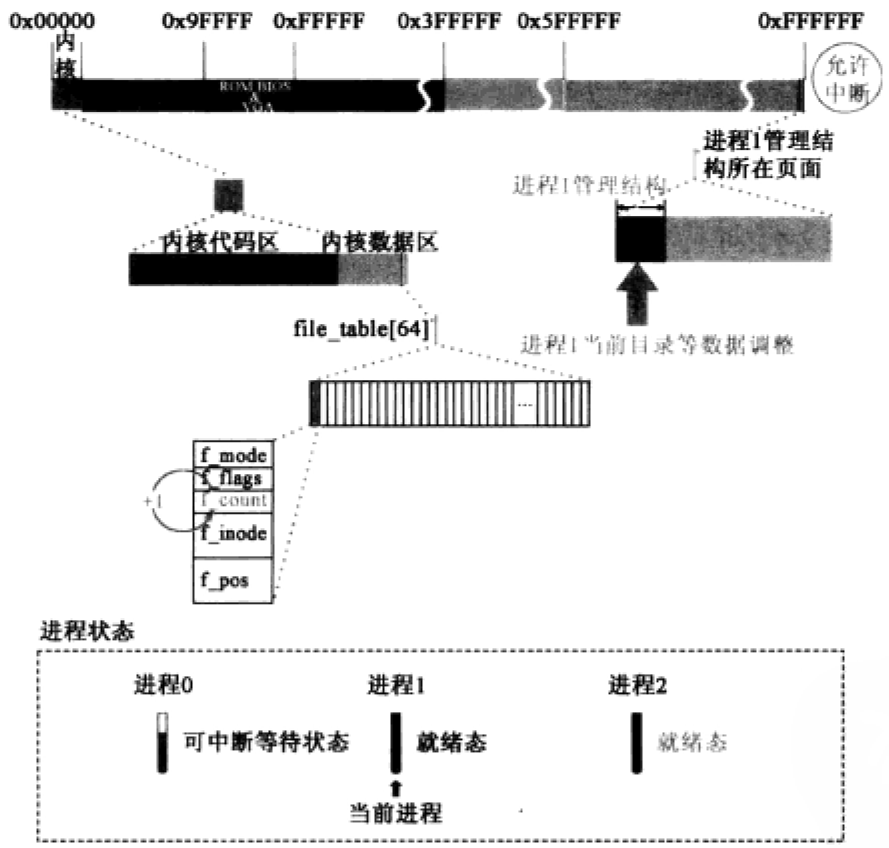


图 2-79　调整进程 2 进程管理结构

执行代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    for (i=0; i<NR_OPEN;i++)
```

```
    if (f=p->filp[i])
        f->f_count++;
    ...
}
```

然后，由于进程 1 在加载根文件系统时，已经将根 i 节点及由此而导致的进程 1 的当前进程工作目录 i 节点 pwd 和当前进程根目录的 i 节点 root 都对应到了“根 i 节点”上，所以，当前进程管理结构的复制，致使进程 2 中的 pwd 和 root 也对应到了根 i 节点上，因此根 i 节点的引用数 i_count 此时增加了两次。

对应代码如下：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    if (current->pwd)
        current->pwd->i_count++;
    if (current->root)
        current->root->i_count++;
    ...
}
```

同样，由于进程 1 并没有对应任何可执行文件，所以进程 1 所对应的“可执行文件的 i 节点”一定是空的，因此将跳过下面的代码：

```
// 代码路径：kernel/fork.c:
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    if (current->executable)
        current->executable->i_count++;
    ...
}
```

最后，将进程 2 的局部描述符表和任务状态描述符表与全局描述符表进行挂接，进程 2 现在就已经具备了参与轮询的能力，于是将进程 2 的状态设置为就绪状态。到此为止，进程 1 已经把进程 2 创建完毕，进程 1 返回 init 继续执行。

2.5.5 进程 1 执行过程中发生时钟中断

进程 1 继续执行，现在进程管理槽中已经有三个进程了，假如此时发生时钟中断，会出现什么样的情况呢？

中断发生后，执行到 _timer_interrupt: 处，将一些寄存器的数据压栈，并最终调用 _do_timer

函数，如图 2-80 所示：

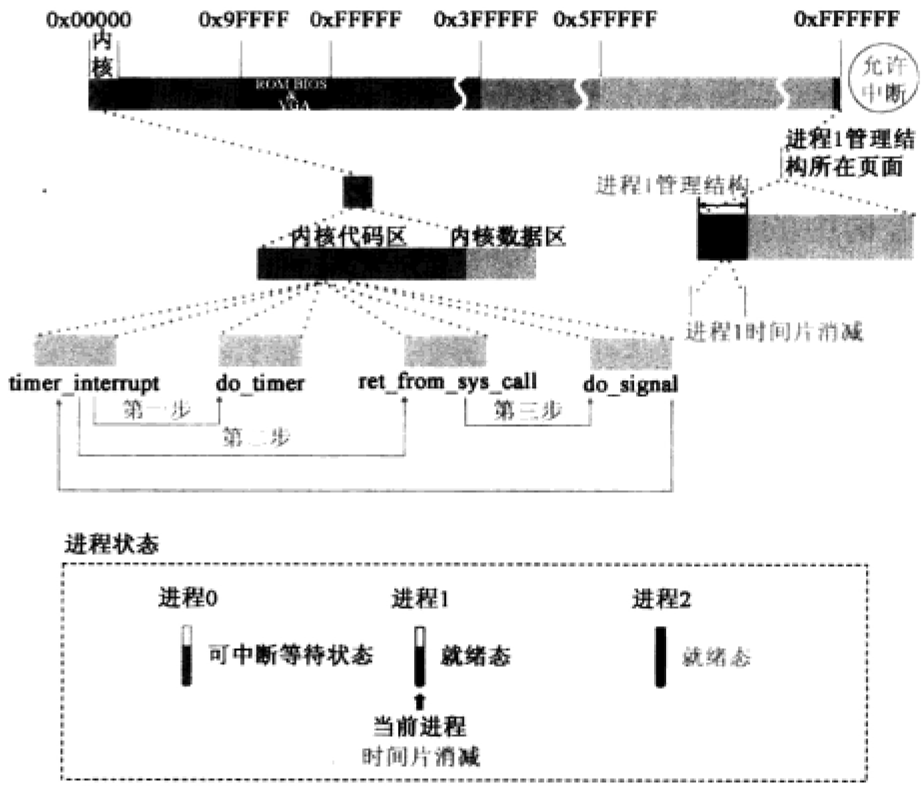


图 2-80　时钟中断处理

执行代码如下：

```
// 代码路径：kernel/system_call.s:
_timer_interrupt:
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx
    pushl %ebx
    pushl %eax
    movl $0x10,%eax
    mov %ax,%ds
    mov %ax,%es
    movl $0x17,%eax
    mov %ax,%fs
    incl _jiffies
    movb $0x20,%al
    outb %al,$0x20
    movl CS(%esp),%eax
    andl $3,%eax
    pushl %eax
    call _do_timer
     ...
```

之后进入 do_timer 函数去执行，由于进程 1 的时间也远没有用完，所以只会削减它的时间片，然后就返回了，不会导致进程切换，执行代码如下：

```
// 代码路径：kernel/sched.c:
void do_timer(long cpl)
```

```
{
...
if ((--current->counter)>0) return;
...
}
```

之后会跳转到 ret_from_sys_call 处，调用 _do_signal 函数检查信号，现在的情况是，并没有哪个进程的管理结构中的信号位图有变化，所以 _do_signal 也会返回。本次时钟中断就结束了。

2.5.6 进程 1 从时钟中断返回，准备切换到进程 2

时钟中断结束后，执行 if (!(pid=fork())) 这一行代码，如图 2-81 所示。这时候就要判断 if 中的数值的真假了，这一点与 2.1.10 节中进程 0 创建完进程 1 后仍然执行进程 0 的道理是一样的。

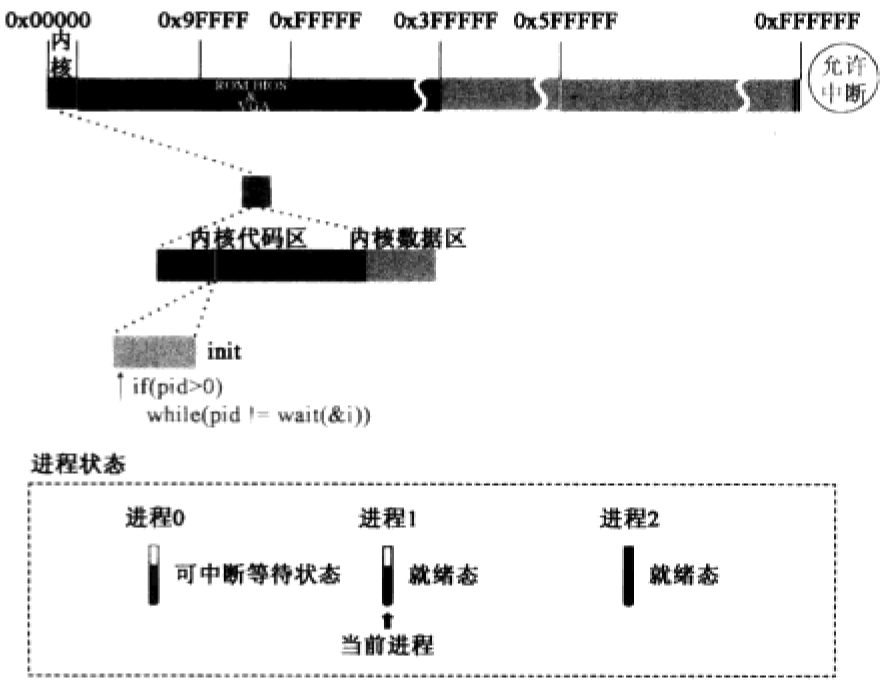


图 2-81 时钟中断后进程 1 的执行情况

到此为止，2.5 节的内容就介绍完了，进程 1 成功创建了进程 2，并将自己的能力全部遗传给了进程 2，这就为以后进程 2 构建人机交互界面奠定了基础。进程 2 现在已经具备运行的能力了，进程 1 的工作也结束了，接下来进程 1 将切换到进程 2，把执行权交给它。

2.6 进程 1 等待进程 2 退出

本节继续完成第三阶段的内容。这里将详细介绍进程 1 切换到进程 2 的过程，本节内容是进程 1 的最后一步工作，但这并不仅仅是一个普通的进程切换，而是通过调用 wait 函数使进程 1 处于等待进程 2 退出的状态，这是因为将来一旦进程 2 因为某种原因退出，进程 1 将重新执行第 2.3 节至第 2.6 节讲述的一切主要工作，重建一个进程，并让其再次构建人机交

互界面。

2.6.1　进程 1 查找它自己的子进程

进程 1 开始做最后的工作。通过调用 wait 函数来完成本节的任务，执行代码如下：

```
//代码路径：init/main.c:
void init(void)
{
    ...
    if (pid>0)
        while (pid != wait(&i))
    ...
}
```

随后会映射到系统调用函数 sys_waitpid 中，该函数先要对所有的进程进行遍历，看看哪个进程的父进程是进程 1，这样进程 2 就被初步选定了，执行代码如下：

```
//代码路径：kernel/exit.c:
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    ...
    if ((*p)->father != current->pid)
    ...
}
```

再对进程 2 进行分析，判断进程 2 究竟是不是进程 1 要找的子进程，执行代码如下：

```
//代码路径：kernel/exit.c:
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    ...
    if (pid>0) {
            if ((*p)->pid != pid)
                continue;
        } else if (!pid) {
            if ((*p)->pgrp != current->pgrp)
                continue;
        } else if (pid != -1) {
            if ((*p)->pgrp != -pid)
                continue;
        }
    ...
}
```

在现在的情况下，进程 1 最终会发现进程 2 确实是要找的子进程。如图 2-82 所示，通过遍历进程槽，进程 2 最终被选中。

2.6.2　对进程 2 的状态进行处理

进程 1 继续执行。这里根据进程 2 的状态来分析是不是要给 flag 的数值赋 1，执行代码如下：

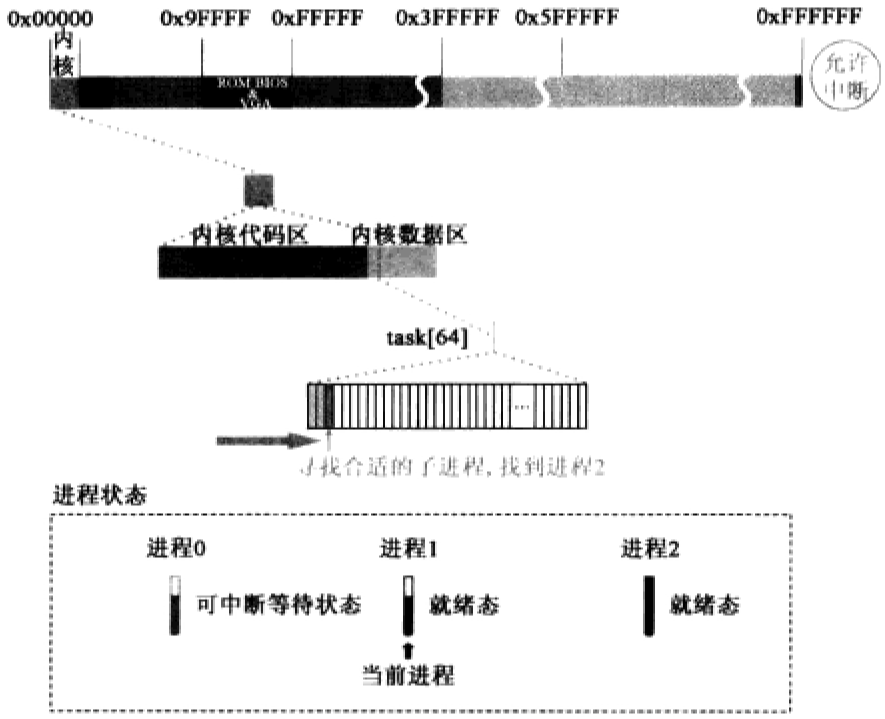


图 2-82　查找进程 1 的子进程

```
// 代码路径：kernel/exit.c:
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    ...
    switch ((*p)->state) {
            case TASK_STOPPED:
                if (!(options & WUNTRACED))
                    continue;
                put_fs_long(0x7f,stat_addr);
                return (*p)->pid;
            case TASK_ZOMBIE:
                current->cutime += (*p)->utime;
                current->cstime += (*p)->stime;
                flag = (*p)->pid;
                code = (*p)->exit_code;
                release(*p);
                put_fs_long(code,stat_addr);
                return flag;
            default:
                flag=1;
                continue;
        }
    ...
}
```

由于进程 2 是新创建的进程，所以它肯定是就绪状态（既不是终止状态 TASK_STOPPED 也不是僵死状态 TASK_ZOMBIE），这样就把 flag 这个标志的数值设置为 1 了，这就说明，已经为进程 1 找到了合适的子进程，如图 2-83 所示。这个函数的目的就是等待指定的子进程退出，现在它指定的子进程是就绪状态，并没有退出，所以，它就要一直等待下去了。

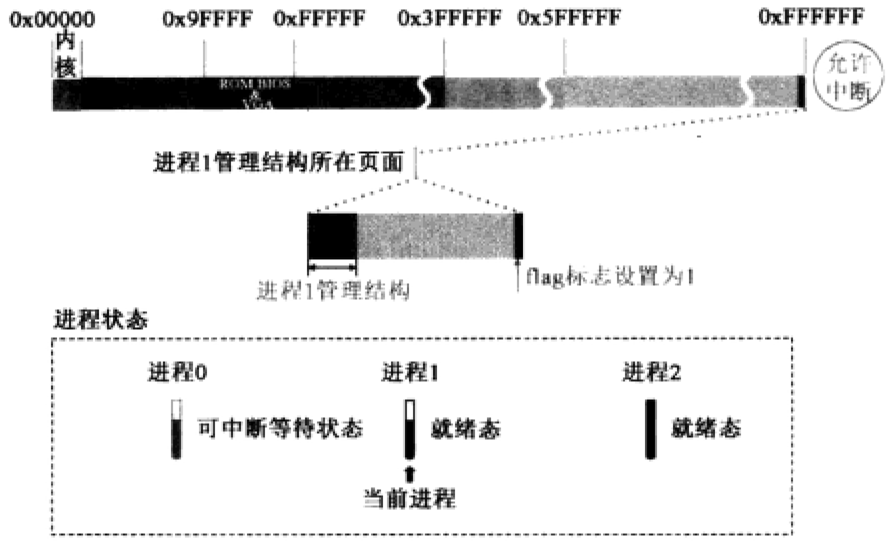


图 2-83 进程 2 的状态处理

2.6.3 切换到进程 2 执行

进入 if (flag) 去执行，进程 1 准备切换到进程 2。进程 1 状态的变化如图 2-84 所示，请读者注意图中进程 1 的状态变化。执行代码如下：

```
// 代码路径：kernel/exit.c:
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    ...
    current->state=TASK_INTERRUPTIBLE;
    schedule();
    ...
}
```

要将当前进程——进程 1，设置为可中断等待状态，之后调用 schedule 函数切换进程。这时候，进程槽中有 3 个进程，除了进程 2 之外，其余的两个都不是就绪状态，所以，一定会切换到进程 2 去执行。如图 2-84 所示，通过遍历进程槽，最终切换到进程 2 执行。

到此为止，在本章第三阶段中，进程 1 创建进程 2，并切换到进程 2 的工作就全面完成了，在下一节中，进程 2 将开始正式执行，并以进程 1 所具备的能力为基础，构建人机交互界面，使系统达到怠速状态。

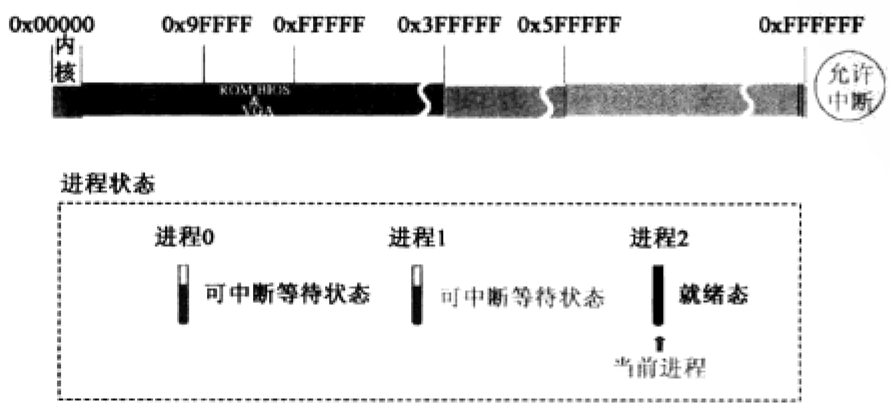


图 2-84 切换到进程 2

2.7 shell 程序的加载

本节继续完成本章第三阶段的内容。进程 2 开始执行了，它将进一步拓展与外设的交互能力，具体表现为，将通过加载 shell 程序，为最终建立起一套人机交互界面打好基础。

shell 程序的加载主要分为两个阶段：第一阶段是为 shell 的加载做准备，主要通过 main.c 中的 close( )、open( ) 和 execve( ) 来实现；第二阶段是根据实际需要，加载 shell 程序，主要通过 page.s 中的 call _do_no_page 实现。

本节第一阶段主要做了 4 个方面的准备。

1. 进程 2 为 shell 程序的执行所做的外围准备

每个进程都有自己的运行环境，包括：配置文件、环境变量（用户对进程运行环境的具体要求，包括日志记录等）、参数（用户希望系统能够处理哪些指令），等等。每个进程的运行环境都有可能不同，这些运行环境需要具体问题具体对待，只有用户在使用系统的时候，根据具体的情况才能做出合适的选择，系统的设计者是无法预计的，所以，操作系统的设计者就为用户留了余地，让用户能够更自由地进行选择，之后系统通过对用户的选择进行分析，做出合适的处理。

2. 进程 2 对 shell 程序所在文件进行全方位检测

所有程序的加载都要受到各方面的限制，shell 程序也不例外，这些限制包括：程序所在的文件是否是常规文件，当前进程是否具备执行权限，文件的代码段和数据段长度是否符合规定，等等。这些信息保存在这个程序所在文件的 i 节点和文件头中，操作系统主要通过分析这些信息来判断程序是否可以加载，这些工作都是在 do_execve 函数中完成的。

3. 针对 shell 程序，对当前进程，即，进程 2 管理信息进行调整

进程 2 的管理结构是由进程 1 复制过来的，但进程 2 对应的程序却是 shell，这就必然导致很多复制过来的信息不再适用，如：进程 2 中对应程序的代码段长度和它的父进程的代码段长度可能不一致；进程 2 用户栈所在的页面和它的父进程不一致等。这些不一致在进程 2 执行的时候，会产生混乱，都需要在 do_execve 函数中重新设置。

4. 进程 2 对 shell 程序将来要执行的第一条指令的地址值进行调整

这是一个承上启下的关键点，shell 程序之所以能够根据具体的实际需要，一部分一部分地加载入内存，都是对这个 EIP 值的特殊调整致使的，这也是 do_execve 函数中的最后一步工作。

本节第二阶段的加载工作主要介绍两个方面的问题：

（1）进程 2 加载 shell 程序的条件现在已经完全具备了，但此时并没有把 shell 程序全部载入内存。Linux 0.11 最大支持 16MB 物理内存，但当时，用户的计算机中实际安装的内存条通常都远小于 16MB，留给主内存的空间更为有限。在这种情况下，把程序代码一次性载入主内存固然可以提高程序的执行效率，但如果程序代码量超过主内存的大小，那么最终程序将会因内存空间不够而无法正常执行，更何况现代操作系统还是支持多进程的，多个进程代码的总大小更容易大于主内存，空间不够的情况就会更容易出现，此外，程序的代码毕竟也是要一行行的执行的，并

不是所有代码都需要同时被使用。于是 Linux 0.11 根据程序实际运行情况并着眼于发展，采取了这样一种策略：需要运行哪部分代码就加载哪部分。虽然对执行效率会造成一些可以承受的影响，但多个大程序在有限的内存空间内共同执行就变成了可能，这就顺应了软件发展的潮流。当进程需要执行的程序代码超出已经载入的代码范围时，Linux 0.11 就加载 4KB 的后续代码，即，1 页面的代码内容，不需要的代码就暂时不加载，这就是“缺页中断机制”，第一阶段最后一步对 EIP 的特殊设置，就是为了产生缺页中断，这个工作主要由 do_no_page 函数中调用的 bread_page 函数来实现。

（2）进程 2 此时仅仅是将 shell 程序中最开始 4KB 的代码载入到物理内存，但在 32 位保护模式下，由于只有在线性地址空间内，才能对 shell 程序进行寻址，所以，还要把刚才载入物理内存的 4KB 内容所在的页面映射到线性地址空间内，以此对寻址进行支持。这个工作主要由 do_no_page 函数中调用的 put_page 函数来实现。

以上就是进程 2 加载 shell 程序的主要过程，这个过程所用技术适用于所有进程的子程序加载，也是操作系统的关键点之一。掌握了这个加载的过程，可以对操作系统的一多半内容有一个整体的理解，并为本书以后章节的学习，打下坚实的基础。

shell 程序的加载过程概要图如图 2-85 所示。

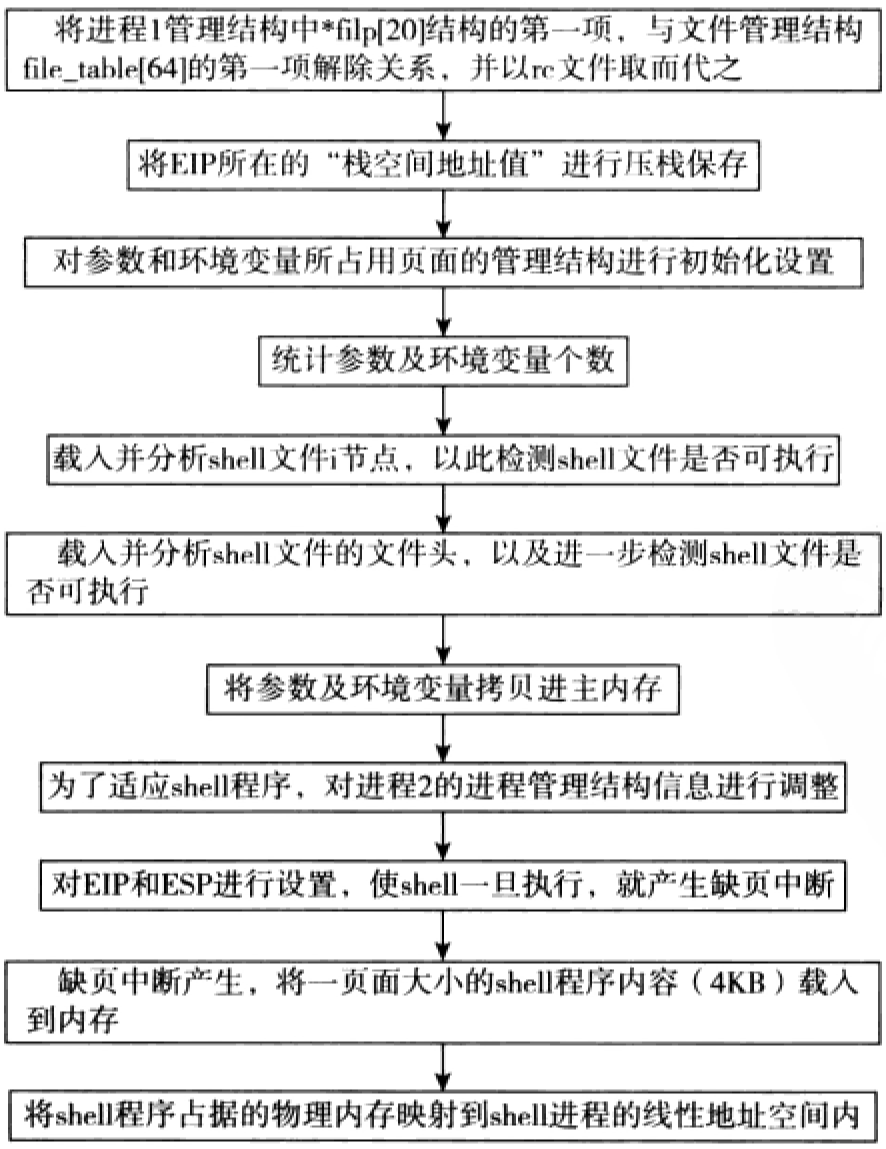


图 2-85　shell 程序加载概要图

2.7.1 进程 2 开始执行

现在开始进行 shell 程序的准备工作，在讲解这个准备工作之前，我们先来关注一下进程 2 是如何开始执行准备工作的。

具体的执行代码如下：

```
// 代码路径：include/unistd.h:
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
 : "=a" (__res) \
 : "0" (__NR_##name)); \
if (__res >= 0) \
 return (type) __res; \
errno = -__res; \
return -1; \
}
```

进程 2 开始执行时，第一行执行的代码和前面 2.3.1 节提到的进程 1 一样，也是 if (__res >= 0) 这一行，因为此时 __res 是 0，所以 return (type) __res 的返回值也是 0。

再看 main.c 中的 init 函数，其中的部分代码如下：

```
// 代码路径：init/main.c:
…
if (!(pid=fork())) {
    close(0);
    if (open("/etc/rc",O_RDONLY,0))
    _exit(1);
    execve("/bin/sh",argv_rc,envp_rc);
    _exit(2);
    }
…
```

前面的 _syscall0 的 return 返回 0，致使执行到 close(0) 这一行时，开始了 shell 的加载工作。

2.7.2 为打开 /etc/rc 文件做准备

进程 2 开始进行 shell 程序加载的外围准备工作，我们先讲解其中的打开配置文件，就是将进程 2 管理结构中 * filp[20] 的第一项和文件管理表 file_table[64] 建立一个关于资源配置文件的关联，以便 shell 程序将来使用。具体的过程是，先调用 close 函数，解除 * filp[20] 的第一项与 file_table[64] 的关系，然后再调用 open 函数，重新建立 * filp[20] 的第一项与 file_table[64] 的关系，下面讲解具体过程。

在打开这个配置文件之前，先要解除当前进程与文件管理表中指定文件的对应关系。解除这个关系的理由是：在程序执行的现阶段，当前进程，即，进程 2 管理信息中的全部内容，都是从进程 1 的管理结构中拷贝过来的，其中就包括文件关系挂接点“* filp[20]”等内容，而这些内容并不完全是进程 2 将来需要的，它还需要建立与 "/etc/rc" 这个配置文件的关系，而这些拷贝过来

的内容中却没有，所以需要先解除旧关系。

解除关系的操作是：调用 close 函数，最终会对应到 sys_close 这个系统调用函数中去执行，将当前进程 * filp[20] 结构中的第一项清空成空闲项，并将 file_table[64] 中与此对应的关系撤销，从而实现旧关系的解除。如图 2-86 所示。

sys_close 函数中执行关系解除动作的主要代码如下：

```
// 代码路径：fs/open.c:
int sys_close(unsigned int fd)
{
    ...
    if (!(filp = current->filp[fd]))
    ...
    if (--filp->f_count)
...
}
```

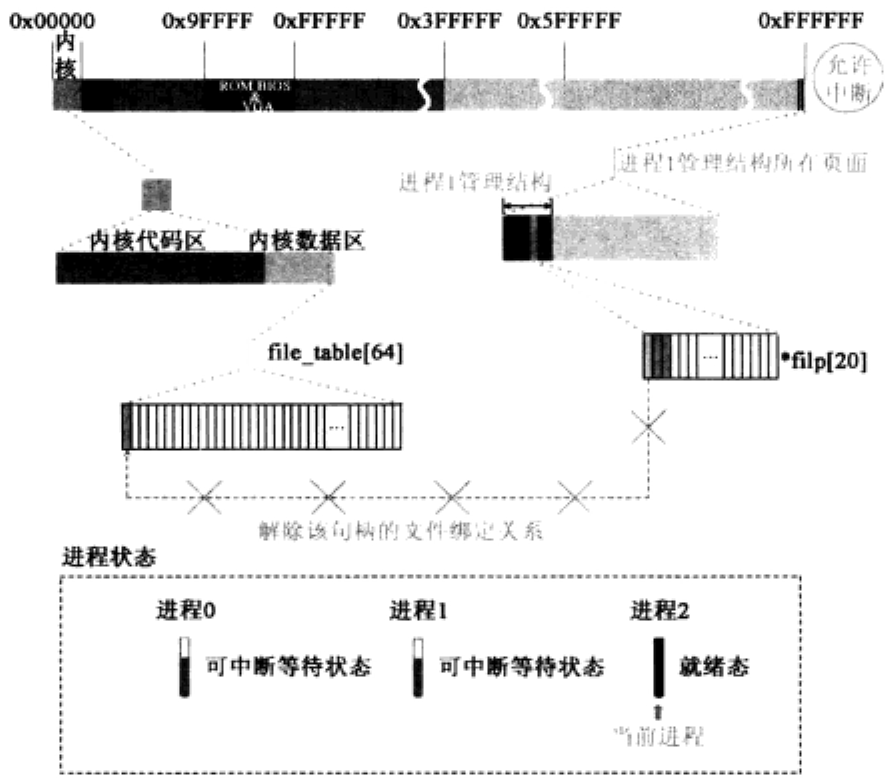


图 2-86　为打开 /etc/rc 做准备

2.7.3　进程 2 打开 “/etc/rc” 配置文件

旧的关系解除后，开始打开 “/etc/rc” 这个配置文件，其目的就是让进程 2 管理结构中 * filp[20] 的第一项和文件管理表 file_table[64] 建立一个关于资源配置文件的关系。这意味着进程 2 具备操作配置文件的能力。

具体步骤是：调用 open 函数，这个调用实际映射到系统调用函数 sys_open 中执行，将进程 2 管理结构中 * filp[20] 刚刚被清空的第一项与 file_table[64] 建立关于 “/etc/rc” 配置文件的关系。

sys _open 函数中建立关系的具体代码，在 4.1.1 节中 “打开终端字符设备文件” 时已经进行了详细的说明。图 2-87 中给出了 “/etc/rc” 配置文件在 i 节点表和内核文件管理表中所占的位置（分别为其中的第 3 项和第 2 项），以及它们之间的关联关系。

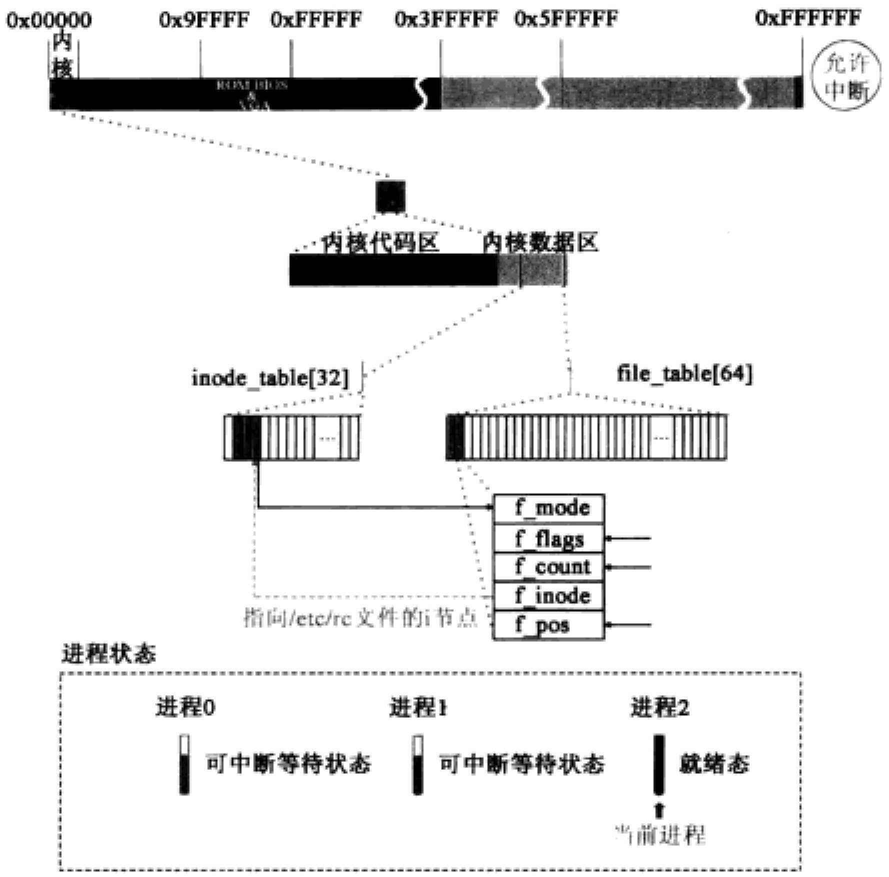


图 2-87　打开 /etc/rc 文件

2.7.4　通过压栈为加载 shell 文件做准备

现在我们来讲解 do_execve 函数的执行情况，在 shell 程序加载的准备工作中，绝大部分操作都是这个函数完成的，对这个函数的理解是整个操作系统中的难点之一。

系统对 do_execve 函数的调用是逐步完成的，在调用到此函数之前，为了支持 shell 程序加载的准备工作更好地完成，系统将一些数据进行了压栈保存，下面我们先来关注一下压栈工作都是如何完成的，它们将会起到怎样的支持作用。

压栈的具体步骤是：系统先调用 execve（"/bin/sh" ,argv_rc,envp_rc）函数，"/bin/sh" 这个实参标识了 shell 程序所在文件的路径名，以便将来系统能够找到 shell 程序所在的文件；argv_rc 和 envp_rc 标识了参数及环境变量，这些数据都是在 main.c 中定义的，代码如下：

```
// 代码路径：init/main.c:
…
static char * argv_rc[] = { "/bin/sh", NULL };
static char * envp_rc[] = { "HOME=/", NULL, NULL };
…
```

以此来支持将来参数和环境变量的加载。

随后就产生软中断，映射到一个系统调用函数 sys_execve 中执行，硬件将 EIP 的值压栈，这个值决定中断程序结束后执行哪一条指令。sys_execve 中要做的最重要的事情，就是把 EIP 值"所在栈空间的地址值"压栈，代码如下：

```
// 代码路径：kernel/system_call.s:
```

```
_sys_execve:
lea EIP(%esp),%eax
pushl %eax
...
```

注意，Intel CPU 是不能通过指令直接修改 EIP 寄存器的值，有了前面的压栈，操作系统就可以通过这个值找到栈中的 EIP 值，修改栈中 EIP 的值，将其修改为指向加载到进程 2 的 shell 程序代码的执行入口，等到中断返回时，硬件就会将这个值赋给 EIP，达到控制中断完成后执行 shell 程序的目的。

压栈工作都完毕后，就开始调用到 do_execve 函数了，代码如下：

```
// 代码路径：kernel/system_call.s:
_sys_execve:
    ...
    call _do_execve
    ...
```

至此，do_execve 函数执行之前的支持工作就做完了，接下来我们开始正式介绍 do_execve 函数为 shell 进程的加载都做了哪些具体的准备工作。整个过程如图 2-88 所示，请读者注意图 2-88 中进程 2 内核栈中所示的 EIP 的位置，将来修改 EIP 的值，就是从这里入手的。

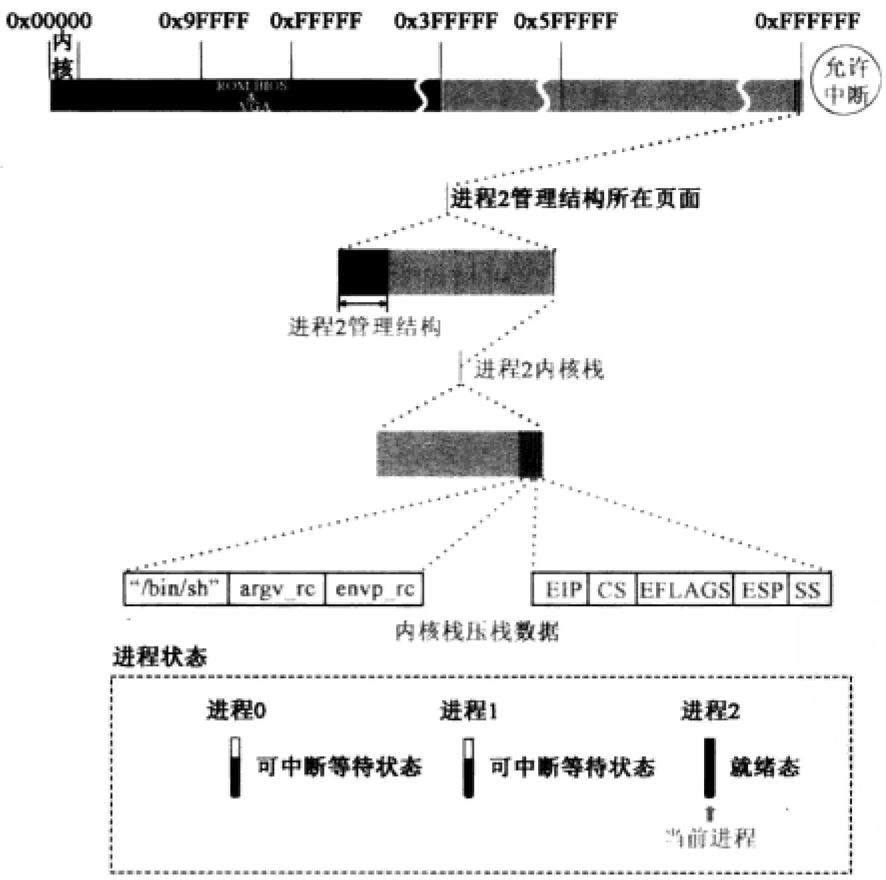


图 2-88　为加载 shell 文件做准备

2.7.5　为参数和环境变量设置做准备

do_execve 开始执行后，马上处理参数和环境变量，目的就是要将参数和环境变量加载到主内存的页面中，下面我们来介绍 do_execve 函数执行之初，在参数和环境变量方面都做了哪

些工作。

为了对参数和环境变量可能占用的内存页面进行管理，do_execve 函数先要构建一个相应的管理结构 page[MAX_ARG_PAGES]，并对这个管理结构初始化清 0，如图 2-89 中右边部分的 page[32] 结构所示。具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    unsigned long page[MAX_ARG_PAGES];
    ...
    for (i=0 ; i<MAX_ARG_PAGES ; i++)
        page[i]=0;
    ...
}
```

接下来，系统将进行另一个准备工作——对 shell 文件进行全方位检测。这就需要把 shell 文件的目录项所在的目录文件读入到缓冲块中，以便得到 shell 文件的 i 节点。这部分内容在本章 2.4 节中已介绍过，在后续第 4 章的文件打开部分还会详细介绍。如图 2-89 的中间部分所示。

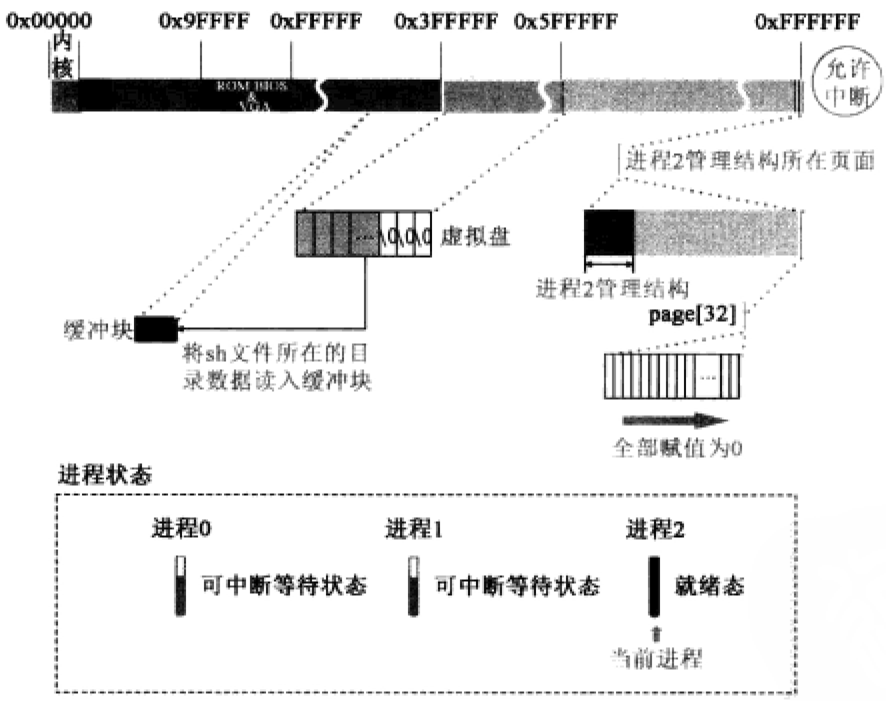


图 2-89　为参数和环境变量设置做准备

2.7.6　得到 shell 文件的 i 节点

系统要想对 shell 文件进行全方位检测，就先要对 shell 文件的 i 节点中提供的文件属性信息进行分析，因为在 Linux 0.11 中，文件自身的所有属性都记载在 i 节点中，没有对 i 节点的分析，其余一切针对 shell 文件的后续检测都无法实现。操作 i 节点的具体步骤是：调用 namei 函数，从虚拟盘上读取 shell 文件的 i 节点。这个函数读取 i 节点的步骤和 4.1.1 节中将要讲解的 sys_ open 函数中调用的 open_namei 函数读取 i 节点的步骤基本一致，我们这里只大致介绍一下：通过分析路径“/bin/sh”，最终把 shell 文件的 i 节点找到，并将其登记到 i 节点管理表中。如图 2-90 所示，最终指向内核 i 节点表的指针信息仍然是通过进程 2 的内核栈返回给 do_execve 函数。

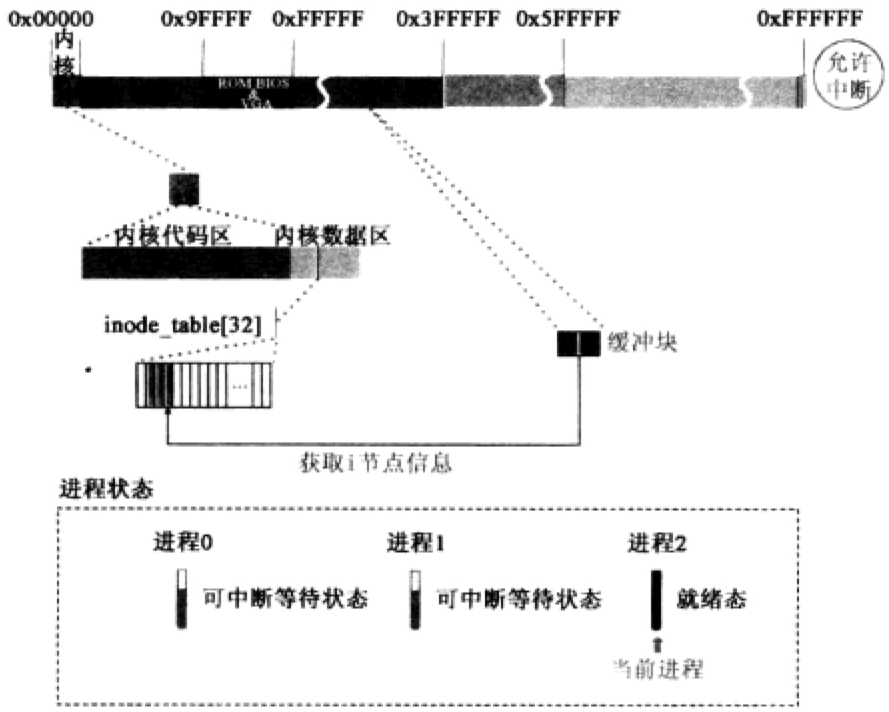


图 2-90　加载 shell 文件 i 节点

这个 i 节点的获取，为后续一系列针对 shell 文件的检测，创造了先决条件。

2.7.7　为加载参数和环境变量做准备

系统对 shell 文件的检测工作先暂停一下，转而进行 shell 程序加载的外围准备工作中对参数和环境变量的准备。继 2.7.5 节中对参数和环境变量所在页面的管理结构进行初始化之后，现在还要对参数的个数和环境变量的个数进行统计。这是因为，参数和环境变量是系统给用户预留的自由设置信息。由于操作系统的版本不同、每个版本具体使用情况不同，参数和环境变量的个数就可能不同，就拿 shell 程序来说，就有好多种版本，只不过 Linux 0.11 这个版本对它们的默认设置是 main.c 中的，代码如下所示：

```
// 代码路径：init/main.c:
...
static char * argv_rc[] = { "/bin/sh", NULL };
static char * envp_rc[] = { "HOME=/", NULL, NULL };
...
```

这两行代码在其他版本中就不见得是这样的默认，而且以后在操作系统的升级换代中，还会发展出更多的版本，更为全面的功能，参数和环境变量也会有所变化，为了适应这种灵活变化，Linux 0.11 在此一律统一计算参数和环境变量的大小，以此得知将来往主内存中拷贝多少信息就足够了，进而得知需要多少内存页面，以免导致内存空间浪费或不足。统计的效果如图 2-91 所示。其中，参数和环境变量本身位于内核数据区，统计的结果存放在进程 2 内核栈的变量中。

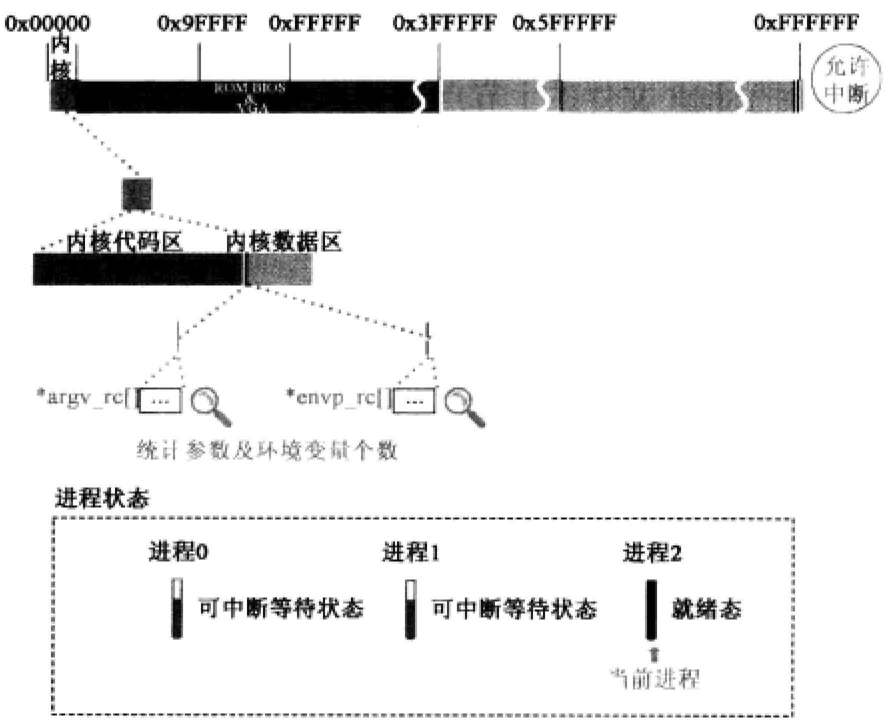


图 2-91 统计参数和环境变量个数

具体的执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    argc = count(argv);
    envc = count(envp);
    ...
}
```

2.7.8 根据 i 节点，对 shell 文件进行检测

外围准备工作中的参数和环境变量大小统计完了，继 2.7.6 节中提到的获取 i 节点工作之后，开始对 shell 文件的检测工作，对 shell 的 i 节点进行分析，具体的操作步骤：

通过对 i 节点中“文件属性”的分析，可以得知这个 shell 文件是不是一个“常规文件”。因为只有是常规文件，即，除了块设备文件、字符设备文件、目录文件、管道文件等特殊文件之外的文件，才有被载入的可能。

执行此任务的代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    if (!S_ISREG(inode->i_mode))
    ...
}
```

毕竟现在系统正在构建一套人机交互界面，既然是人机交互界面，就一定会涉及与用户相

关的一些设置，所以依据 i 节点中提供的：inode->i_mode、inode->i_uid 和 inode->i_gid，检测当前进程是否有执行 shell 文件的权限，并对用户 ID 等信息进行设置。如图 2-92 所示，注意图中 shell 文件的 i 节点占用的位置，是 inode_table 中的第四项。

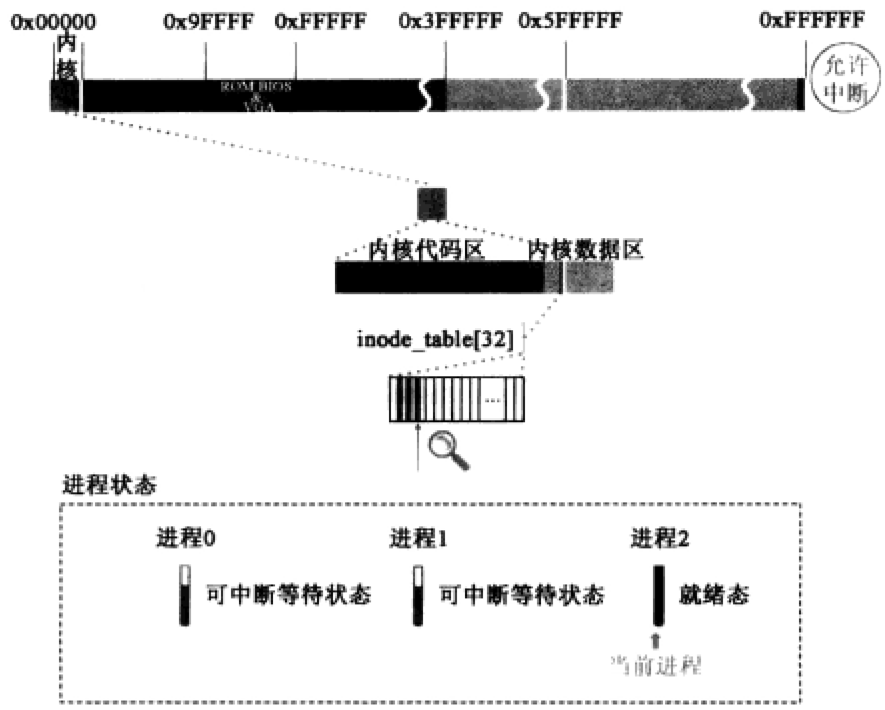


图 2-92 对 shell 文件 i 节点进行检测

2.7.9 检测 shell 文件头

接下来对 shell 文件头进行检测，这是对 shell 文件进行全方位检测的重要一环。这个文件头中记载着这个 shell 程序的代码段长度和数据段长度等关键的属性信息，这些信息直接影响到 shell 程序将来被加载入内存后是否能够正常执行，而且通过我们后面的讲解也可得知，如果 shell 程序确实能够执行，那么这个文件头中的信息将引导系统对 shell 程序的加载。

由于 shell 文件也是以数据块的形式存储在虚拟盘上的，所以它的文件头，正常情况下在它第一个逻辑块内，于是，系统就调用 bread 函数，把这个文件头从虚拟盘上读进缓冲块内，如图 2-93 所示，代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    bh = bread(inode->i_dev,inode->i_zone[0]
    ...
}
```

bread 函数的操作方法，在本章 2.3.3 节至 2.3.8 节中已经详细说明了。

2.7.10 备份文件头并进行分析

文件头内容载入缓冲块后，先对 shell 文件头进行备份。这是因为缓冲块在系统中总是被频

繁使用，而载入缓冲块中的文件头内容，在接下来的 shell 文件分析过程中，被频繁使用，系统选择将其进行备份并释放该缓冲块，就可以让两者既能频繁使用，提高运行速度，又不额外占用缓冲区资源。如图 2-94 所示，文件头的内容从缓冲区备份到了进程 2 的内核栈。

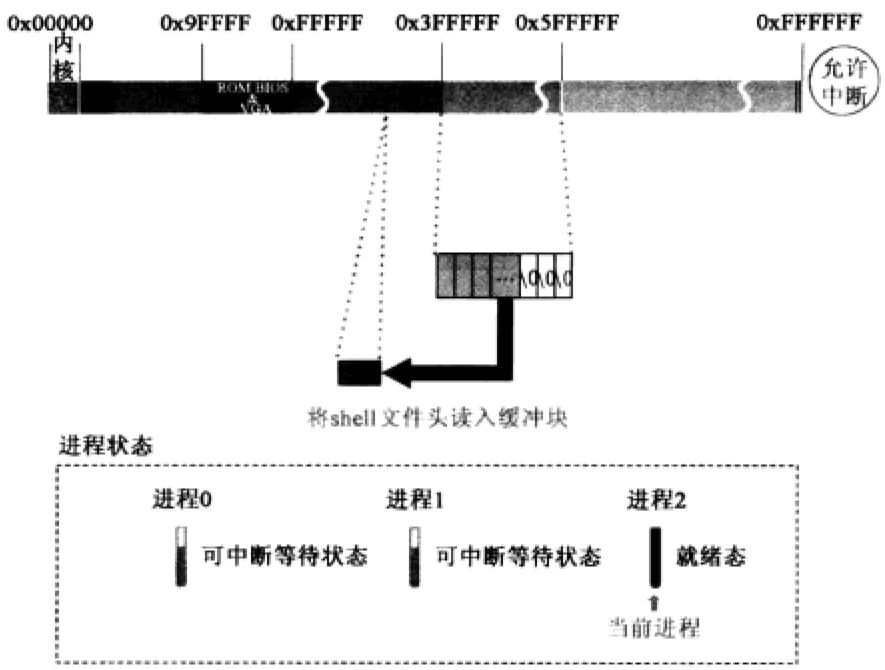


图 2-93　将 shell 文件头读入缓冲区

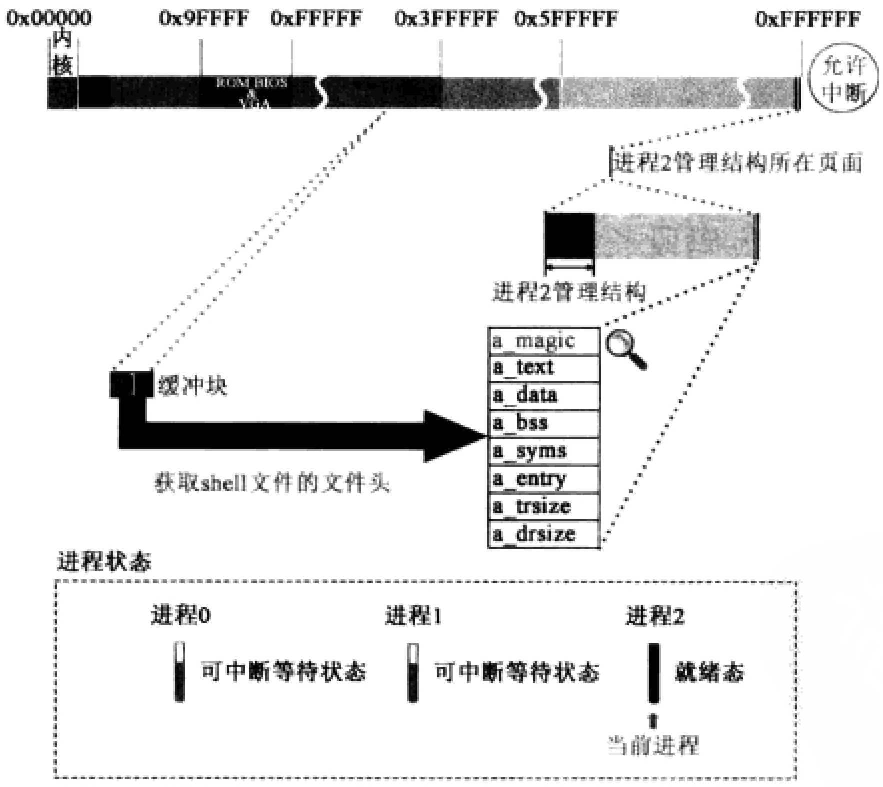


图 2-94　通过分析文件头检测是否是脚本文件

备份之后，开始对文件头进行分析，如果 shell 是脚本文件，就会进入到 if 里面执行，显然 shell 并不是脚本文件，如图 2-94 右上部分所示。所以程序将跳过后面相当多的代码，直接执行后面的释放缓冲块这个动作。

备份、分析、释放缓冲块的具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
```

```
    char ** argv, char ** envp)
{
    ...
    ex = *((struct exec *) bh->b_data);
    if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang))
    ...
    brelse(bh);
    ...
}
```

2.7.11　对 shell 文件进行进一步分析

继续依据文件头和 i 节点提供的信息，对 shell 文件进行检测，判断下面两种情况是否会出现，只要有一种出现，shell 程序就将不能被加载。

情况一：如果执行文件不是需要分页可执行文件（ZMAGIC）、代码重定位部分长度 a_trsize 不等于 0、数据重定位信息长度不等于 0、代码段 + 数据段 + 堆段长度超过 50MB、i 节点表明的该执行文件长度小于代码段 + 数据段 + 符号表长度 + 执行头部分长度的总和，则这个 shell 文件就不能执行。这是因为 Linux 0.11 对可执行文件是有限制的，比如说，每个进程的最大线性地址空间最多就 64MB，那么这个进程的代码如果无限制地大下去，现在的 Linux 0.11 版本将无法处理；再比如，i 节点中提供的信息和实际对文件头中测算的相应信息不符，就说明这个可执行文件很有可能出现了错误或文件损坏等，显然这个 shell 程序就无法执行了。检测所用到的文件头中的字段如图 2-95 右半部分的文件头中的绿色部分所示。

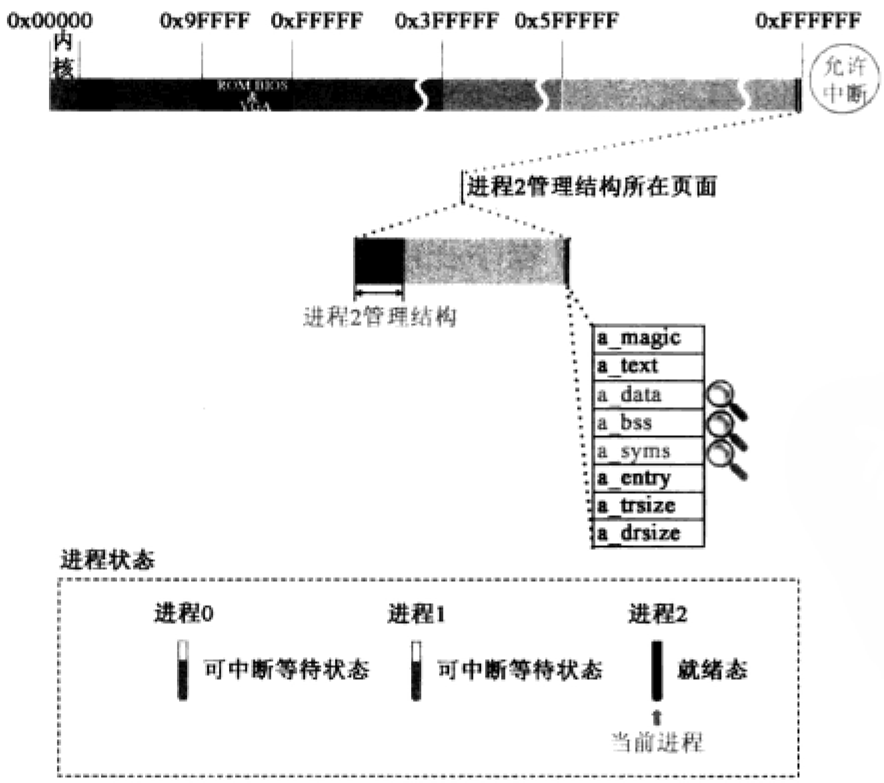


图 2-95　继续对文件头进行分析检测

具体检测执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
```

```
    char ** argv, char ** envp)
{
    ...
if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
        ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
        inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
        retval = -ENOEXEC;
        goto exec_error2;
        }
...
}
```

情况二：如果文件头部分长度不等于一个缓冲块大小（1024 字节），也不能执行。因为这个文件头的信息，先是被加载到缓冲块里面的，如果这个文件头大于一个缓冲块，说明加载不完整。如果连加载都不完整，文件头肯定就无法完成引导 shell 程序加载到内存的这项工作了。检测的具体执行代码：

```
//代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
if (N_TXTOFF(ex) != BLOCK_SIZE) {
printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
retval = -ENOEXEC;
goto exec_error2;
}
...
}
```

综上所述，系统要将问题消灭在萌芽阶段，一旦发现 shell 文件可能出现以上问题，直接就不加载了，这还不仅仅是为了提高效率，更重要的是，shell 文件是可执行文件，它是要执行的，如果放过这些错误，而且碰巧这个 shell 程序在内存中还能执行，那就可能导致一些意想不到的后果。

2.7.12 拷贝参数和环境变量

经过前面在 shell 文件检测方面所做的工作，系统可以确定，shell 程序确实具备加载的能力，于是这方面的准备工作完毕了，接下来将继续对外围准备工作中的参数和环境变量的准备做工作，要把参数和环境变量及其相关辅助信息拷贝到内存指定的页面中去，具体执行步骤是这样的：

两次调用 copy_strings 函数，分别将环境变量和参数拷贝到主内存指定的页面中，拷贝的时候要参考环境变量和参数在前面各自统计出来的个数，并依托前面设置的页面管理结构 page[MAX_ARG_PAGES] 和偏移指针 p。效果如图 2-96 所示。

具体执行代码如下：

```
//代码路径：fs/exec.c:
```

```
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    p = copy_strings(envc,envp,page,p,0);
        p = copy_strings(argc,argv,page,p,0);
...
}
```

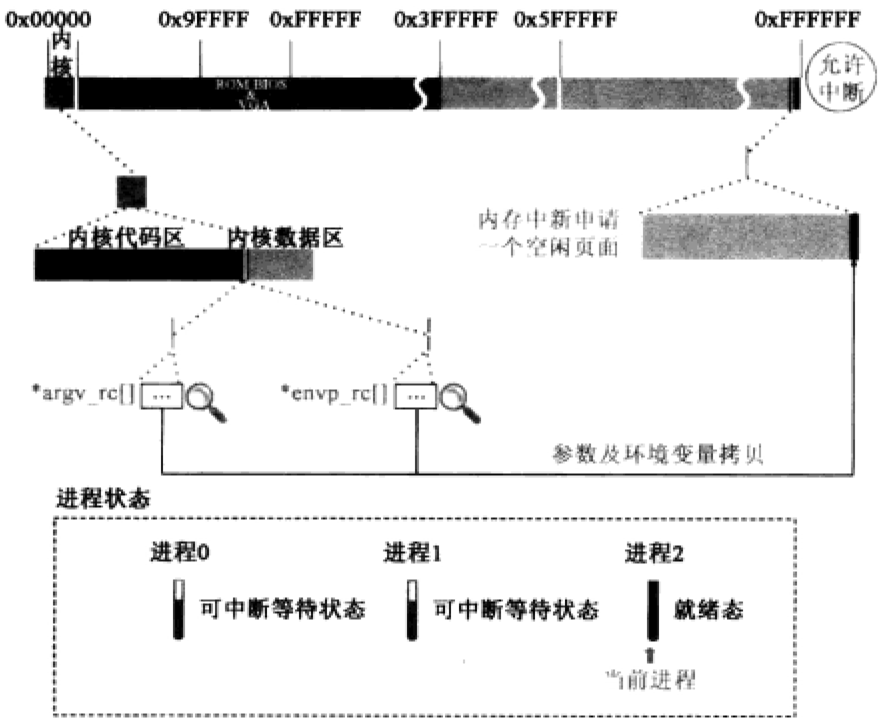


图 2-96 拷贝参数及环境变量

2.7.13 调整进程 2 的管理结构

到现在为止，为 shell 程序加载所做的全部准备中，对 shell 文件的检测已经全部完成，对外围方面的准备也已经基本完成，接下来系统为了适应 shell 程序，对当前进程，即进程 2 自身的管理信息进行调整，这项工作也是全部准备工作中重要的组成部分。

系统开始从可执行文件方面进行调整。executable 是进程管理结构中的一个成员，它表示该进程所对应的可执行文件的 i 节点指针，这个指针将会直接指向 i 节点表。由于进程 2 管理信息的内容都是从进程 1 拷贝过来的，所以这个成员的数据也一定被拷贝过来了，这导致它们都与同一个可执行文件保持着关系，现在的情况是，进程 2 此时需要与新的可执行文件，即 shell 文件建立关系了，所以原来的关系就要解除掉并建立新的关系。具体操作步骤如下：系统先判断进程 2 的 executable 成员是否由于信息拷贝保留着原有的可执行文件关系，现在的情况是，进程 1 本来就没有与哪个可执行文件建立过关系，所以拷贝过来的 executable 成员也不存在原有保留关系，于是就用不着调用 iput 函数将 i 节点关系解除了，而是直接将 shell 文件的 i 节点指针赋值给进程 2 的 executable 这个成员，这将使它直接指向 i 节点表中 shell 文件的 i 节点项，从而建立关系。如图 2-97 所示。

具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    if (current->executable)
            iput(current->executable);
        current->executable = inode;
...
}
```

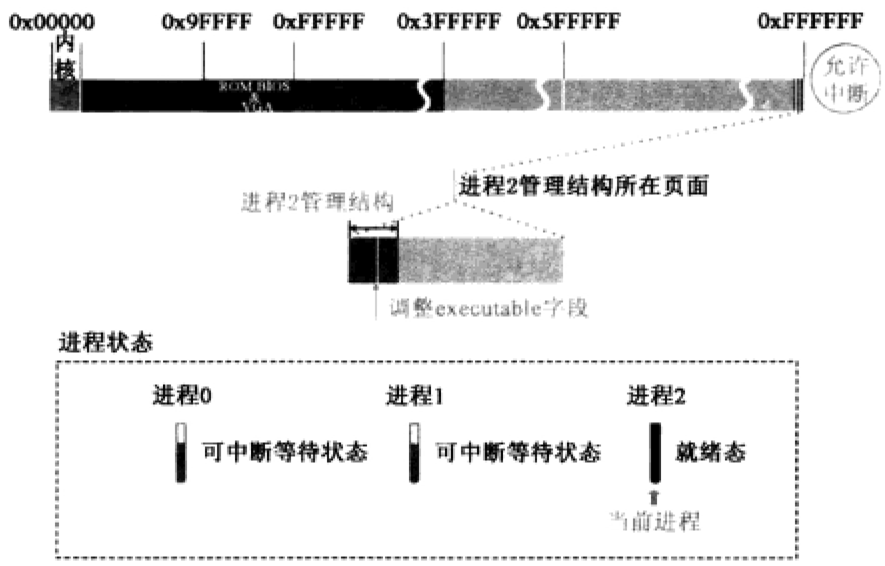


图 2-97　调整进程 2 的管理结构

2.7.14　继续调整进程 2 管理结构

系统为了适应 shell 程序的加载，继续调整进程 2 管理结构的信息，现在要针对两方面进行调整，即信号处理函数句柄和文件打开标志，这些都是针对从进程 1 拷贝过来的管理信息已经不再适用而进行的调整。

在信号处理函数句柄方面，进程 2 尚未运行过任何程序，所以也就不可能有哪个用户程序将信号处理函数的句柄挂接在 sigaction 结构上，因此，为了将来挂接的方便，这里统一将所有句柄都清空，如图 2-98 底部所示。

具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    for (i=0 ; i<32 ; i++)
        current->sigaction[i].sa_handler = NULL;
...
}
```

在关闭打开的标志方面，原来进程 1 如果在执行时打开了哪些文件，就都以“位”的形式记载在 close_on_exec 这个成员中，这些信息也都随着进程管理信息的复制拷贝给进程 2 了，所以，系统也需要针对 close_on_exec 进行调整，close_on_exec 所标识的打开的文件，现在都需要关闭，

并最终将 close_on_exec 所有位清 0，如图 2-98 中间部分所示。

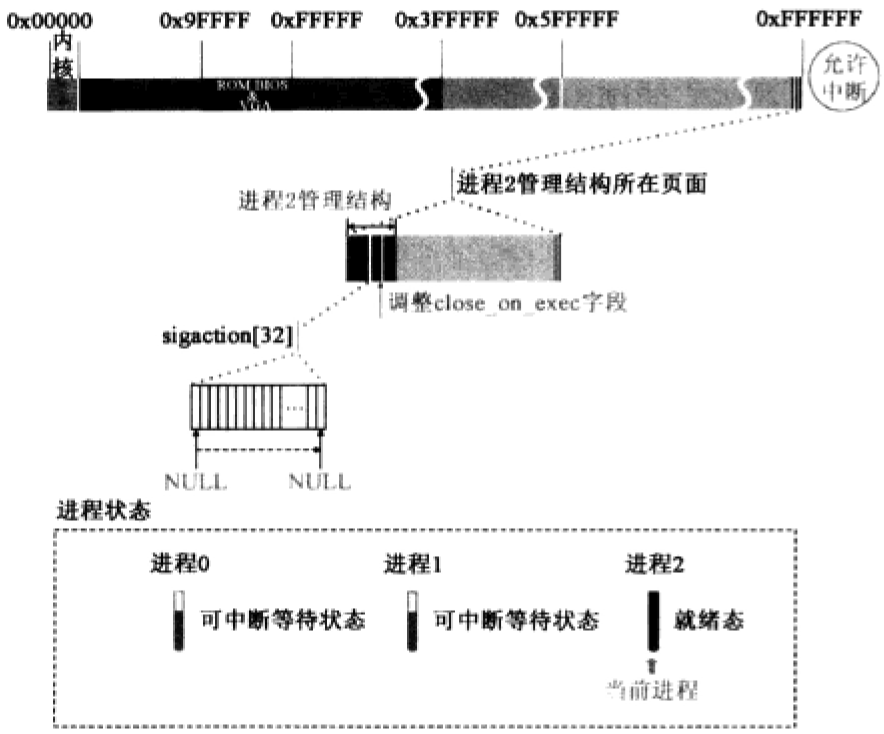


图 2-98 清空信号处理句柄以及检测并清 0 文件的位图

具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
for (i=0 ; i<NR_OPEN ; i++)
    if ((current->close_on_exec>>i)&1)
        sys_close(i);
current->close_on_exec = 0;
...
}
```

2.7.15 释放进程 2 继承的页面

继续在内存管理方面对进程 2 的管理结构信息进行设置。进程 1 中代码段和数据段所占用的页面信息及相应的页表信息也都复制给了进程 2，而现在进程 2 将要对应全新的 shell 程序了，它将不能再继续沿用原来的代码段和数据段中的数据信息，所以要将代码段和数据段所占用的页面及相关的管理页表释放掉。具体操作步骤如下：

调用 free_page_tables 函数，并以进程 2 管理信息中局部数据描述符表内提供的代码段和数据段以及段限长的信息为参照，将原来代码和数据所占用的物理页面以及页表所占用的物理页面“释放”掉，这里所指的释放，其本质是将那些物理页面的引用次数减 1，而并非彻底清 0，这是因为进程 2 的这些信息都是从进程 1 拷贝过来的，这就导致两者现在暂时共用相同的物理页面，进程 2 现在不需要了，进程 1 以后还要用，所以只能将引用次数减 1，效果如图 2-99 所示。

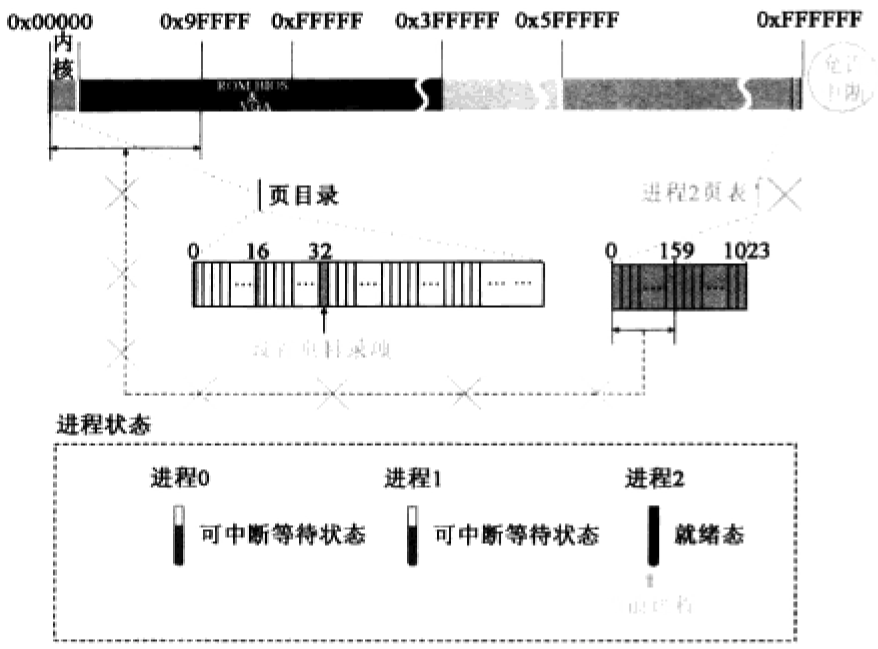


图 2-99　释放代码段和数据段所占页面

具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
   free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
   free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
...
}
```

2.7.16　检测协处理器

继续对进程 2 的管理结构信息在协处理器使用方面进行设置。操作系统中有一个关于协处理器使用的标志：last_task_used_math，如果这个标志对应的是当前进程，那就一定要将该标志清空，这是因为当前进程肯定没有与协处理器发生过关系。另外，进程管理信息中也有协处理器使用的标志，也要清 0。如图 2-100 右边部分所示：

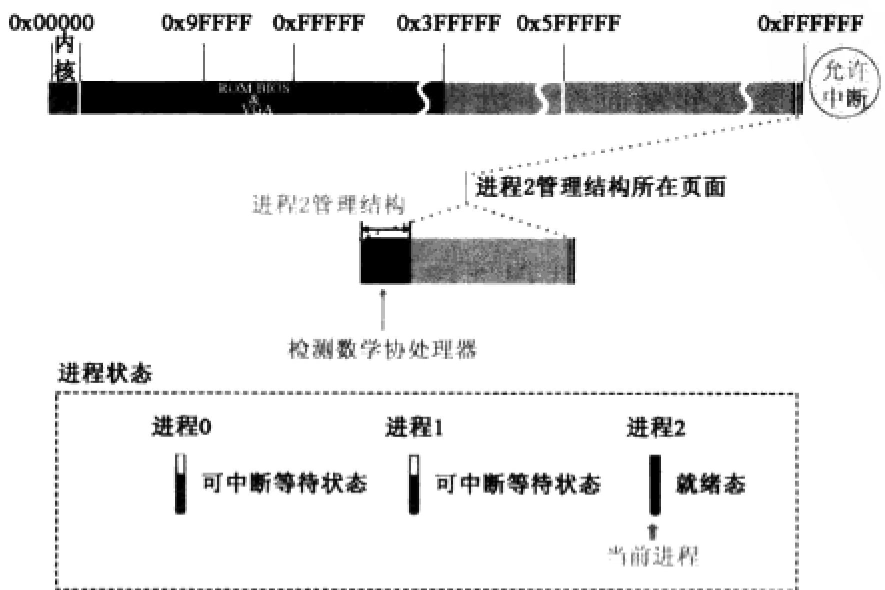


图 2-100　检测数学协处理器

相应的执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    if (last_task_used_math == current)
        last_task_used_math = NULL;
   current->used_math = 0;
...
}
```

2.7.17　调整 shell 程序所在的线性空间地址

继续对进程 2 的管理信息在局部数据描述符表方面进行设置。从 2.7.15 节的介绍中我们得知，进程 2 肯定不能再沿用从进程 1 中拷贝过来的局部数据描述符了，因为它要对应新的 shell 程序，现在 shell 程序的文件头信息已经载入内存了，系统就可以通过文件头中的信息来确定 shell 程序的代码段等情况，这些将为重新设置进程 2 的局部数据描述符表提供支持，具体执行步骤如下：

调用 change_ldt 函数来实现局部数据描述符表的重新设置，这里面涉及两个重要的实参传递，一个是 ex.a_text，这是 2.7.10 节备份的文件头信息中提供的 shell 程序代码段的长度，通过这个数据，系统就可以设置进程 2 的局部数据描述符表以及推测出段限长；另一个是 page，在 2.7.5 节中我们已经介绍过，它是用来管理进程 2 中参数和环境变量所占用内存页面的数据结构，在 change_ldt 函数中，它所对应的物理内存，将作为进程 2 的一部分，被映射到其线性地址空间内。change_ldt 函数执行完毕后，新的段限长数值将被返回，这样系统就可以据此测算出参数和环境变量在内存中的最新空间占用情况，为后面创建参数和环境变量指针管理表做准备。调整的效果如图 2-101 所示。

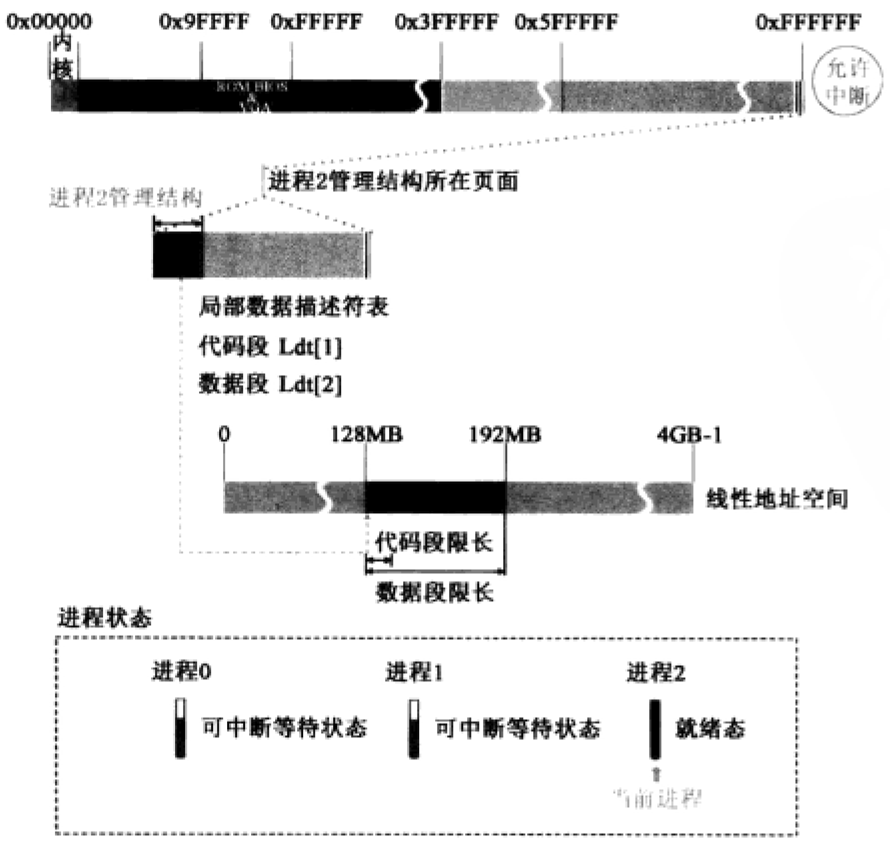


图 2-101　调整代码段与数据段段基址

具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
...
}
```

2.7.18 为 shell 程序准备参数和环境变量

现在系统先把对进程 2 管理结构的设置暂停，转而去处理 shell 程序加载准备工作中的另一项重要外围准备工作——参数和环境变量的准备，这将是此项准备工作的最后一步，即，创建参数和环境变量的指针管理表。参数和环境变量既然可以由用户来自由设置，就肯定不止一个，如果能有一个指针管理表来协助管理，将会带来方便，具体操作步骤如下：

调用 create_tables 函数，去新建这个指针管理表。这里将以 2.7.7 节中统计出来的参数和环境变量的个数为依托，并参考 2.7.17 节中刚刚测算出来的参数和环境变量在内存中最新的空间占用情况，将这个指针表的管理信息，拷贝到内存指定的位置去。之后，系统将根据这个函数的返回值，再次重新确定参数和环境变量的内存中最新的空间占用情况，这次确定的数据，将成为进程 2 的栈底地址。至此，全部的外围准备工作就结束了。如图 2-102 所示：

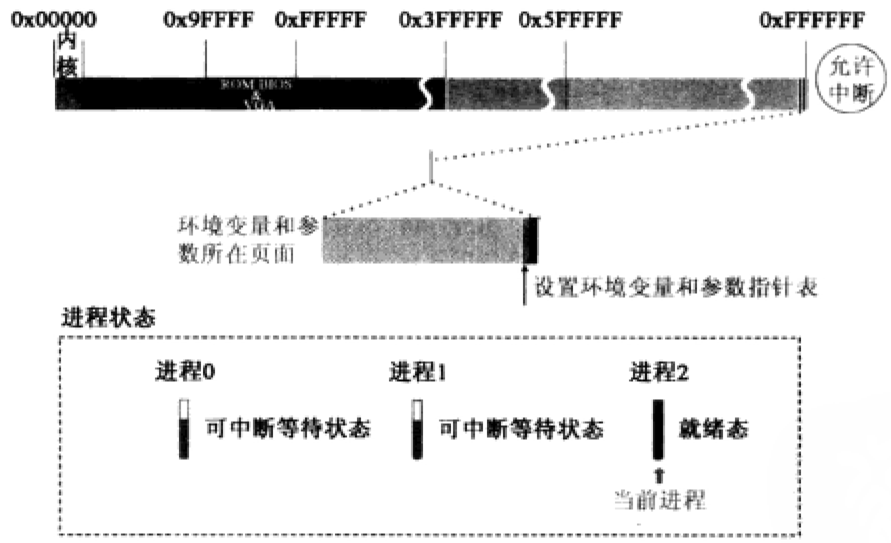


图 2-102　重新设置各段信息

具体执行代码如下：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
p = (unsigned long) create_tables((char *)p,argc,envc);
...
}
```

2.7.19 继续调整进程 2 管理结构

系统转而继续去完成准备工作中，对进程 2 管理结构的调整工作，包括了对以下一些字段的设置：进程代码段尾值字段 end_code、进程数据段尾字段 end_data、进程堆结尾字段 brk、栈底位置字段 start_stack、有效用户 ID euid 和有效组 ID egid，最后，再将主内存中 BSS 段的一页面数据全部清 0。以上调整过程如图 2-103 所示。

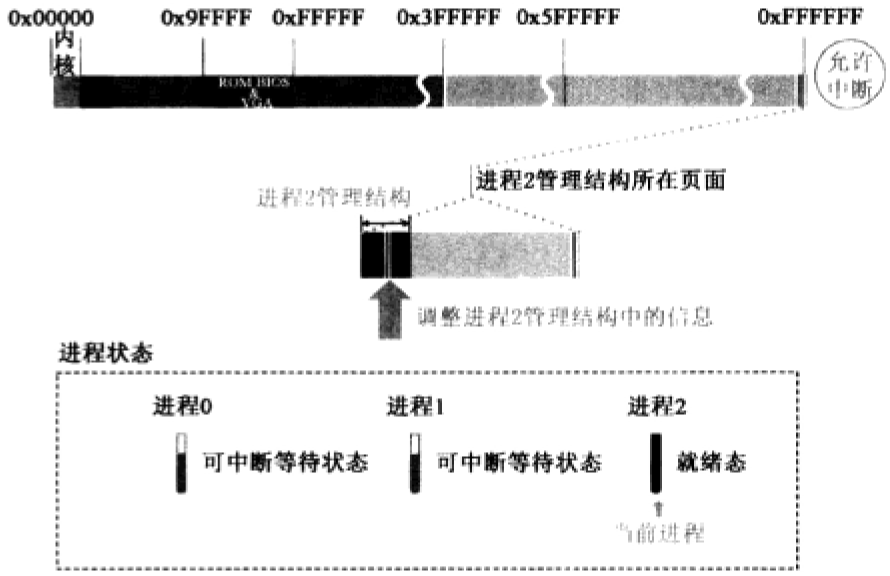


图 2-103 根据环境变量和参数信息调整进程管理结构

到此，系统为 shell 程序加载所做的准备中，调整进程 2 管理结构的这一准备工作就全部做完了。具体执行代码为：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
current->brk = ex.a_bss +
        (current->end_data = ex.a_data +
        (current->end_code = ex.a_text));
    current->start_stack = p & 0xfffff000;
    current->euid = e_uid;
    current->egid = e_gid;
    i = ex.a_text+ex.a_data;
    while (i&0xfff)
        put_fs_byte(0,(char *) (i++));
...
}
```

2.7.20 调整 EIP，使其指向 shell 程序入口地址

系统现在开始为 shell 程序的加载做最后的准备工作，即对 EIP 的值进行调整，使之指向 shell 程序的第一条指令处，这将导致软中断服务程序执行完毕后，系统会从 shell 程序起始位置开始执行。另外，为了保证 shell 执行后能够使用栈空间，也对栈顶指针进行了设置。具体步骤如下：

在前面2.7.4节中已经介绍过，系统将EIP值“所在栈空间的地址值”压栈，这个地址值将成为这里所进行设置的基础，系统将分别用shell程序起始地址值ex.a_entry和当前进程栈顶位置值p（此时栈顶和栈底是重合的），来对EIP和栈顶指针ESP进行设置，设置完毕后，通过sys_execve函数中的ret指令，将重新设置的EIP值载入eip寄存器，并指引代码的执行。具体的执行代码为：

```
// 代码路径：fs/exec.c:
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
eip[0] = ex.a_entry;
eip[3] = p;
return 0;
...
}

_sys_execve:
   ...
   addl $4,%esp
   ret
```

注意：代码中eip[0]表示从“压栈保存的EIP栈空间地址位置处”偏移量为0的位置，应该就是eip。

代码中eip[3]表示从“压栈保存的EIP栈空间地址位置处”偏移量为3的位置，应该是esp，以上设置就为产生缺页中断埋下了伏笔，如图2-104所示。

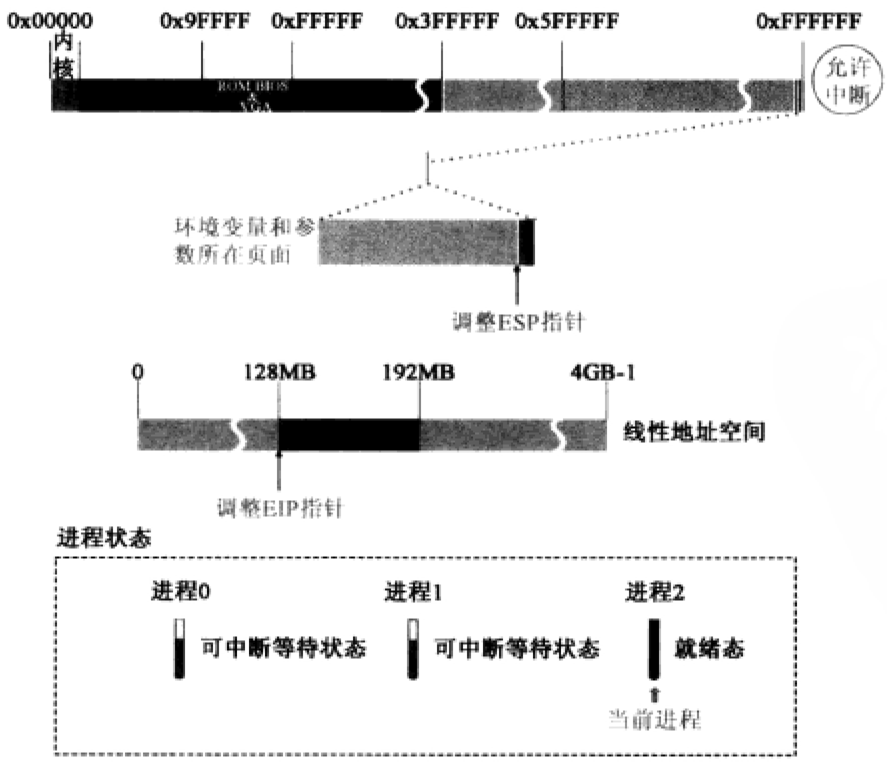


图2-104 设置EIP，为缺页中断埋伏笔

至此，系统为shell程序加载所做的全部准备工作都已经完成了，下面开始正式加载shell程序。

2.7.21　shell 程序执行引发缺页中断

shell 程序是被动加载的，系统中并没有加载 shell 程序的代码。sys_execve 函数返回后，中断服务程序就完成了，系统将依据 EIP 提供的指令地址继续执行，通过 2.7.20 节中我们介绍的内容来看，接下来要执行的就是 shell 程序的第一条指令，但是 shell 现在并没有任何代码被载入内存，也就不会在内存中占据任何页面。因此 shell 程序的第一条指令所对应的线性地址经过硬件解析后，一定会发现并不存在对应的页面，于是硬件将产生一个“缺页中断”信号，系统为了响应这个信号，页面错误函数 page_fault 将会被调用，具体步骤如下：

中断描述符表寄存器将通过中断描述符表，跳转找到 page_fault 函数去执行。在此函数中，只有可能继续调用两个函数，一个是“缺页中断”服务程序 do_no_page，另一个是“页写保护”服务程序 do_wp_page；在当前情况下，经过分析得之，下面将要调用“缺页中断”服务程序 do_no_page 函数去执行，如图 2-105 所示：

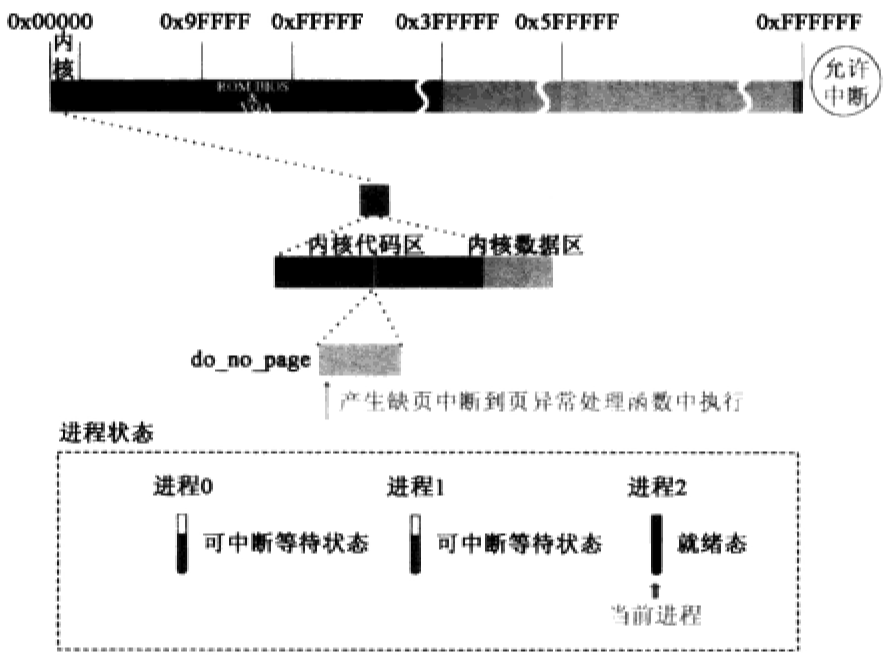


图 2-105　发生缺页中断

具体执行代码如下：

```
// 代码路径：mm/page.s:
_page_fault:
    ...
    testl $1,%eax
    jne 1f
1:  call _do_no_page
...
```

2.7.22　缺页中断中 shell 程序加载前的检测

进入 do_no_page 函数后，开始 shell 程序的加载工作。系统先要进行两方面的检测，然后再正式把 shell 程序的内容从虚拟盘上读进指定页面，检测工作将分为两个方面。

一方面是：缺页中断产生后，并不必然意味着确实需要把外设上的内容拷贝进新申请的内存页面中去。因此，这个检测是必须的，因为系统中有其他一些情况同样会导致缺页，比如程序执行压栈时需要新的栈空间，也会产生缺页中断，那么系统仅给它申请新的内存页面就足够了，与外设上的数据没有任何关系，等等。具体执行代码如下：

```
// 代码路径：mm/memory.c:
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
address &= 0xfffff000;
    tmp = address - current->start_code;
    if (!current->executable || tmp >= current->end_data) {
        get_empty_page(address);
        return;
     }
    ...
}
```

另一方面是：检测 shell 程序是否可以与其他进程共享内存页面。由于 Linux 0.11 是多进程操作系统，所以其他进程此时也有可能与当前进程执行同一个程序，调用 share_page 函数来判断是否有可能让当前进程与那个进程共享同一个程序，如果能够共享，就不用再从外设上重新载入代码了。具体执行代码如下：

```
// 代码路径：mm/memory.c:
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
if (share_page(tmp))
        return;
    ...
}
```

很显然，两次检测的结果都不会导致程序返回，所以确实要新申请一个内存页面，并将 shell 程序从虚拟盘上载入。如图 2-106 所示。

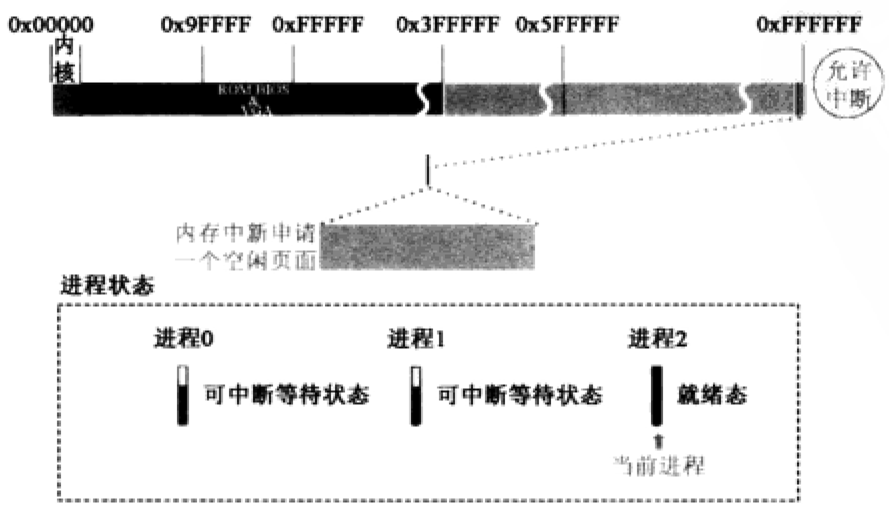


图 2-106　申请空闲页面

2.7.23　为即将载入的内容申请页面

这里要调用 get_free_page 函数为即将载入的 shell 内容申请内存页面，而且系统还要对申请的结果进行检测，如果申请不到内存页面，比如说内存不够用，当前进程则退出。当然，此时系统还是能够申请到内存页面的，但以后发生缺页中断时都是由这个函数来处理，那时就不一定了，所以这个检测是必须的。新申请的页面如图 2-107 右上角所示，shell 程序就准备载入这个新获得的页面。

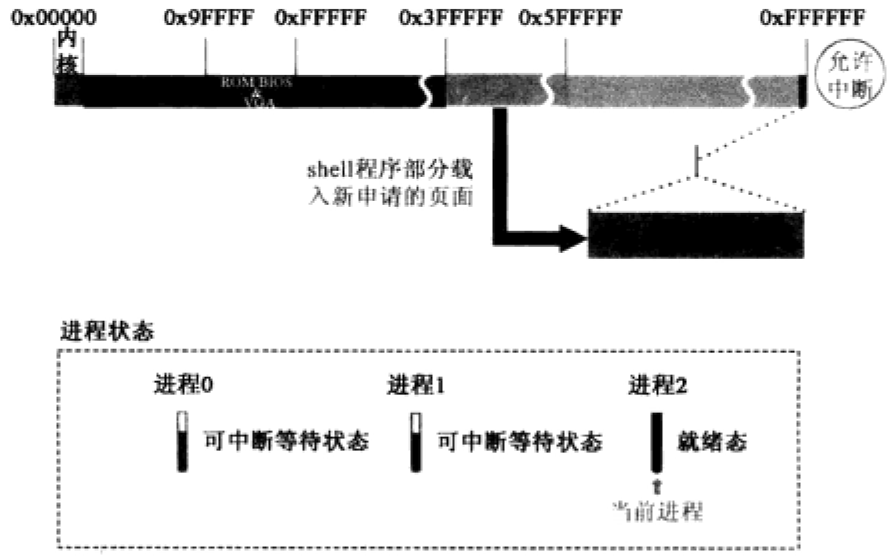


图 2-107　缺页中断载入 shell 程序

具体执行代码如下：

```
// 代码路径：mm/memory.c:
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
if (!(page = get_free_page()))
        oom();
...
}
```

2.7.24　将 shell 程序载入新获得的页面

这里执行加载 shell 程序的动作。因为一个页面的大小为 4KB，所以一次缺页中断发生就要把 4KB 大小的程序内容读进这个页面中，具体步骤如下：

调用 bmap 函数，根据 shell 程序的 i 节点提供的信息，读取数据块在设备上对应的所有逻辑块号（一共是 4 个，必须全部读取出来）。具体执行代码如下：

```
// 代码路径：mm/memory.c:
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
for (i=0 ; i<4 ; block++,i++)
        nr[i] = bmap(current->executable,block);
...
}
```

调用 bread_page 函数，从虚拟盘上把 nr[4] 指定的 4 个 shell 程序逻辑块载入缓冲区，并最终复制到内存的指定空间内，具体执行代码如下：

```
// 代码路径：mm/memory.c:
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
bread_page(page,current->executable->i_dev,nr);
...
}
```

bread_page 函数的执行步骤与本章 2.3.3 节至 2.3.8 节中讲到的 bread 函数执行步骤大体一致，前面已经详细说明了。

这 4KB 的程序 shell 可以先使用，如果后续的执行过程中又需要执行新的程序，则会再次产生缺页中断，并按照 do_on_page 函数中同样的方式将另外 4KB 的内容载入，这就是进程所对应的程序“按照需求进行加载”的具体方法。

2.7.25 根据 shell 程序的情况，调整页面的内容

shell 程序的内容已经载入了，内存中增加了一个物理页面，但进程的线性地址空间内并没有与此物理页面对应的关系，而进程执行时的寻址是线性地址，没有对应关系就无法通过线性地址映射到具体的物理内存地址上，程序也就无法正常运行。所以，接下来还要把这个物理页面映射到进程的线性地址空间内。

在这个映射动作之前，我们先关注一下系统对 shell 程序新页面的一个处理动作。此时系统将检测：如果加载进内存的程序内容小于一个页面（尽管目前加载的 shell 程序不是这样，但是 do_no_page 适用于所有程序执行过程中产生缺页中断的情况，有的比较小的程序代码本身就不到 4KB，所以要进行检测），就将页面中剩余的空间清 0，这部分空间在程序执行起来（比如压栈等动作发生时）有可能会用到。如图 2-108 所示。

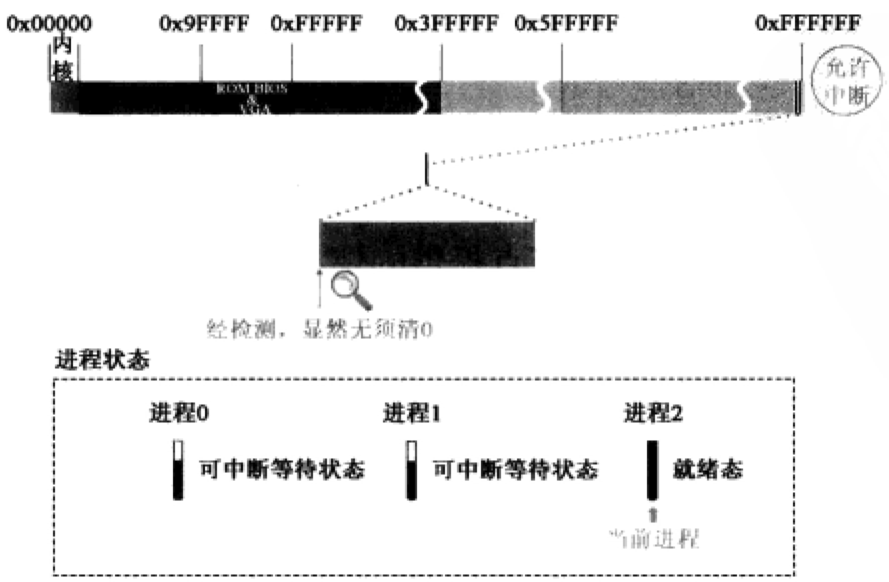


图 2-108 根据进程管理结构信息调整页面内容

具体执行代码如下：

```
// 代码路径：mm/memory.c:
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
i = tmp + 4096 - current->end_data;
    tmp = page + 4096;
    while (i-- > 0) {
        tmp--;
        *(char *)tmp = 0;
    }
...
}
```

2.7.26　将线性地址空间与程序所在的物理页面对应

接下来系统将完成此次 shell 程序加载的最后一步，即将程序内容所在的物理页面映射到线性地址空间内。在此过程中，线性地址将被执行“页目录项”、“页表”和“页面”这三级解析，使线性地址和物理地址建立关系。值得注意的是，在 2.7.15 节中我们已经介绍过，进程 2（也就是现在的 shell 进程）解除了与原来的页表及页面的关系，现在则是对这个关系的重建，而且是针对 shell 程序的重建，建立了映射关系之后的物理内存与线性地址的关系，如图 2-109 所示：

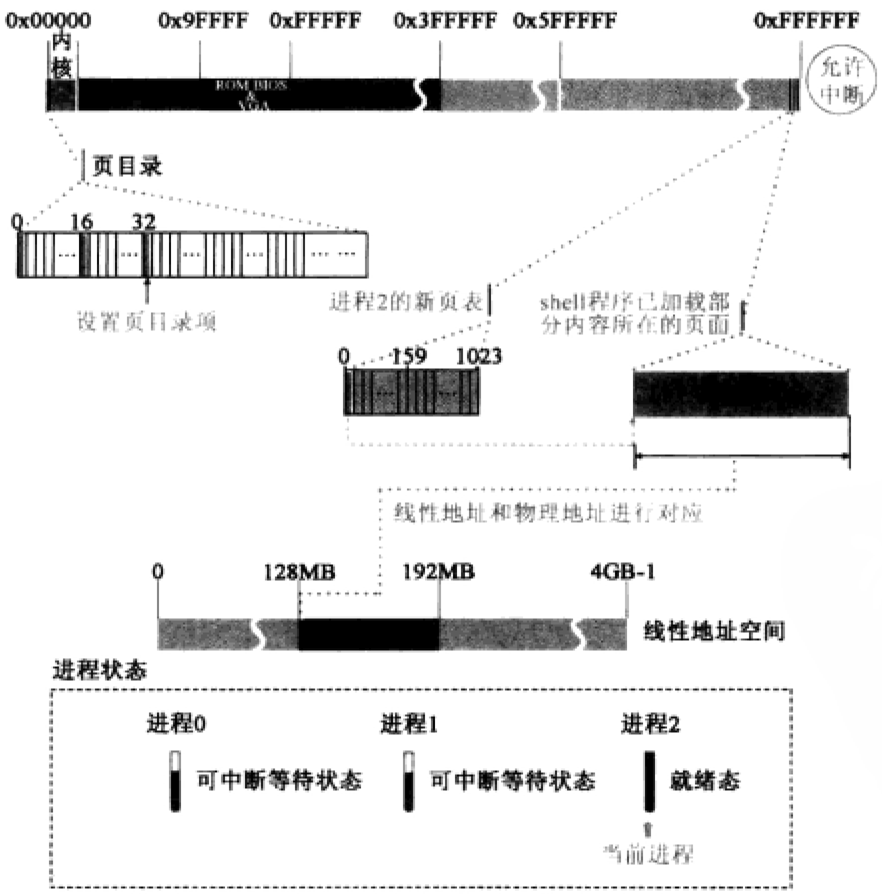


图 2-109　线性地址与物理地址对应

具体执行代码如下：

```
// 代码路径：mm/memory.c:
void do_no_page(unsigned long error_code,unsigned long address)
```

```
{
    ...
put_page(page,address)
...
}
```

到此为止，已经将 shell 程序加载前的准备工作以及第一次加载的具体过程全部介绍完了。从 2.8 节开始，我们将详细介绍系统达到怠速前的最后工作。

2.8 系统实现怠速

通过第 2.7 节中的介绍我们得知，shell 进程已经有部分内容载入到了内存中，并具备了执行的能力。本节将讲述，从这个 shell 程序开始执行，一直到系统达到怠速状态的全部过程。这个过程分为两个阶段：第一个阶段是 shell 程序开始执行后，创建一个新的进程，然后 shell 进程退出并切换到进程 1 去执行；第二个阶段是进程 1 重新创建 shell 进程，并由此构建人机交互界面，使系统达到怠速状态。下面我们来具体地讲解。

本节内容执行的程序架构图如图 2-110 所示：

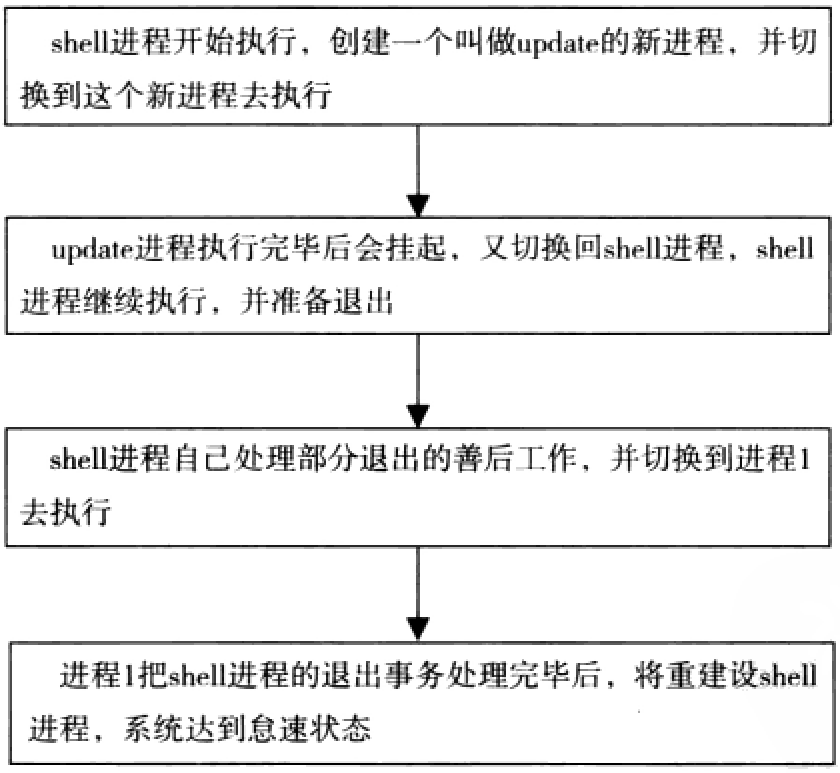


图 2-110　第 2.8 节的程序架构图

2.8.1 shell 进程准备创建 update 进程

这里介绍本节第一阶段工作：shell 进程将准备创建一个新的进程。shell 程序开始执行后要对输入设备上的数据信息进行读取，为此，我们在介绍 shell 程序的执行之前先来关注一下此时输入设备的情况。

在 2.7.2 节中我们得知，shell 进程的管理结构中 *filp[20] 结构的第一项与系统中文件管理表

file_table[64] 中表示 tty0 的这个表项之间解除了关系，并重新打开了一个新的文件 "/etc/rc"，使 shell 进程的管理结构中的 *filp[20] 这个结构的第一项与系统中文件管理表 file_table[64] 建立了一个关于此新文件的关系，这个文件现在将作为输入设备被使用。下面，我们来看一下当 shell 从这个文件中读取信息时会发生什么。

2.8.2 进程 2 开始执行 /etc/rc 文件

继续介绍本节第一阶段的工作，shell 程序要读取一个脚本文件中的命令，根据命令来创建进程。“/etc/rc” 文件是一个存储于虚拟盘中的脚本文件，脚本文件的特点是，文件中记录着一些命令，这些命令可以指示应用程序的执行。

因此，shell 从 "/etc/rc" 脚本文件中读取了一些命令，其中主要包括以下两条命令：

```
/ etc/update &
```

和

```
echo "/dev/hd1 /" >/etc/mtab
```

这些命令被读进内存后（读文件操作的具体步骤将在后续章节中详细介绍），shell 将对它们进行解析，这些命令将引导 shell 程序下一步的执行，下面我们来看对第一条命令的解析。如图 2-111 所示。

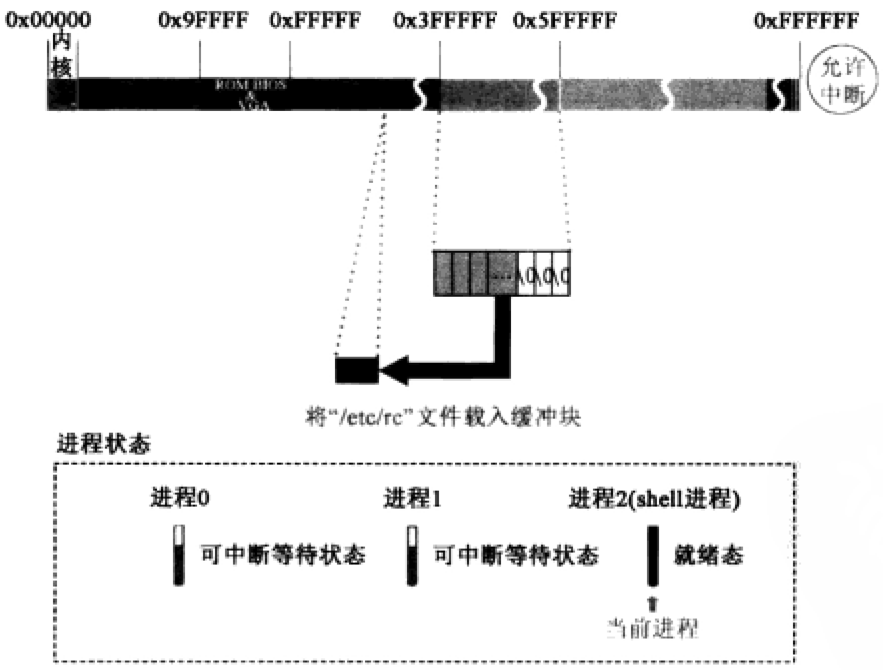


图 2-111 将“/etc/rc”文件载入缓冲区

2.8.3 准备加载 update 进程

继续介绍本节第一阶段工作，shell 程序根据脚本中的命令创建出了新的进程，并将执行权转交给新进程。shell 解析完 / etc/update & 这条命令后将会创建一个新的进程，并将 update 这个可执行文件的程序载入，使之执行。于是，shell 创建一个新进程，通过 2.2.2 节中的介绍我们

可以得知，这个新进程的进程号将是 3（shell 进程的进程号是 2，所以依次累加，它的进程号就是 3），它在进程槽 task[64] 这个结构中的“项号”也是 3，我们后面一律称之为“update 进程”。创建完毕后，shell 进程将加载 update 这个可执行文件的执行程序到内存中（程序加载方式与 2.7 节中 shell 程序的加载方式完全一致，都是通过调用 execve 函数来实现的），并最终将执行权转交给 update 进程，由它去执行。下面我们来对 update 进程的情况进行介绍。效果如图 2-112 所示。

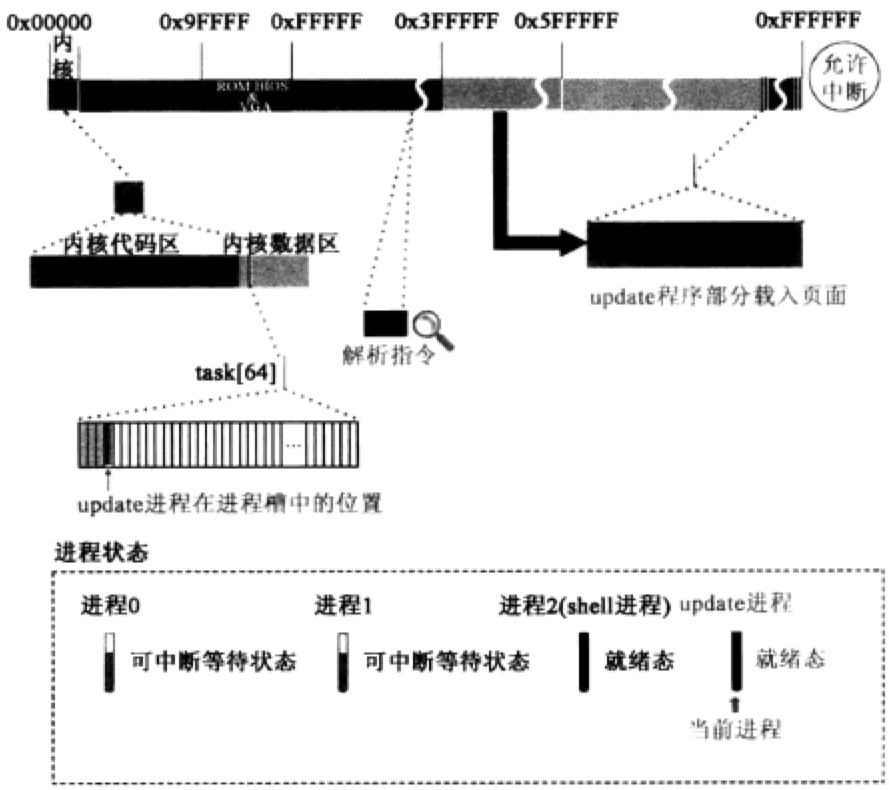


图 2-112　update 进程的状态

2.8.4　update 进程的作用

继续介绍本节第一阶段的工作，shell 程序创建的新进程执行完毕后，又切换到 shell 进程去执行。update 进程有一项很重要的任务：将缓冲区中的数据同步到外设（如：软盘、硬盘）上。在 Linux 0.11 中，当进程需要往外设文件上写数据的时候，为了提高系统的整体执行效率，并不把数据直接写入到外设上，而是先写入到缓冲区中。之后，再将数据从缓冲区中输入到外设上，以便缓冲区中的数据与外设上的数据一致，从而真正的实现“写文件操作”，这个输入的过程就叫做“同步”。

每隔一段时间，这个进程就会被唤醒，并把数据往外设上同步一次，同步工作完成后，这个进程会被挂起，即被设置为可中断等待状态，等待着下一次被唤醒后继续执行，如此周而复始。这样做是因为，每个进程只要参与轮询，当轮到它执行时，都会消耗时间，这将影响到其他进程执行的总效率。如果这个进程要执行的内容是重要且紧急的，那么这个进程就必须参与轮询，而 update 进程要做的同步工作是重要的，但并不紧急，也就是说，缓冲区中的内容，并不是急需往硬盘上同步，所以它可以被暂时挂起，以此来提高所有进程运行的总效率。

现在该 update 进程执行了，它执行完毕后，将会被挂起，并继续切换到 shell 进程去执行。如图 2-113 所示。update 进程将常驻于系统中，随时准备将缓冲区中的数据同步到外设上。下面，我们来介绍切换到 shell 进程后，它将如何执行。

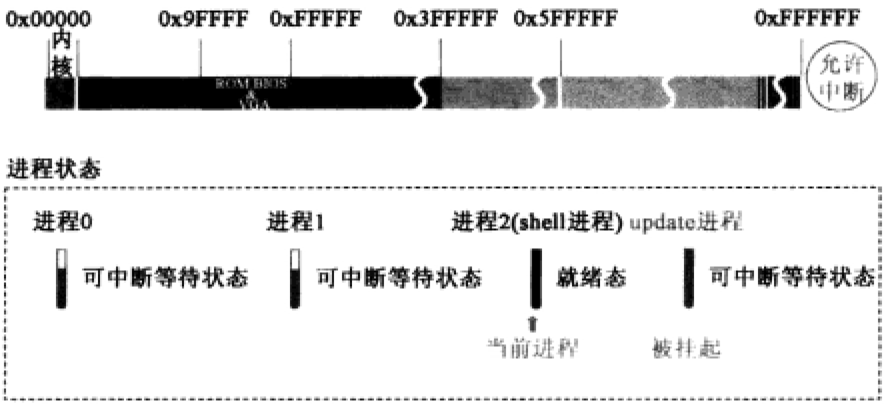


图 2-113　进程状态变化

2.8.5　shell 程序检测 “/etc/rc” 文件

通过对 shell 进程执行的规则进行分析可以得知，shell 一旦开始执行，就去检测输入设备上是否有数据要读取，检测输入设备的本质其实就是检测 shell 进程管理结构中 *filp[20] 这个结构的第一项与系统中文件管理表 file_table[64] 建立关系的文件。在这里就是检测 "/etc/rc" 这个文件，如果该文件中有数据，就去调用 read 函数，并最终映射到系统调用函数 sys_read 去读取，执行代码如下：

```
// 代码路径：fs/read_write.c:
int sys_read(unsigned int fd,char * buf,int count)
{
    ...
    if (inode->i_pipe)
        return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
    if (S_ISCHR(inode->i_mode))
        return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
    if (S_ISBLK(inode->i_mode))
        return block_read(inode->i_zone[0],&file->f_pos,buf,count);
    if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
        if (count+file->f_pos > inode->i_size)
            count = inode->i_size - file->f_pos;
        if (count<=0)
            return 0;
        return file_read(inode,file,buf,count);
    }
   printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
   return -EINVAL;
}
```

然后，shell 程序将根据返回值来确定下一步如何执行，这就是 shell 执行的规则。

现在情况是这样的：

由于 "/etc/rc" 文件是普通文件，所以会执行如下代码：

```
// 代码路径：fs/read_write.c:
int sys_read(unsigned int fd,char * buf,int count)
 {
    ...
    if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
        if (count+file->f_pos > inode->i_size)
            count = inode->i_size - file->f_pos;
        if (count<=0)
            return 0;
        return file_read(inode,file,buf,count);
    }
    ...
}
```

在文件读取结束后，返回值应该是 0（文件读取的具体操作步骤将在后续的文件操作章节中详细介绍），这个返回值将导致 shell 进程退出，下面就来介绍 shell 程序退出的具体过程。

2.8.6 shell 进程退出

继续介绍本节第一阶段工作，这里着重介绍 shell 进程退出过程中与“解除其程序占用内存页面关系”相关的步骤。shell 程序会调用 exit 函数执行退出，于是产生软中断，并最终映射到 sys_exit 函数去执行，这个函数将调用 do_exit 函数，执行代码如下：

```
// 代码路径：kernel/exit.c:
int sys_exit(int error_code)
{
    return do_exit((error_code&0xff)<<8);
}
```

进入 do_exit 函数后，开始为 shell 的退出做善后工作，先要释放当前进程代码段和数据段所占的内存页面和页表，执行代码如下：

```
// 代码路径：kernel/exit.c:
int do_exit(long code)
{
    ...
    free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
    free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    ...
}
```

这样，shell 程序所占用的内存页面的引用次数将被减 1，对于 shell 占用的绝大多数页面来讲，这些页面的引用次数将被削减为 0，于是将被系统视为空闲页面而具备了重新使用的能力。如图 2-114 所示。

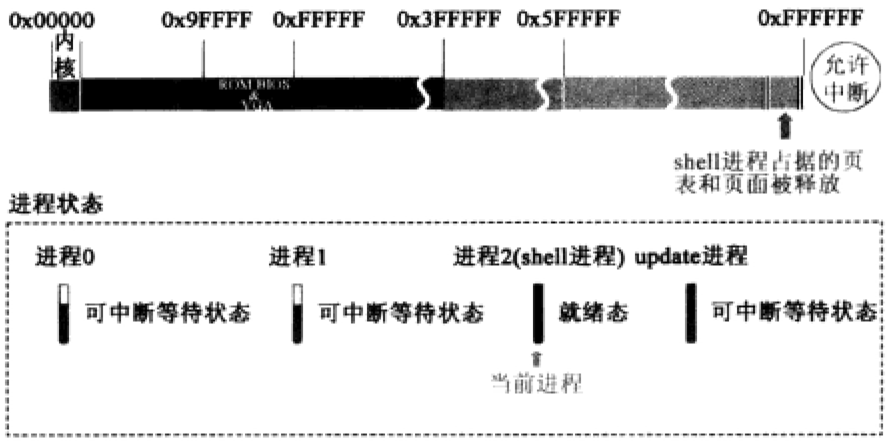


图 2-114　释放 shell 进程占据的页表和页面

2.8.7　shell 进程退出善后处理

继续介绍本节第一阶段的工作，这里着重介绍 shell 进程退出过程中，与“重新设置 shell 进程管理结构以及重新选定其子进程的父进程”相关的步骤。这里继续为 shell 进程做善后处理工作。先判断这个 shell 是否有子进程，我们在 2.8.3 节中已经介绍过，shell 确实创建了一个子进程，即 update 进程，现在 shell 进程要退出了，这就意味着这个子进程的父进程即将不存在了，所以要给它重新找一个父进程，于是将这个 update 进程的父进程设置为进程 1，执行代码如下：

```
// 代码路径：kernel/exit.c:
int do_exit(long code)
{
    ...
    task[i]->father = 1;
    ...
}
```

之后，还要针对 shell 进程的管理结构进行处理，比如关闭当前进程打开着的所有文件，对当前进程的工作目录 pwd、根目录 root 及 shell 程序所对应的可执行文件的 i 节点 executable 执行同步操作，并分别置空（释放），还要将这个 shell 进程的状态设置为僵死，等等。这样设置是因为 shell 进程要退出了，与之对应的文件关系全都需要解除，它的状态也不再是就绪态，执行代码如下：

```
// 代码路径：kernel/exit.c:
int do_exit(long code)
{
    ...
    for (i=0 ; i<NR_OPEN ; i++)
        if (current->filp[i])
            sys_close(i);
   iput(current->pwd);
   current->pwd=NULL;
   iput(current->root);
   current->root=NULL;
   iput(current->executable);
```

```
    current->executable=NULL;
    if (current->leader && current->tty >= 0)
         tty_table[current->tty].pgrp = 0;
    if (last_task_used_math == current)
         last_task_used_math = NULL;
    if (current->leader)
         kill_session();
    current->state = TASK_ZOMBIE;
    current->exit_code = code;
     ...
}
```

最后，调用 tell_father 函数向它的父进程——进程 1 发送“子进程终止”信号，并进行进程切换。如图 2-115 所示。

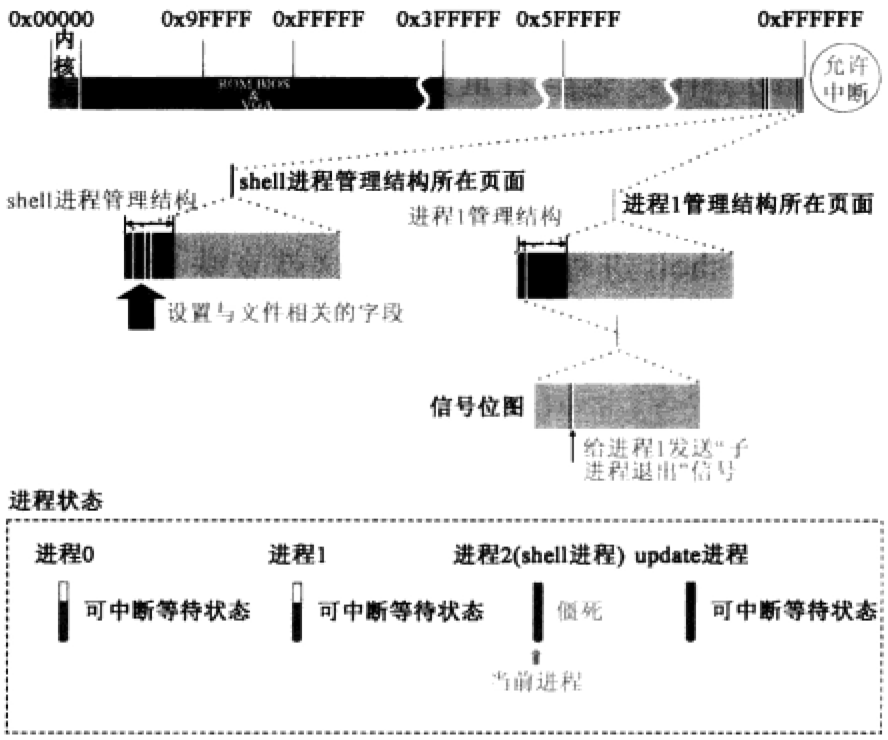


图 2-115　shell 进程退出善后处理

具体执行代码如下：

```
// 代码路径：kernel/exit.c:
int do_exit(long code)
{
     ...
     tell_father(current->father);
     schedule();
     ...
}
```

到这里为止，shell 进程的善后工作基本完成了，还剩下它自身的进程管理结构所占用的页面没有被清理掉，这个清理工作，将由它的父进程——进程 1 来完成。图 2-116 表示了进程 1 清理 shell 进程管理结果所占页面以及 shell 进程重新加载的流程。

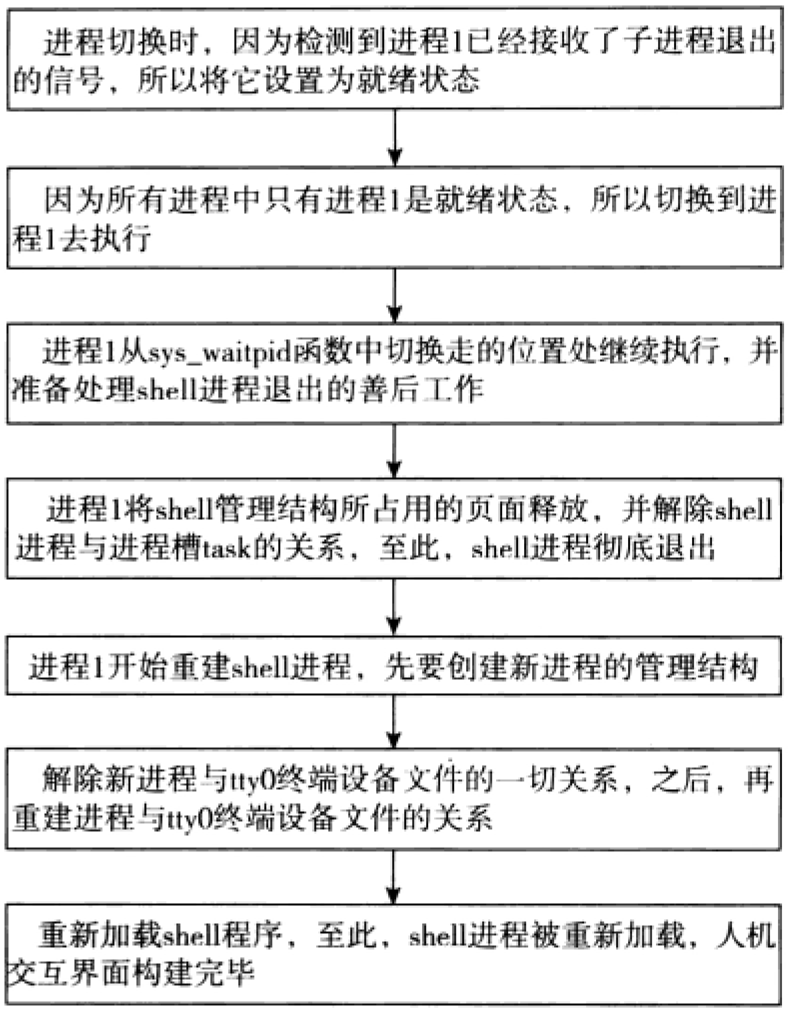


图 2-116 shell 进程如何重新加载

2.8.8 进程 1 清理 shell 进程管理结构

继续介绍本节第一阶段工作，shell 进程的部分退出工作完毕后，将切换到进程 1 执行，进程 1 将彻底把 shell 的退出工作处理完毕，并准备重建 shell 进程。shell 进程最后执行进程切换，于是进入 schedule 函数去执行，代码如下：

```
// 代码路径：kernel/sched.c:
void schedule(void)
{
    ...
    if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING
    ...
}
```

在 2.8.7 节中讲过，shell 进程给进程 1 发送了信号，而进程 1 又是可中断等待状态（理由在本章第 2.6.3 节中已经说明），所以在第一次遍历所有进程时，上述代码中的 if 语句执行结果将为真，于是进程 1 将被设置为“就绪状态”。

下面在进行第二次轮询的时候，由于系统中现在只有进程 1 是就绪态，所以将会切换到进程 1 去执行，执行代码如下：

```
// 代码路径：kernel/sched.c:
void schedule(void)
{
    ...
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    switch_to(next);
}
```

通过 2.6 节的介绍我们得知，进程 1 是在执行 sys_waitpid 函数时切换到进程 2 的，所以切换到进程 1 后，还要在 sys_waitpid 函数中继续执行，对应代码如下：

```
// 代码路径：kernel/exit.c:
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    ...
    if (!(current->signal &= ~(1<<(SIGCHLD-1))))
            goto repeat;
    ...
}
```

这两行代码的意思是，如果进程 1 等待的子进程确实退出了，就跳转到 repeat 标号处执行，这样，sys_waitpid 函数的主体内容将再次被执行一次，对应代码如下：

```
// 代码路径：kernel/exit.c:
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    ...
    repeat:
    ...
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
    ...
    if ((*p)->father != current->pid)
            continue;
    ...
        switch ((*p)->state) {
            ...
            case TASK_ZOMBIE:
                current->cutime += (*p)->utime;
                current->cstime += (*p)->stime;
```

```
                flag = (*p)->pid;
                code = (*p)->exit_code;
                release(*p);
                put_fs_long(code,stat_addr);
                return flag;
            ...
        }
    }
    ...
}
```

这里要遍历所有的进程，并从中查找哪个进程是进程 1 的子进程，而且它的状态是僵死状态，很显然，只能是进程 2，所以，接下来要做两件重要的事情：

1. 将 shell 进程的善后工作处理完

通过执行 release(*p); 这一行代码，将进程 2 退出的善后工作彻底完成，该函数代码展开为：

```
// 代码路径：kernel/exit.c:
void release(struct task_struct * p)
{
    ...
    for (i=1 ; i<NR_TASKS ; i++)
        if (task[i]==p) {
            task[i]=NULL;
            free_page((long)p);
            schedule();
            return;
        }
    ...
}
```

通过 task[i]=NULL; 这一行我们可以看出，shell 与进程槽的关系解除了，进程槽这一项就成了空闲项；通过 free_page((long)p); 这一行，将 shell 进程的管理结构所占用的页面释放掉，到这里为止，shell 进程退出的所有善后工作全都结束了，shell 这个进程在系统中将不复存在。如图 2-117 所示。

2. 将 shell 进程的进程号返回

即执行 return flag;，这个进程号应该是 2。这样，sys_waitpid 函数就返回了，接下来我们来看一下返回后进程 1 又该如何执行，先来看一段代码：

```
// 代码路径：init/main.c:
    if (!(pid=fork())) {
        close(0);
        if (open("/etc/rc",O_RDONLY,0))
            _exit(1);
        execve("/bin/sh",argv_rc,envp_rc);
        _exit(2);
    }
    if (pid>0)
        while (pid != wait(&i))
    while (1)
```

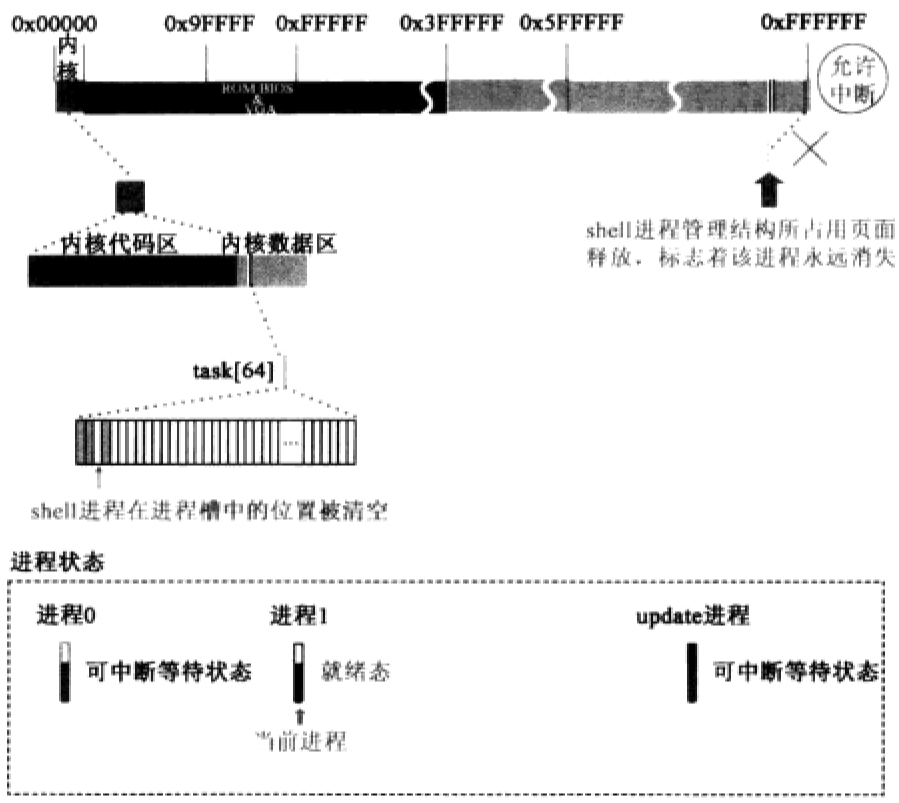


图 2-117　进程 1 撕毁 shell 进程的户口本

这段代码是 main.c 中的代码，在 2.6 节中我们介绍过，进程 1 等待进程 2 退出就是调用了 wait 函数，见这一行代码 while (pid != wait(&i))，既然 sys_waitpid 函数返回了，那么 wait 函数也就该返回了，而且返回值是“2”。下面我们来关注一下此时 pid 的值是多少，在 2.5 节中介绍过，进程 1 创建进程 2 执行的就是 if (!(pid=fork())) 这一行，当时返回值肯定是进程 2 的进程号，即“2”，所以 pid 的值是 2，于是 while (pid != wait(&i)) 这一行相当于在判断 while (2!=2)，结果必然导致循环退出，并最终开始执行 while (1) 这一行代码。

到这里为止，本节第一阶段的内容就全部执行完毕了，shell 进程在第一次加载完毕后做完了它应该做的工作，最后它退出了，并切换到进程 1，接下来将介绍本节第二阶段的工作，进程 1 将重建 shell 进程。

2.8.9　系统开始重建 shell

这里开始介绍本节第二阶段的内容，进程 1 将在此重建进程 2。执行代码如下：

```
// 代码路径：init/main.c:
    while (1) {
        if ((pid=fork())<0) {
            printf("Fork failed in init\r\n");
            continue;
        }
        if (!pid) {
            close(0);close(1);close(2);
            setsid();
        (void) open("/dev/tty0",O_RDWR,0);
            (void) dup(0);
            (void) dup(0);
                _exit(execve("/bin/sh",argv,envp));
```

```
    }
    while (1)
        if (pid == wait(&i))
            break;
    printf("\n\rchild %d died with code %04x\n\r",pid,i);
    sync();
}
```

这部分代码在 main.c 中先执行 if ((pid=fork())<0) 这一行，这一行的意思是新建一个进程，新建这个进程的步骤与进程 0 创建进程 1，以及进程 1 创建进程 2 完全一致，这已经在 2.2 节和 2.5 节中讲过。但是，这里值得注意的是这个新进程的进程号和在进程槽 task[64] 结构中的“项号”。

通过 2.2.2 节中的介绍我们得知，进程号是通过 last_pid 这个全局变量的累加而得出的，现在的情况是，在 2.8.3 节中，shell 进程创建 update 进程时，update 进程的进程号是 3，这表明 last_pid 已经被累加为 3 了，那么现在进程 1 再次创建新的进程，last_pid 继续累加，所以这个新进程的进程号应该就是 4。

通过 2.8.8 节中对 release 函数的介绍我们得知，shell 进程与进程槽的关系已经解除，所以这个进程槽现在就应该是空闲的，这个新建进程在进程槽中的项号，就应该是 shell 进程原来解除关系那一项的项号，所以，应该是第 3 项。如图 2-118 所示。

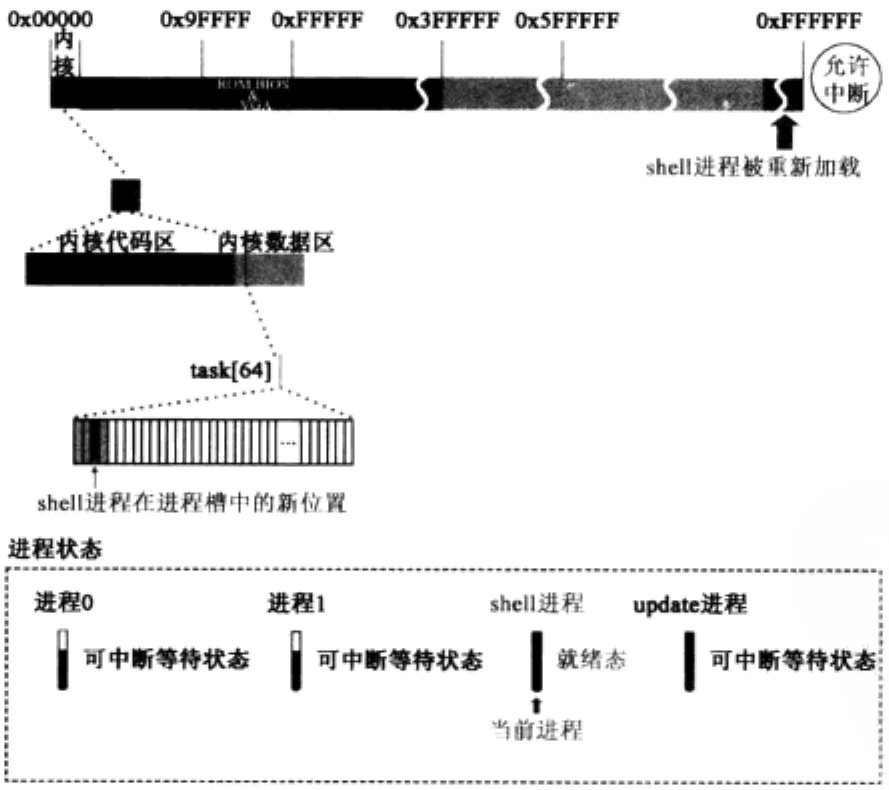


图 2-118 重建 shell 进程

新进程创建完毕后，进程 1 将会等待这个新进程的退出，执行代码如下：

```
// 代码路径：init/main.c:
while (1)
    if (pid == wait(&i))
```

这个 wait 函数的执行过程已经在 2.6 节中说明。于是，进程 1 再次被设置为可中断等待状

态，并切换到进程 2 执行。

进程 2 开始执行后，会执行到 if (!pid) 中去，要将原来的输入设备、标准输出设备和标准错误输出设备全部清除，执行代码如下：

```
// 代码路径：init/main.c:
    while (1) {
        ...
        if (!pid) {
            close(0);close(1);close(2);
            setsid();
            (void) open("/dev/tty0",O_RDWR,0);
                (void) dup(0);
                (void) dup(0);
                    _exit(execve("/bin/sh",argv,envp));
        }
    ...
}
```

之后，重新打开 tty0 这个终端设备文件并复制其句柄，执行代码如下：

```
// 代码路径：init/main.c:
    while (1) {
        ...
        if (!pid) {
            ...
            (void) open("/dev/tty0",O_RDWR,0);
                (void) dup(0);
                (void) dup(0);
                    _exit(execve("/bin/sh",argv,envp));
        }
    ...
}
```

以此作为最新的输入设备、标准输出设备和标准错误输出设备。这些步骤我们在本章第 4 节中已经说明。

最后，重新加载 shell 进程，执行代码如下：

```
// 代码路径：init/main.c:
execve("/bin/sh",argv,envp),
```

这回加载的 shell 进程将真正成为人机交互界面，到这里为止，系统也就达到怠速状态了。本章第二阶段的内容介绍完毕了。

2.8.10 shell 进程为何不会再次退出

shell 开始运行后，会出现一个疑问：同样的 shell 程序，为什么第一次执行时会创建一个新进程并使自己退出，而 shell 怠速后再执行时就不会出现这种情况呢？

答案是这样的：我们现在再仔细回过头去看一下 2.8.5 节中所介绍的 shell 程序执行的规则，这个程序执行后，一定要从标准输出设备中读取数据，并分析执行。这个标准输出设备就是 shell 进程管理结构中 *file[20] 结构的第一项与系统中文件管理表 file_table[64] 建立关系的文件，

在第一次加载 shell 程序时，这个文件是 "/etc/rc"，它的属性是一个“普通文件”，读此文件时，执行的代码如下：

```
// 代码路径：fs/read_write.c:
int sys_read(unsigned int fd,char * buf,int count)
 {
     ...
     if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
         if (count+file->f_pos > inode->i_size)
             count = inode->i_size - file->f_pos;
         if (count<=0)
             return 0;
         return file_read(inode,file,buf,count);
     }
     ...
}
```

它的返回值将导致 shell 程序退出，这一点在 2.8.5 节中已经说明，但是第二次加载 shell 时情况不同了，代表标准输出设备的文件换成了 tty0，这一点在 2.8.9 节的最后一段已经介绍了，这个 tty0 不是普通文件，而是设备文件。所以，同样是调用 sys_read 函数，执行代码就变成了：

```
// 代码路径：fs/read_write.c:
int sys_read(unsigned int fd,char * buf,int count)
 {
     ...
     if (S_ISCHR(inode->i_mode))
         return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
     ...
}
```

这样，在进入 rw_char 函数后，shell 进程将被设置为可中断等待状态，即便将来该函数执行完毕，返回值也不会导致 shell 退出，所以，两次同样是加载 shell，而最终的执行结果却完全不同。这也体现了字符设备文件与普通文件的区别。

怠速以后，shell 进程要通过 tty0 这个字符设备文件上的数据信息来确认自己下一步将如何执行。具体的工作原理是这样的：系统将所有外设都抽象成了“文件”，这就使得每个进程在与外设进行交互时都等同于与“文件”进行交互，这一点对于终端设备文件也不例外，系统将键盘抽象成了“设备文件”，而 tty0 就是这种设备文件，它是一种特殊的文件。具体表现为它只有 i 节点，却没有实际的内容存储在虚拟盘上，它会和系统中的一个字符缓冲队列相挂接，用户通过键盘输入的字符数据就存储在这个字符缓冲队列上，这些就相当于是 tty0 文件的“内容”了。

shell 进程一旦开始执行就会从 tty0 中读取数据，这就相当于从这个字符缓冲队列中读取数据，以此检测用户是不是通过键盘下达了指令。如果没有读取到任何数据，就证明用户并没有通过键盘下达指令，于是 shell 进程将会被设置为可中断等待状态，即被挂起。当用户通过键盘下达指令时，一方面，敲击产生的字符信息将存储在字符缓冲队列上；另一方面，也必将产生键盘中断，键盘中断产生后会给 shell 进程发信号，这个信号将导致 shell 进程被设置为就绪状态，即被唤醒，唤醒后的 shell 进程一开始执行就去 tty0 文件上读取数据信息，也就相当于到字符缓冲

队列中读取数据信息，读取后进行分析，并根据分析结果执行用户的命令，等命令执行完毕了，tty0 设备文件——也就是字符缓冲队列上的数据信息也就处理完毕了，shell 进程将再次被挂起，等待下一次键盘中断将其唤醒，继续。

2.9 小结

本章继续第 1 章的工作，完成怠速之前的准备，并最终使系统达到怠速状态。具体分为“开中断前的准备”、“进程 0 创建进程 1”、“进程 1 加载根文件系统并创建进程 2”、“进程 2 加载 shell 程序”，以及“shell 程序的重新加载”5 个步骤。

开中断前，操作系统在主机和外设两方面进行了一系列的初始化设置，使得进程 0 具备了执行的能力。开中断后，进程 0 开始工作，创建进程 1 并切换到进程 1 去执行。一旦没有进程可以运行，就由进程 0 来接管执行。进程 1 开始工作后，先加载根文件系统，使操作系统可与外设以文件形式进行数据交互。而后打开终端设备文件，创建进程 2 并切换到进程 2 去执行。进程 2 将加载 shell 程序，shell 程序在启动 update 进程之后退出，由进程 1 接管执行，进程 1 会重新创建新进程并加载 shell 程序，新进程就是 shell 进程，它将开始执行，直到操作系统达到怠速状态。

第 3 章

安装文件系统

在 2.3 节中加载了根文件系统以后，进程已经具备了“依托系统与外设以文件形式进行数据交互”的能力，但此时所指的外设还仅限于根设备，即虚拟盘，进程还不能与硬盘以文件的形式进行数据交互，因为沿着根 i 节点，在虚拟盘上尚无法找到与硬盘上的文件有关的信息。所以，本章要介绍的就是，要让进程“依托系统”与硬盘也能以文件的形式进行数据交互，即安装一个关于硬盘的文件系统，使之与根文件系统建立关系。

安装文件系统分为三步：

（1）将硬盘上的超级块读取出来，并载入到系统的超级块管理结构 super_block[8] 中；

（2）将虚拟盘上指定的 i 节点读出，并将此 i 节点加载到系统的 i 节点管理结构 inode_table[32] 中；

（3）将硬盘上的超级块挂接到 inode_table[32] 中指定的 i 节点上，以此来安装文件系统。

在本书中，70% 以上的内容都会涉及硬盘的文件操作，而文件系统将为此提供最关键的数据和操作支持。

硬盘的文件系统安装成功后，用户程序如果需要操作硬盘上的文件，就可以从根 i 节点开始查找，直至找到 mnt 这个目录文件的 i 节点，之后以此为参照，找到硬盘的超级块，最后再通过硬盘的超级块，一步一步地找到硬盘上指定的文件。硬盘文件系统安装后的效果如图 3-1 所示：

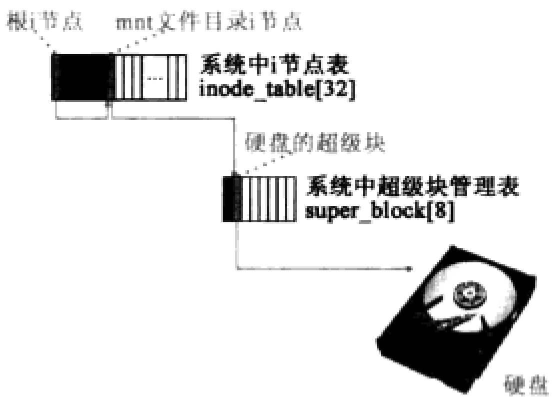


图 3-1　硬盘文件系统安装成功后的示意图

具体的文件查找过程会在第 4 章中详细介绍。

图 3-2 是安装硬盘文件系统的程序架构图：

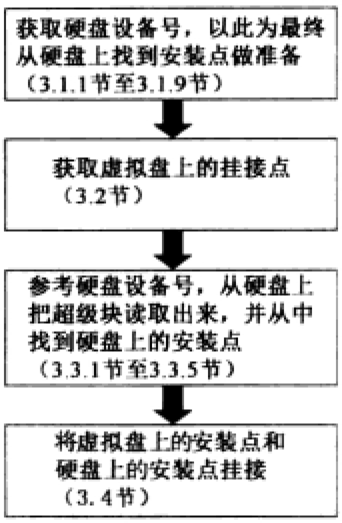


图 3-2　安装硬盘文件系统的程序架构图

3.1　获取硬盘设备号

在硬盘的超级块结构中，有一个成员变量叫做 s_imount，这个点就是安装点。为了得到这个安装点，系统先要把它所在的位置——硬盘中的超级块读进内存中，而要想读出超级块，就要先得到这个硬盘的设备号，本节将介绍如何得到这个硬盘的设备号。

3.1.1　用户发出安装硬盘文件系统指令

在用户终端 shell 下执行命令：mount /dev/hd1 /mnt 来安装文件系统。注意看这个命令中间的两个空格，它们将这个命令分隔成三个参数，分别是“mount”、“/dev/hd1”和“/mnt”，其中“mount”是这个命令的名字，表明这个命令是要安装文件系统；“/dev/hd1”和“/mnt”是两个路径名，它们都是这条命令的参数，整条命令的意思是：将设备“hd1”的文件系统挂接在“mnt”目录文件下。

小贴士

硬盘都是可以分区的，每个分区都可以算做是一个设备，这里我们为了便于研究问题和突出本章的重点，默认整个硬盘就是一个分区，因此 hd1 就代表了硬盘这个设备。

shell 进程接到该命令后，会创建一个新进程，然后该新进程调用 mount 函数产生软中断，并最终映射到 sys_mount 这个系统调用函数中去执行。mount 函数通过软中断映射到 sys_mount 函数的过程，与 fork 函数映射到 sys_fork 函数和 open 函数映射到 sys_open 函数的过程基本类似，已在第 2 章 2.2 节和 2.4 节中说明。上述过程的对应代码如下：

```
// 代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
```

```
{
...
}
```

本章中全部内容都是在 sys_mount 这个函数中实现的，因此，所有的动作处理都由内核完成。

3.1.2 从分析路径开始，准备查找 hd1 设备的挂接点

从这里开始，先为第一个目的做准备，即找到“hd1 这个设备需要拿出来挂接的位置点”，从 hd1 这个硬盘的角度看，只要能够与其“超级块”建立关系，就足以对其文件进行操作了，“超级块上的 s_imount”就是这个“位置点”，因为超级块是硬盘上逻辑块位图和 i 节点位图的总管，是所有线索的根本所在。

系统能从用户输入的命令中得到的唯一线索就是“/dev/hd1”这个路径，接下来系统就要根据这唯一的线索把硬盘的超级块找到，进而最终找到 s_imount；下面与 hd1 相关的所有处理，都是为实现这个最终目的的，接下来，系统就要对这个路径进行解析了，我们看看通过这个解析能得到什么。

调用 namei 函数，准备获取 hd1 这个设备的 i 节点，执行代码如下：

```
//代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
    ...
    if (!(dev_i=namei(dev_name)))
    ...
}
```

namei 函数的执行过程与第 2 章 2.4 节中所介绍的 open_namei 函数大体相同。下面我们只做简单的介绍。

3.1.3 以根目录 i 节点为依托，得到 dev 目录文件的 i 节点

namei 函数进一步调用 dir_namei 函数，通过最终调用 get_dir 函数从“/dev/hd1”这个路径名中解析出枝梢 i 节点，即 dev 这个目录文件的 i 节点，并计算出“/dev/hd1”这个路径名中自 dev 以后的字符长度，为最终获取 hd1 这个设备文件的 i 节点做准备，执行代码如下：

```
//代码路径：fs/namei.c：
static struct m_inode * dir_namei(const char * pathname,
    int * namelen, const char ** name)
{
    ...
    if (!(dir = get_dir(pathname)))
          return NULL;
    basename = pathname;
    while (c=get_fs_byte(pathname++))
        if (c=='/')
            basename=pathname;
```

```
    *namelen = pathname-basename-1;
    *name = basename;
    return dir;
}
```

其中，得到 dev 目录 i 节点的大体步骤如下：

（1）以根设备（即虚拟盘）根目录 i 节点为起点，得到根目录文件；

（2）在根目录文件中查找名为“dev”的目录项；

（3）根据“dev”目录项所提供的信息，最终得到“dev”目录文件的 i 节点内容。

这 3 点与第 2 章 2.4 节中以 "/dev/tty0" 路径名为起点得到“dev”目录文件的 i 节点的内容完全相同，读者可以参考 2.4.2 节到 2.4.3 节的内容。图 3-3 给出了根据根设备的根目录 i 节点，得到 dev 目录文件的 i 节点号的过程，有了 i 节点号，得到 i 节点就顺理成章了。

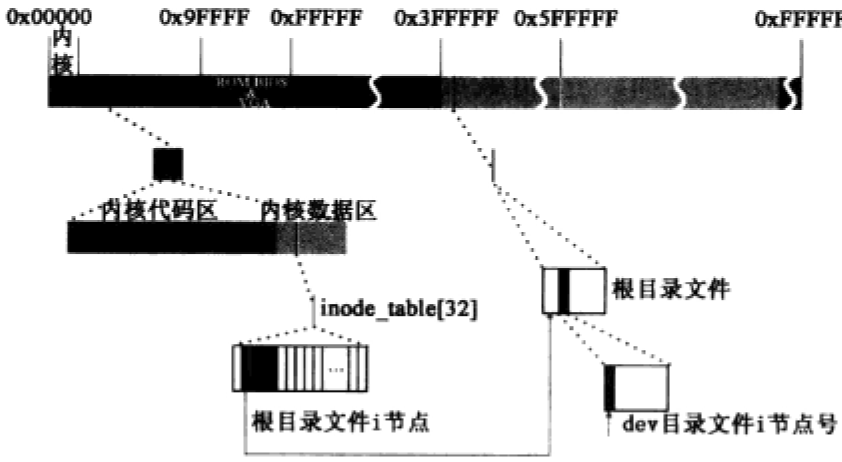


图 3-3　根据 dev 目录项，得到 dev 目录的 i 节点号

3.1.4　从 dev 目录文件中找到代表 hd1 设备文件的目录项

之后，从 dir_namei 函数中返回，namei 函数继续调用 find_entry 函数，根据 dev 目录文件的 i 节点把 dev 目录文件从设备上读入到缓冲块中，以便在其中查找名为“hd1”的目录项，图 3-4 表示了从 dev 目录文件中找到名为“hd1”的目录项的过程，其中 dev 目录文件是根据 3.1.3 节中得到的 dev 目录文件的 i 节点得到的。

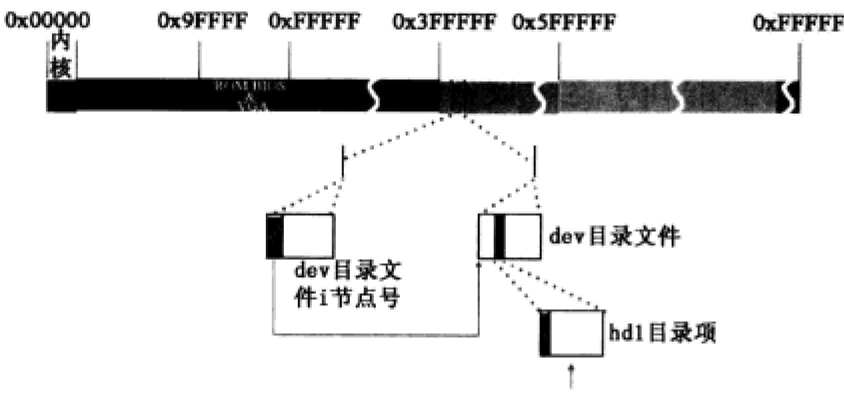


图 3-4　在 dev 目录文件中找到名为“hd1”的目录项

执行代码如下：

```
// 代码路径：fs/namei.c：
struct m_inode * namei(const char * pathname)
{
    ...
    bh = find_entry(&dir,basename,namelen,&de);
    ...
}
```

将 dev 目录文件读出来之后，就可以找到 dev 目录文件里面承载的名为“hd1”的目录项了，目录项一共有两个成员，一个是 hd1 这个字符串本身，另一个是 hd1 这个目录项所对应的目录文件的 i 节点号，而这个 i 节点号将是后面继续能够找到 hd1 这个设备文件的唯一线索，这就是 Linux 0.11 中的文件查找方式，无论路径名有多长，都是这样查找的。

点评

现在虚拟盘的优势体现出来了。由于采取了“空间换时间”的策略，这时只要从虚拟盘中读取就可以了，比从软盘这种外设上读会节省时间。所以，这次操作虚拟盘并没有实质性的和外设打交道，当前进程并不需要等待缓冲区的解锁而被挂起。

3.1.5　得到 hd1 设备文件的 i 节点号

得到了 dev 这个目录文件就可以从中得到 hd1 这个目录项，继而从目录项把其中的另一个成员“i 节点号”取出来。这个 i 节点号就是“hd1 这个设备文件”对应的 i 节点号，之后备份 hd1 这个设备的 i 节点号和虚拟盘的设备号，以便最终将 hd1 这个设备文件的 i 节点读取出来，执行代码如下：

```
// 代码路径：fs/namei.c：
struct m_inode * namei(const char * pathname)
{
    ...
    inr = de->inode;
    dev = dir->i_dev;
    ...
}
```

如图 3-5 表示了 hd1 目录项中提供的 i 节点号与 hd1 设备文件的 i 节点之间的对应关系：

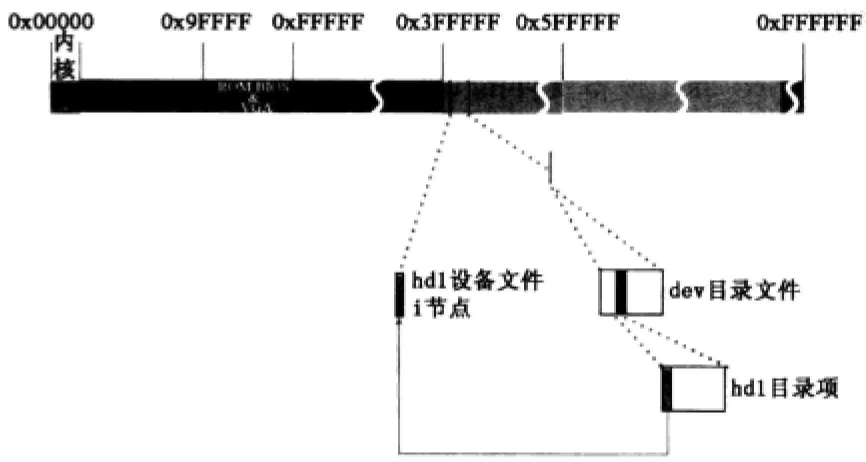


图 3-5　得到 hd1 设备文件的 i 节点号

3.1.6 释放 dev 目录文件的相关内容

备份后，缓冲块中的 dev 目录文件已经没有使用价值，同样，dev 目录文件的 i 节点也没有使用价值了，所以要释放它们，以便节省资源，执行代码如下：

```
// 代码路径：fs/namei.c：
struct m_inode * namei(const char * pathname)
{
    ...
        brelse(bh);
    iput(dir);
    ...
}
```

3.1.7 得到 hd1 设备文件的 i 节点

调用 iget 函数，通过备份的 hd1 这个设备的 i 节点号和虚拟盘的设备号，获取 hd1 这个设备文件的 i 节点，执行代码如下：

```
// 代码路径：fs/namei.c：
struct m_inode * namei(const char * pathname)
{
    ...
    dir=iget(dev,inr);
    ...
}
```

图 3-6 给出了为存储 hd1 设备文件的 i 节点而在 inode_table[32] 中找到的空闲项的位置，从虚拟盘上读出的 hd1 设备文件 i 节点的内容，会被拷贝到该空闲项中以备继续使用：

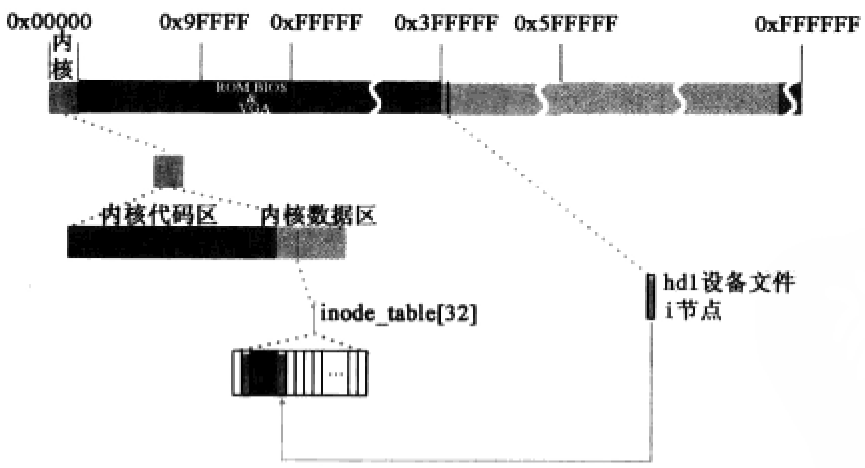


图 3-6　hd1 设备文件的 i 节点在内核 i 节点表的位置

3.1.8 获得 hd1 设备的设备号

namei 函数的工作已经全部完成了，此时已经得到了 hd1 这个设备文件的 i 节点。namei 函数返回后，通过对它的属性进行分析，就可以得到此设备文件对应的设备号，这个设备号就是硬盘的设备号，执行代码如下：

```
// 代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
    ...
    dev = dev_i->i_zone[0];
    ...
}
```

图 3-7 形象地表示了获得 hd1 设备文件设备号的方式，图右下方表示了 hd1 设备文件对应的 i 节点结构，i_zone[0] 位于其中的 i_zone[9] 字段中。

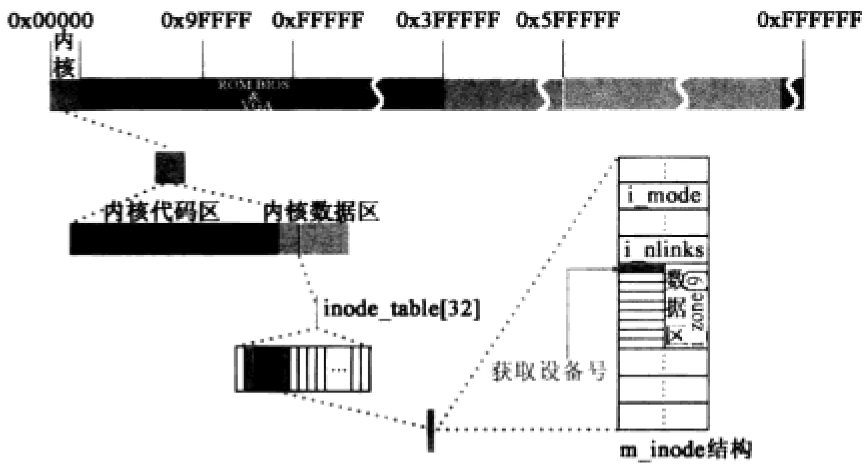


图 3-7 获得 hd1 设备文件的设备号

3.1.9 释放 hd1 设备文件的 i 节点

现在 hd1 这个设备文件的 i 节点已经使用完了，于是释放这个 i 节点，执行代码如下：

```
// 代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
    ...
    iput(dev_i);
    ...
}
```

图 3-8 形象地表示了释放 hd1 设备文件对应的 i 节点的过程：

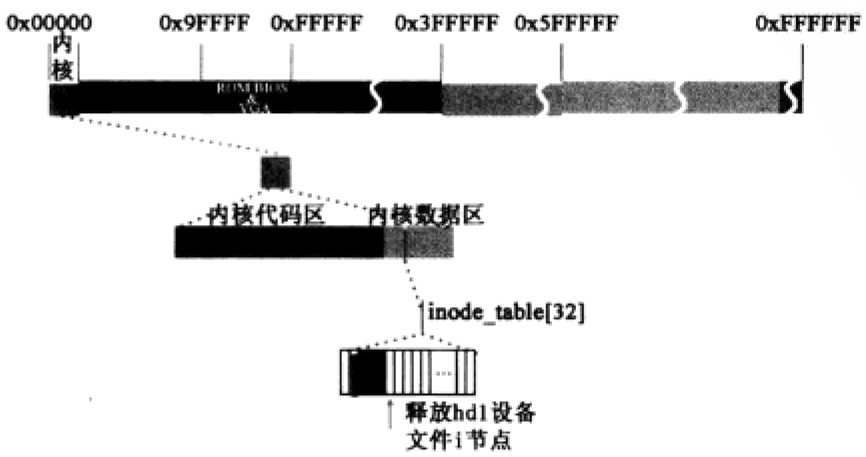


图 3-8 释放 hd1 设备文件 i 节点

到这里为止，已经得到了 hd1 设备文件的设备号，即“hd1 这个设备需要拿出来挂接的位置点”已经确定了，也就意味着已经具备了实现安装文件系统的三个步骤中的第一个步骤的条件。接下来就可以根据已经获得的硬盘的设备号得到硬盘的超级块，得到了超级块，就可以找到超级块中的 s_imount 这个成员，而 s_imount 正是第一步中所讲到的“硬盘上的安装点”。

3.2 获取虚拟盘上的挂接点

硬盘的设备号已经得到了，现在第一步工作暂停，转而开始第二步工作，即从虚拟盘上找到 hd1 文件的挂接点，通过 3.1.1 节中的介绍我们得知，这个安装点是一个名为“mnt”的目录文件，本节就是要通过调用 namei 函数获取 mnt 这个目录文件的 i 节点，如图 3-9 所示，其中 mnt 目录文件的 i 节点在 inode_table[32] 中占据的位置，正是图 3-8 中刚刚被释放掉的 hd1 设备文件曾占据的位置。从图 3-8 和图 3-9 的对比中，读者可以对空闲 i 节点的查找规则有一个直观的了解。

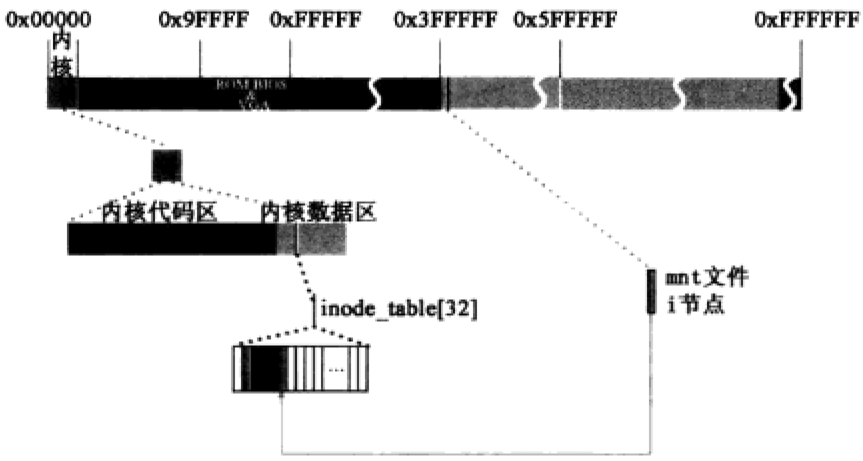


图 3-9　得到 mnt 目录文件的 i 节点

执行代码如下：

```
// 代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
    ...
    if (!(dir_i=namei(dir_name)))
    ...
}
```

namei 函数的执行步骤已经在 3.1 节中说明。

这样，本章的第二步工作——从虚拟盘上找到被安装点就完成了，接下来，继续完成本章的第一步工作，即通过获得的硬盘设备号把硬盘的超级块从设备上读出来。

3.3 得到 hd1 设备文件的超级块

本节的工作就是把硬盘上的超级块读取出来并加载到系统中的超级块管理表“super_ block[8]”中。加载设备文件的内容我们在 2.3 节中已经介绍，当时加载的是虚拟盘上的超级块，硬盘上的超级块功能与虚拟盘上的超级块功能是一样的，都是用来管理本设备上逻辑块位图、i 节点位图、数据块起始位置、魔数等信息的，而这些信息又直接管理着硬盘上的文件。所以，加载了超级块就可以使进程依托系统对硬盘上的文件进行操作。

3.3.1　准备读取 hd1 设备文件超级块

调用 read_super 函数，以 hd1 这个设备的设备号为参照，从硬盘上读取超级块。对应代码如下：

```
// 代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
    ...
    if (!(sb=read_super(dev)))
    ...
}
```

3.3.2　为 hd1 设备文件的超级块找到存储位置

进入 read_super 函数后，调用 get_super 函数，在系统的超级块管理结构 super_block[8] 中为硬盘上的超级块找出一个空闲项。图 3-10 给出了该空闲项的位置，由于现在根设备的超级块已经占据了第一项，所以此次找到的空闲项应该是第二项，它将用来存储 hd1 设备文件的超级块：

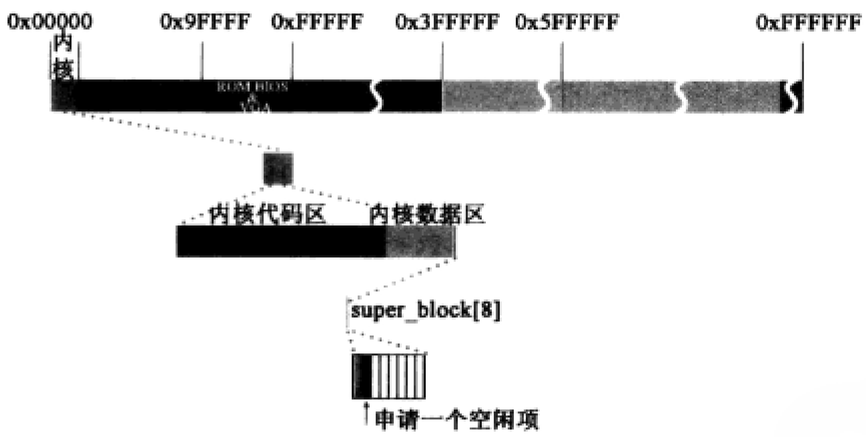


图 3-10　hd1 设备文件的超级块在内核超级块表的存储位置

执行代码如下：

```
// 代码路径：fs/super.c：
static struct super_block * read_super(int dev)
{
    ...
    if (s = get_super(dev))
    ...
}
```

3.3.3　初始化空闲超级块并加锁

申请到空闲的超级块项后，先对这个空闲项进行初始化设置，之后再给这个空闲项加锁。这

是因为系统中的超级块管理结构 super_block[8] 属于公共资源，即所有进程都有可能依托系统操作它。给它加锁，就是为了避免其他进程依托系统操作这个被选中的表项，从而导致数据混乱。如图 3-11 所示：

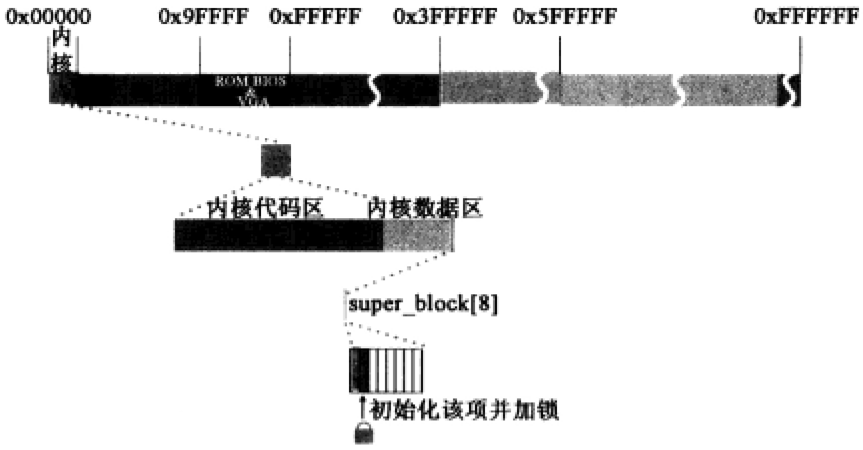


图 3-11　初始化空闲超级块并加锁

对应的代码如下：

```
// 代码路径：fs/super.c：
static struct super_block * read_super(int dev)
{
    ...
    s->s_dev = dev;
    s->s_isup = NULL;
    s->s_imount = NULL;
    s->s_time = 0;
    s->s_rd_only = 0;
    s->s_dirt = 0;
    lock_super(s);
    ...
}
```

3.3.4　从硬盘获得 hd1 设备文件的超级块

从硬盘上把这个超级块读入缓冲区并加载到空闲项上，之后对这个超级块中的魔数进行分析，以此判定这个超级块是不是符合系统的处理要求。bread 函数在 2.3.3 节至 2.3.8 节已有详细介绍。执行代码如下：

```
// 代码路径：fs/super.c：
static struct super_block * read_super(int dev)
{
    ...
    if (!(bh = bread(dev,1)))
    ...
    *((struct d_super_block *) s) =
        *((struct d_super_block *) bh->b_data);
    ...
    if (s->s_magic != SUPER_MAGIC)
    ...
}
```

图 3-12 表示了将超级块数据从硬盘加载到 3.3.2 节在内核超级块管理结构 super_block[8] 中

找到的空闲项中。

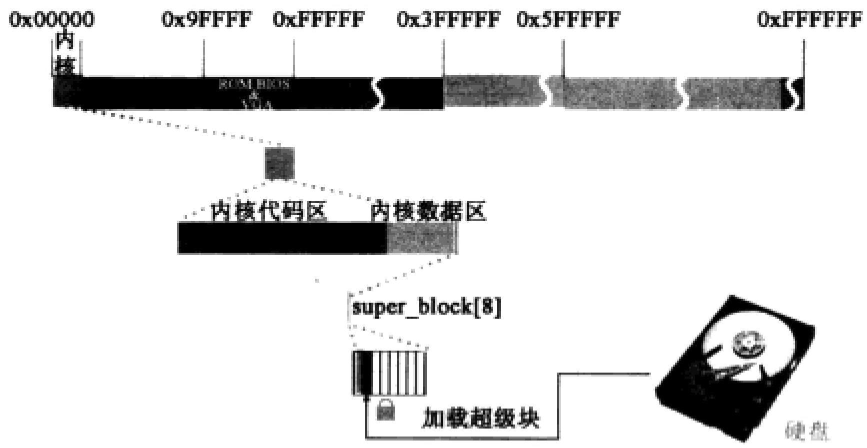


图 3-12　从硬盘读取 hd1 设备文件的超级块

3.3.5　加载逻辑块位图和 i 节点位图

超级块加载完毕后，根据超级块中提供的硬盘逻辑块位图信息和 i 节点位图信息将硬盘上的逻辑块位图与 i 节点位图加载至缓冲区，并使其常驻于高速缓冲区中，一切操作完毕后，将超级块解锁，执行代码如下：

```
// 代码路径：fs/super.c：
static struct super_block * read_super(int dev)
{
    ...
    for (i=0;i<I_MAP_SLOTS;i++)
        s->s_imap[i] = NULL;
    for (i=0;i<Z_MAP_SLOTS;i++)
        s->s_zmap[i] = NULL;
    block=2;
    for (i=0;i<s->s_imap_blocks ; i++)
        if (s->s_imap[i]=bread(dev,block))
            block++;
        else
            break;
    for (i=0;i<s->s_zmap_blocks ; i++)
        if (s->s_zmap[i]=bread(dev,block))
            block++;
        else
            break;
    ...
    free_super(s);
    ...
}
```

图 3-13 表示了超级块管理逻辑块位图和 i 节点位图的方式，以及 s_imap[8] 和 s_zmap[8] 与实际存储 i 节点位图和逻辑块位图的缓冲块的对应关系。

到这里为止，本章介绍的第二步内容就已经完成了，加载了超级块，就可以直接使用超级块上的 s_imount 这个成员，该成员记载了安装到目录文件的 i 节点号。接下来，将开始介绍本章第三步的内容，即将两者挂接。

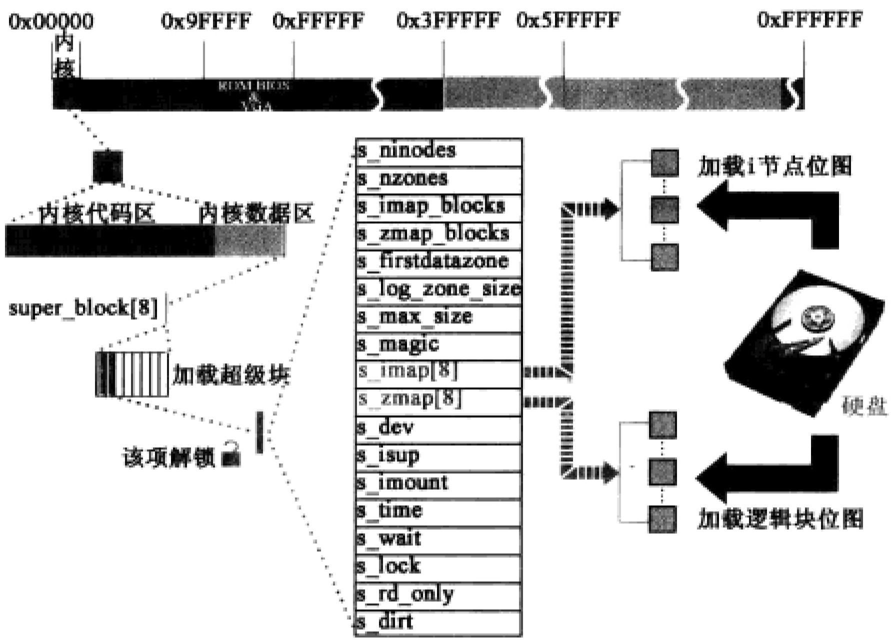


图 3-13　超级块管理逻辑块位图和 i 节点位图的方式

3.4　将 hd1 设备文件与 mnt 目录文件的 i 节点挂接

这里将“从硬盘上找到安装点”和“从虚拟盘上找到被安装点”进行挂接，这样，用户将来访问硬盘文件的道路就打通了，就能够以根 i 节点为起点，最终找到指定要读取文件的 i 节点，执行代码如下：

```
// 代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
    ...
    sb->s_imount=dir_i;
    ...
}
```

以上代码中的 dir_i 就是虚拟盘上被安装的 i 节点，图 3-14 形象地表示了 hd1 设备文件的超级块与 mnt 目录文件的 i 节点挂接的效果。

接下来还要对这个 i 节点的属性进行设置。值得注意的是：应对该 i 节点中的 i_mount 成员进行设置，这个成员表明了 mnt 这个 i 节点上挂接着其他设备的超级块，这个 i 节点是“跨设备查找文件”的拐点，执行代码如下：

```
// 代码路径：fs/super.c：
int sys_mount(char * dev_name, char * dir_name, int rw_flag)
{
    ...
        dir_i->i_mount=1;
       dir_i->i_dirt=1;
    ...
}
```

现在，硬盘的文件系统已经成功安装。

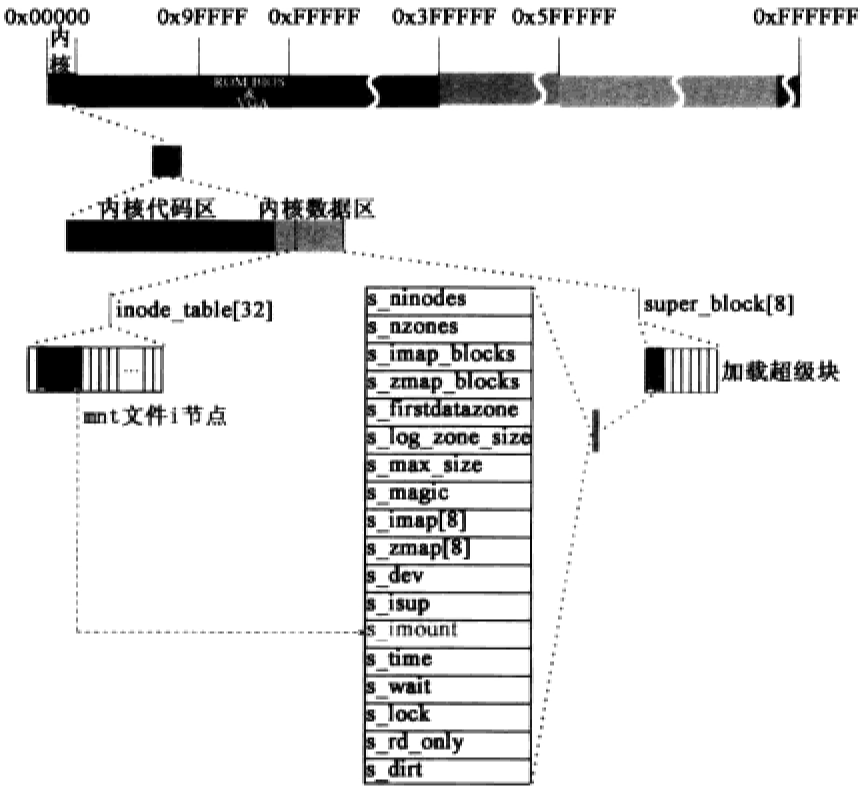


图 3-14 hd1 设备文件与 mnt 目录文件挂接

3.5 小结

系统在怠速后，通过用户指令，从虚拟盘上获取 mnt 目录文件的 i 节点、从硬盘上获取超级块，并将超级块中的 s_imount 挂接点和 mnt 目录文件的 i 节点进行挂接，以此来安装硬盘的文件系统。

成功安装硬盘的文件系统，标志着：系统从此以后，可以与硬盘这个外设，以“文件”的形式进行数据交互了。

第 4 章

文件操作

在第 3 章中，已经加载了硬盘文件系统，标志着用户能够以文件的形式对硬盘上的数据进行访问了。为何要以文件的方式访问硬盘数据，直接以扇区的方式对硬盘进行访问难道不行吗？不能说不行，但是存在许多的问题。比如，通常来说，用户所要使用的数据都要大于一个扇区，这就意味着这些数据需要存储在多个扇区上。一个简单的方法是将这些数据存储在一段物理上连续的扇区中。但是，随着数据不断地被修改、删除和添加，硬盘中很快就会产生大量没有存储数据的碎片，而且理论和实际都表明碎片会越来越多，最终整个硬盘几乎被碎片填满，比碎片尺寸大的数据将无法利用这些碎片。如果不进行整理，硬盘虽然有大量的未用空间，却无法写入数据，浪费了宝贵的硬盘存储空间。整理碎片又非常耗时。

操作系统的设计者想出了“以碎制碎”的策略。既然对数据的频繁操作会使得硬盘存储空间变碎，索性就事先把硬盘的存储空间人为分成许多规则的、整齐划一的小块，把数据存放在这样的块中，这就是块设备的由来。通常，块的大小是扇区大小的整数倍，不同的操作系统块的大小略有不同，在 Linux 0.11 中，一个块的大小是 1KB，就是两个扇区。

为了实现这个创意，操作系统的设计者提出了文件系统的设计方案。文件系统不仅实现了这个创意，还进一步解决了在存储数据时不会覆盖已经存在的数据的问题，这一点意义重大。

文件系统首先设计了一个数据结构，管理属于同一个文件的所有的块。对于 Linux 0.11 来说，这个数据结构就是 i 节点。由于 i 节点的数量极其庞大，需要一个数据结构来管理 i 节点的使用情况，这个数据结构就是 i 节点位图。

另外，硬盘上的块数量也是极其庞大的，同样需要一个数据结构来标识块的使用情况，这个数据结构就是逻辑块位图。

以上这些数据结构的功能、大小、所处位置各不相同，需要一个总的数据结构来管理，这个数据结构就是超级块。

通过以上介绍的这些数据结构来管理文件系统主要操作的具体表现为：

在写操作的过程中，通过“超级块→逻辑块位图→数据块”这条路线可以查找到硬盘上的空闲数据块；通过“超级块→ i 节点位图→ i 节点表中的 i 节点→数据块”这条路线可以访问到文件中指定的数据块，如图 4-1 所示：

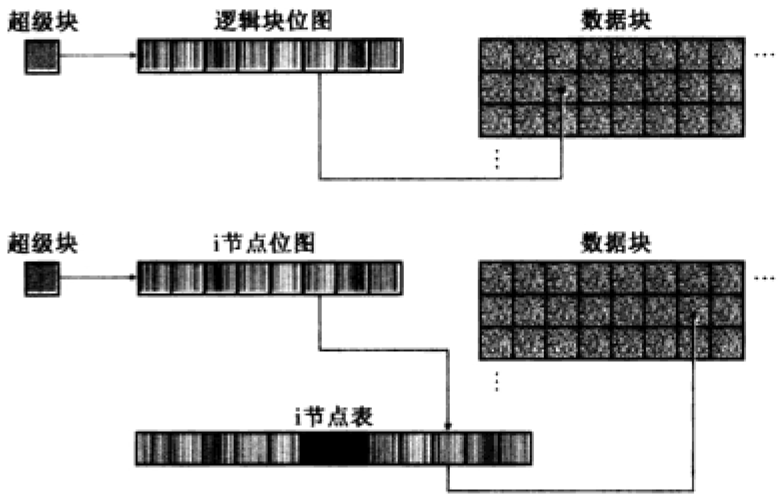


图 4-1　查找硬盘空闲数据块和访问文件指定数据块的示意图

当然，文件系统最终是为人的使用而设计的，人习惯于使用路径名、文件名，这就需要在文件系统中建立一个特殊的数据结构，将路径名、文件名与上述管理数据结构相关联，这个数据结构就是目录项。

值得一提的是文件系统中涉及 i 节点的代码非常多，其中的规律却不复杂，就是一个 i 节点对应一个文件，而且只对应一个文件，反之亦然。也就是说 i 节点和文件是严格的一一对应的，不存在任何形式的例外。另外还需要注意的是，目录文件也是文件，也是有 i 节点的。文件系统通常的组织规则是：

根目录文件 i 节点→根目录文件→根目录文件中 A 的目录项→ A 目录文件的 i 节点→ A 目录文件→ A 目录文件中 B 的目录项→ B 目录文件的 i 节点→ B 目录文件→ B 目录文件中 C 的目录项→……M 目录文件→ M 目录文件中 N 的目录项→ N 文件的 i 节点→ N 文件内容

这个规律如图 4-2 所示：

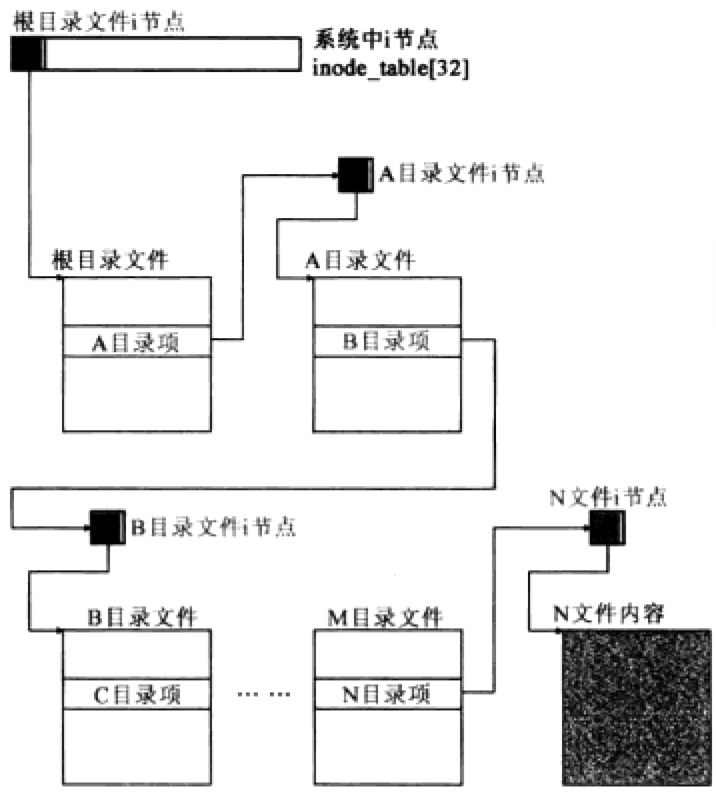


图 4-2　文件操作共性示意图

这个规律在绝大多数情况下都是适用的，我们只要牢记这两点，就会比较容易掌握文件系统。

本章将通过 3 个实例的实际运行来详细讲解文件系统的工作原理和实际运行状况。我们假设本章中对文件的操作都是由一个用户进程完成的。

实例 1：该用户进程对硬盘上一个已有的文件执行“打开”和“读取”这两个操作。

实例 2：该用户进程在硬盘上新建一个文件，并将内容写入这个文件。

实例 3：关闭此文件，之后将其从文件系统中删除。

这 3 个实例的内容涉及了 Linux 0.11 中对文件的所有基本操作，非常典型，掌握了这 3 个实例，就等于掌握了 Linux 0.11 中文件系统的绝大部分内容。

下面先来详细讲解实例 1，我们设想的应用场景是，硬盘上已经存在一个事先写好的名为 helloc.txt 的文件。

实例 1：用户进程打开一个在硬盘上已存在的文件，并读取文件的内容

本实例分为两部分：打开文件和读取文件，用户进程中所使用的代码如下：

```
void main()
{
//打开文件
char buffer[12000];
int fd = open("/mnt/user/helloc.txt", O_RDWR ,0644));
//读取文件
int size = read(fd,buffer,sizeof(buffer));
return;
}
```

这个用户进程的程序所要达到的最终目的就是将 helloc.txt 这个文件中的内容读出来。要想实现此目的，就要满足以下要求：

- 要确定读取哪个文件；
- 要确定具体读取文件的哪部分内容，是全部内容还是部分内容；
- 把文件内容读出后，要能够存储在用户事先指定的空间内。

以上 3 个要求就是 Linux 0.11 操作系统中所有文件读取操作的核心要求，这个进程程序中对 open 和 read 这两个库函数的调用就是为了满足以上要求，从而实现最终读取文件内容的目的。

确定要读取哪个文件，用操作系统的术语表述就是“打开文件”；确定要写哪个文件，用操作系统的术语表述也是“打开文件”，这一点将在 4.4 节中讲解。准确地说，确定要操作哪个文件，用操作系统的术语表述都是“打开文件”。

- open 函数的核心目的就是实现第一个要求，我们将在 4.1 节中详细介绍。
- read 函数的核心目的就是实现第二个要求和第三个要求，我们将在 4.2 节中详细介绍。

下面，先来看实例 1 中打开文件的具体实现方法。

4.1 打开文件

本节要实现实例 1 中的第一个要求，要确定读取哪个文件，即“打开文件”。

因为在操作系统看来，文件是根据进程的需求来操作的，又因为文件的 i 节点是记载文件属性的最关键的数据结构，所以，只要使进程与指定的 i 节点建立关系，就可以实现实例 1 中的第一个要求：确定读取哪个文件，即“打开文件”。“打开文件”的本质就是为了建立这套关系。

具体表现为，以系统中的文件管理表 file_table[64] 为中转站，让进程管理结构 task_struct 中的文件管理指针表 *filp[20] 与将要打开的 helloc.txt 文件的 i 节点建立关系，从而使进程具备操作这个文件的能力。本节要讲述的就是这个过程的具体实现方法。

实现的具体过程分为三个步骤进行：

第一步：将用户进程管理结构 task_struct 中的文件管理指针表 *filp[20] 与内核中的文件管理表 file_table[64] 进行挂接；

第二步：以用户给定的路径名“/mnt/user/helloc.txt”为线索，找到 helloc.txt 文件的 i 节点；

第三步：将 helloc.txt 对应的 i 节点在文件管理表 file_table[64] 中进行登记，从而使进程具备操作 helloc.txt 文件的能力。

图 4-3 是本节将要讲述的主要内容的架构图：

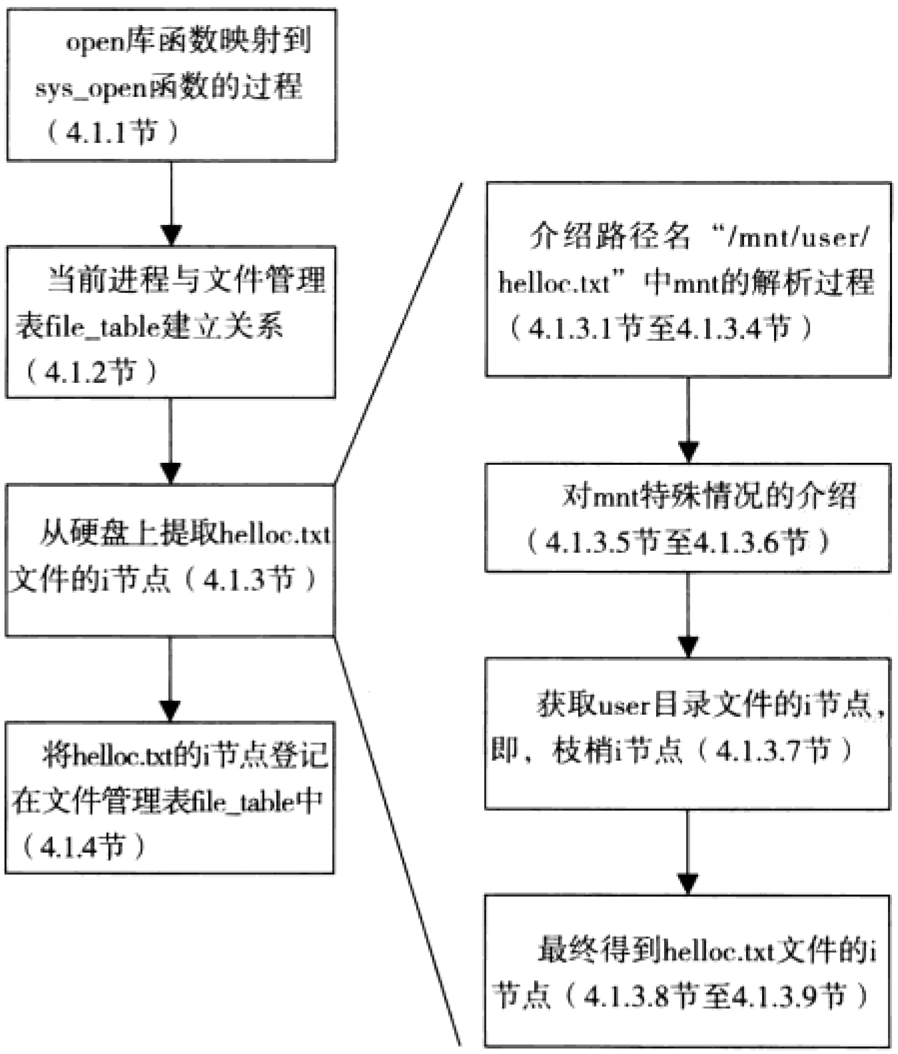


图 4-3 打开操作实例执行过程的架构图

4.1.1 用户程序调用 open 库函数产生软中断

通过实例 1 的源代码我们可以得知，用户程序是通过调用 open 函数打开文件的。open 函数是一个库函数，该函数被调用后，会产生 int 0x80 软中断，最终映射到 sys_open 中执行。该过程如图 4-4 所示：

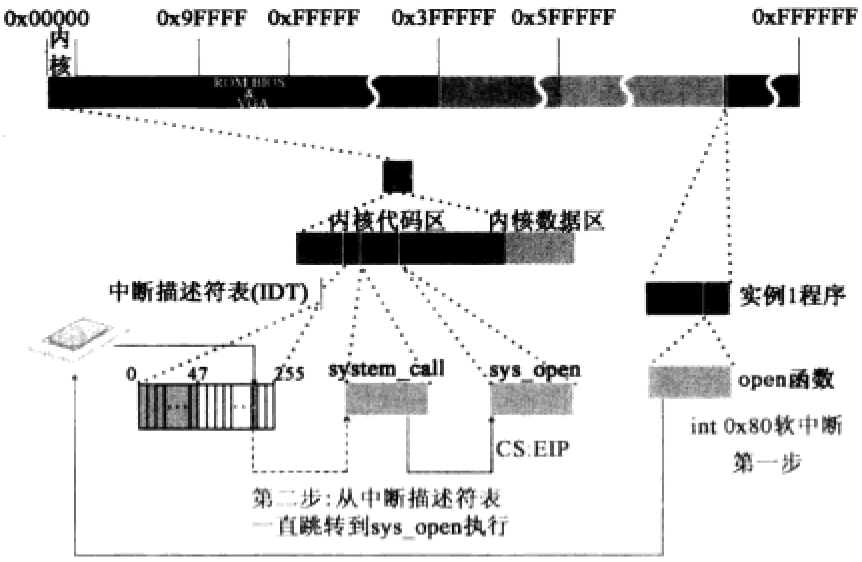


图 4-4 执行 sys_open

具体的映射步骤与 2.1.1 节中的 fork 函数映射到 sys_fork 的步骤类似。区别在于，open 函数需要传递 3 个参数，所以会执行 _syscall3 这个宏。对应代码如下：

```
//代码路径：include/unistd.h：
#define _syscall3(type,name,atype,a,btype,b,ctype,c)
```

在该宏函数中利用 ebx、ecx 和 edx 这 3 个寄存器，分别将 open 需要传递的 3 个参数传递给内核。对应代码如下：

```
//代码路径：include/unistd.h：
#define _syscall3(type,name,atype,a,btype,b,ctype,c) \
type name(atype a,btype b,ctype c) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(a)),"c" ((long)(b)),"d" ((long)(c))); \
if (__res>=0) \
      return (type) __res; \
errno=-__res; \
return -1; \
}
```

这个宏展开之后，看上去就像如下的形式：

```
int open(const char* filename, int flag, int mode)
{
long __res;
__asm__ volatile ("int $0x80"
    : "=a" (__res)
```

```
    : "0" (__NR_open),"b" ((long)( filename)),"c" ((long)( flag)),"d" ((long)(mode)));
if (__res>=0)
      return (int) __res;
errno=-__res;
return -1;
}
```

最终，映射到 sys_open 函数中，打开文件的具体过程就是在 sys_open 函数中实现的。接下来，我们开始介绍文件打开的具体过程。

4.1.2 建立用户进程与文件管理表的关系

这里我们来介绍打开文件的第一步，即将用户进程管理结构 task_struct 中的文件管理指针表 *filp[20] 与内核中的文件管理表 file_table[64] 进行挂接，从而为用户进程操作文件打下基础。

图 4-5 中形象地表示了这个挂接动作：

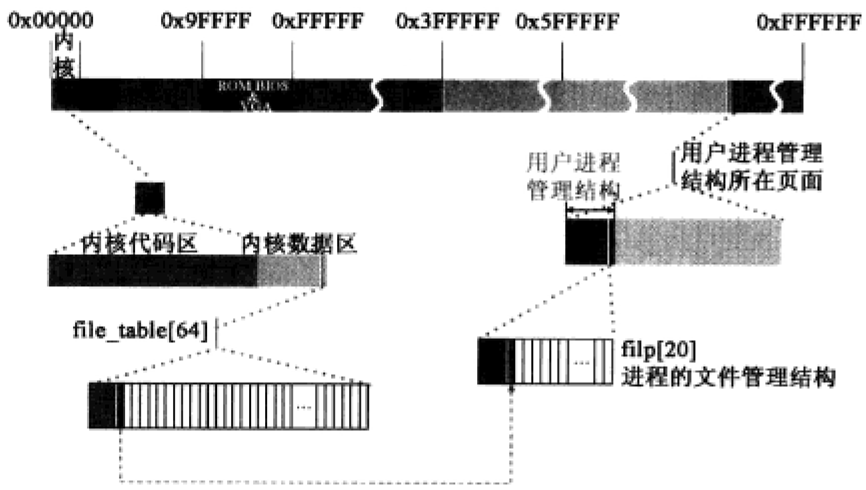


图 4-5 获得空闲文件槽和文件句柄

执行步骤如下：

（1）先通过遍历当前进程管理结构 task_struct 中的文件管理指针表 *filp[20] 中的每一项，找到一个指针记录为空的项，哪一项指针记录为空，就表明该项为空闲项，执行代码为：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
    ...
    for(fd=0 ; fd<NR_OPEN ; fd++)
        if (!current->filp[fd])
            break;
    ...
}
```

（2）再通过遍历内核文件管理表 file_table[64] 中的每一项，找到一个引用计数 f_count 为 0 的项，哪一项 f_count 为 0，就表明该项为空闲项，对应代码为：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
```

```
    ...
    f=0+file_table;
    for (i=0 ; i<NR_FILE ; i++,f++)
        if (!f->f_count) break;
    ...
}
```

（3）最后，将两个空闲项挂接，并把在内核文件表 file_table[64] 中找到的空闲项的引用计数加 1，以表明该文件槽已经被使用了，对应代码为：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
    ...
    (current->filp[fd]=f)->f_count++;
    ...
}
```

这样挂接的理由是：所有的进程，只要想操作文件，就要通过操作系统提供的文件管理表 file_table[64] 来实现；所有的文件，只要想被进程操作，也要登记在文件管理表 file_table[64] 中。可见，这个文件管理表 file_table[64] 就是进程操作文件的“中转站”，现在将两个空闲项挂接，就为进程最终具备操作 helloc.txt 的能力创造了条件。

4.1.3 从硬盘上获取 helloc.txt 文件的 i 节点

这里我们来介绍打开文件的第二步，即以用户给定的路径名“/mnt/user/helloc.txt”为线索，找到 helloc.txt 文件的的 i 节点。这一步的最终目的就是要把 helloc.txt 文件的 i 节点从硬盘上读出来，从而为把 helloc.txt 文件的 i 节点在文件管理表 file_table[64] 中找到的空闲项上登记创造条件。

Linux 0.11 是通过对路径名的不断解析来找到指定文件的 i 节点的，这里我们先对将要参与解析的要素进行介绍。

我们在本章开头已经对文件系统的构成要件进行了介绍，每个文件都对应一个唯一的 i 节点，路径名中的“helloc.txt”是普通文件的文件名，helloc.txt 文件毫无疑问会有一个唯一的 i 节点；路径名中的“mnt”和“user”是目录文件的文件名，mnt 文件和 user 文件也都会有一个唯一的 i 节点。这就说明文件和 i 节点的一一对应性适用于整个文件系统中的全部文件。

目录文件与普通文件也是有区别的，它们存储的内容不一样。目录文件中存储着若干个目录项，每个目录项由两部分构成：一个是文件名，表明了该目录项所对应的文件名，这个文件名可以是一个普通文件的文件名，也可以仍然是一个目录文件的文件名；另一个是 i 节点号，表明了该目录项所对应的文件在 i 节点表中的项号，通过这个 i 节点号就可以在 i 节点表中找到所对应的文件的 i 节点。

参与解析的要素大体包括上述这些，即 i 节点、文件名、目录项、目录文件等。系统就是从路径名上不断地采集信息，然后根据这些信息对以上要素进行分析，最终找到 helloc.txt 这个普

通文件的i节点的。我们这里首先大致给出一张“解析示意图”，以此标明解析的技术路线，如图4-6所示。

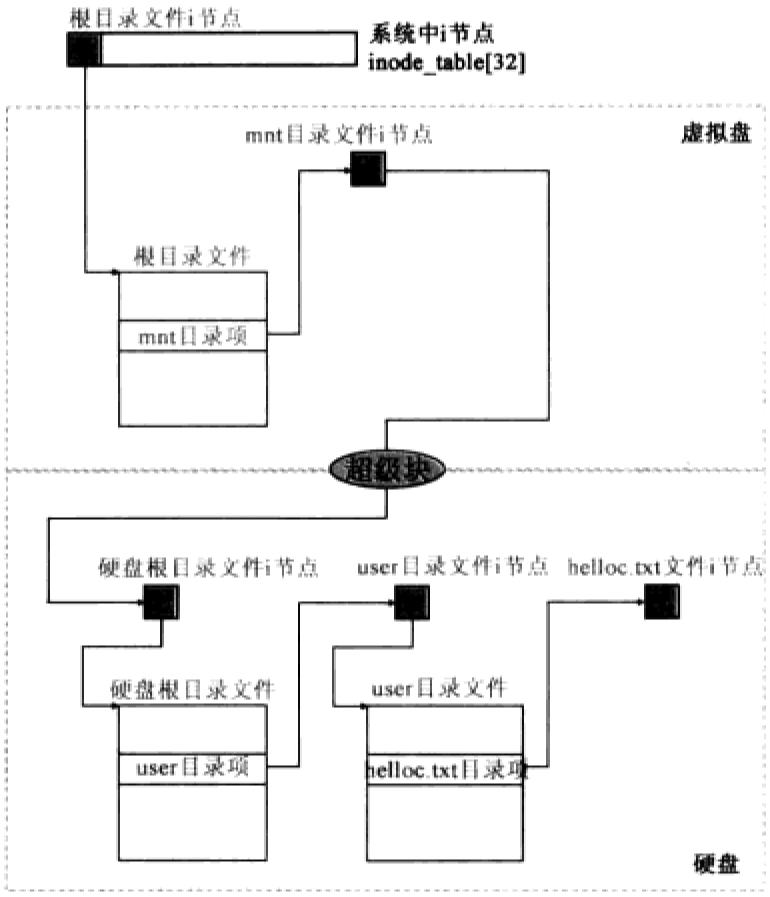


图4-6 文件路径解析示意图

通过此示意图我们可以看出，路径名的解析具有相当的同构性，解析的技术路线是：寻找i节点→通过i节点找到目录文件→通过目录文件找到目录项→通过目录项找到后续路径名的i节点号。如此周而复始地解析，直到最终找到helloc.txt这个普通文件的i节点。这就意味着，只要我们把某一个“文件名→i节点”的前后解析步骤分析清楚，其余的步骤大体就可以推导出来。

接下来，我们就着重分析一下路径名中的一个文件名的解析步骤，以此贯穿到整个路径名的解析中。

4.1.3.1 准备查找枝梢i节点——user目录文件的i节点

通过路径名解析出i节点，这是由open_namei函数完成的，所以sys_open函数接下来要调用open_namei函数来完成解析步骤，执行代码如下：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
    ...
    if ((i=open_namei(filename,flag,mode,&inode))<0)
    ...
}
```

进入open_namei函数后，调用dir_namei函数，通过解析路径名得到枝梢i节点，对于“/mnt/user/helloc.txt”这个路径名而言，枝梢i节点就是user这个目录文件的i节点，执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    if (!(dir = dir_namei(pathname,&namelen,&basename)))
    ...
}
```

进入 dir_namei 函数后，调用 get_dir 函数，同时将路径名也传递下去，开始实质性的 i 节点解析工作，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * dir_namei(const char * pathname,
    int * namelen, const char ** name)
{
    ...
    if (!(dir = get_dir(pathname)))
    ...
}
```

4.1.3.2 确定查找操作的起点

进入 get_dir 函数后，开始对路径名进行分析，通过 4.1.3 节的图 4-4“解析示意图”我们可以看出，首先要确定“从哪里”开始解析，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * get_dir(const char * pathname)
{
    ...
    if ((c=get_fs_byte(pathname))=='/') {
        inode = current->root;
        pathname++;
    }
    ...
}
```

这几行代码的意思是，如果路径名的第一个字符是‘/’，就从根目录文件 i 节点（以下简称根 i 节点）进行解析，现在，根 i 节点就是解析路径名的起点。我们继续介绍解析的步骤。

4.1.3.3 获得名为 mnt 的目录项

我们继续看图 4-6，接下来就要通过根 i 节点在根目录文件中找到名字为 mnt 的目录项，以便完成从虚拟盘到硬盘的过渡。图 4-7 表示了获取 mnt 目录项的主要过程：

具体执行步骤是这样的：

（1）先解析出 mnt 这个字符串的长度，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * get_dir(const char * pathname)
{
    ...
```

```
    for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
    ...
}
```

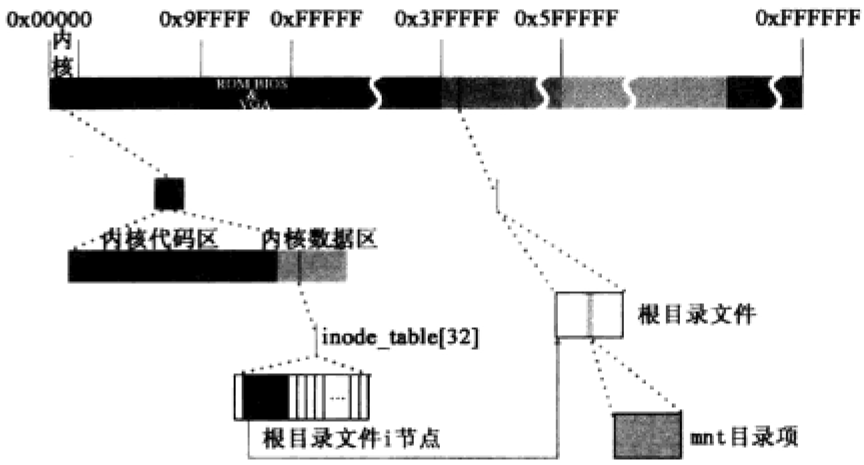


图 4-7 准备从虚拟盘切换到硬盘

（2）再以 mnt 这个名字为参照，在根目录文件中找到名字为 mnt 的目录项，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * get_dir(const char * pathname)
{
    ...
    if (!(bh = find_entry(&inode,thisname,namelen,&de)))
    ...
}
```

find_entry 函数的两个实参，thisname 标识了 mnt 字符串的起始位置，namelen 标识了该字符串的长度，这样就能锁定 mnt 这个字符串。

进入 find_entry 函数后，开始将根目录文件读取出来，并从中找出名字为 mnt 的目录项，读取并分析的过程是这样的：先确定这个目录文件的大小，因为确定了大小就可以确定目录文件包含多少条目录项，方法是通过根 i 节点中表示根目录文件大小的 i_size 字段除以每个目录项占用的字节数，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct buffer_head * find_entry(struct m_inode ** dir,
    const char * name, int namelen, struct dir_entry ** res_dir)
{
    ...
    entries = (*dir)->i_size / (sizeof (struct dir_entry));
    ...
}
```

（3）之后，就从目录文件的第一个数据块开始读取数据，将数据块读入到缓冲块，并调用 match(namelen,name,de) 函数分析这些数据中有没有名字为 mnt 的目录项，如果有，目的就达到了；如果没有，就接着从第二个数据块继续读取并分析，直到找到名字为 mnt 的目录项为止，执行代码如下：

```
//代码路径：fs/namei.c：
static struct buffer_head * find_entry(struct m_inode ** dir,
    const char * name, int namelen, struct dir_entry ** res_dir)
{
    ...
    if (!(bh = bread((*dir)->i_dev,block)))
    ...
    de = (struct dir_entry *) bh->b_data;
    ...
    while (i < entries) {
        if ((char *)de >= BLOCK_SIZE+bh->b_data) {
            brelse(bh);
            bh = NULL;
            if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
                !(bh = bread((*dir)->i_dev,block))) {
                i += DIR_ENTRIES_PER_BLOCK;
                continue;
            }
            de = (struct dir_entry *) bh->b_data;
        }
        if (match(namelen,name,de)) {
            *res_dir = de;
            return bh;
        }
        de++;
        i++;
    }
}
```

（4）最终找到了 mnt 这个目录项，值得注意的是，根目录文件在虚拟盘上，所以此时调用 bread 函数从设备上读取数据并不会产生中断。之后，find_entry 函数就返回了，继续回到 get_dir 函数中执行。

4.1.3.4 获得虚拟盘到硬盘转换点——mnt 目录文件的 i 节点

我们继续看 4.1.3 节起始部分的图 4-6，得到了 mnt 目录项，就要从该目录项中解析出 i 节点号，如图 4-8 的右下角所示：

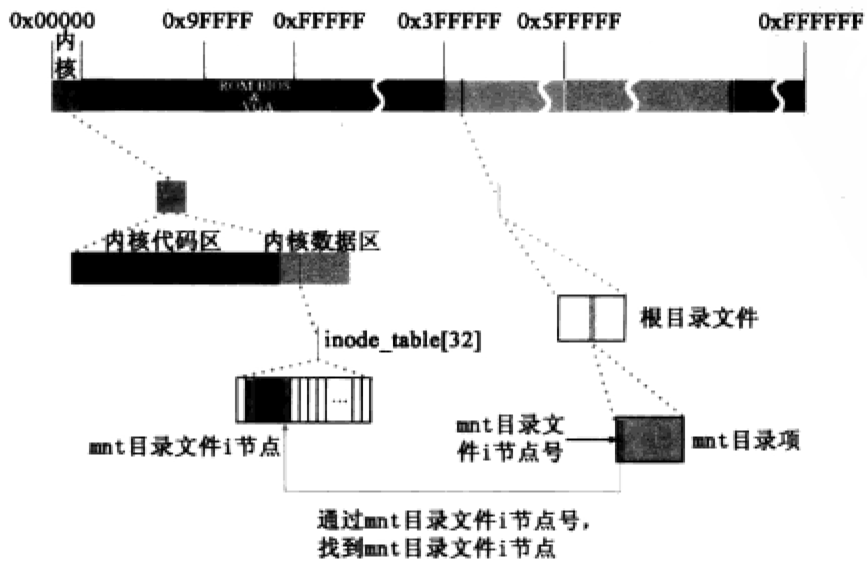


图 4-8 获得 mnt 目录的 i 节点号和 i 节点

执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * get_dir(const char * pathname)
{
    ...
    inr = de->inode;
    ...
}
```

通过4.1.3.2节中的介绍我们得知，现在inode表示根i节点，所以通过“idev = inode->i_dev”这行语句可获得根设备号，即虚拟盘的设备号。所以，接下来就要通过调用iget函数从虚拟盘中把mnt目录文件的i节点读取出来，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * get_dir(const char * pathname)
{
    ...
    idev = inode->i_dev;
    ...
    if (!(inode = iget(idev,inr)))
    ...
}
```

进入iget函数后，先要在系统的i节点管理表inode_table[32]中找到一个空闲项，以便存储将要读取的i节点。具体地，通过get_empty_inode函数在i节点管理表中找到一个i_count字段为0的空闲项，并在找到该空闲项后，把它对应的存储区域清0，这个动作的意义在于它实际上完成了内存中i节点管理结构的初始化。随后将该空闲项的引用计数置1，表示它已被使用。执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ...
    empty = get_empty_inode();
    ...
}
// 代码路径：fs/inode.c：
struct m_inode * get_empty_inode(void)
{
    ...
    do {
        inode = NULL;
        for (i = NR_INODE; i ; i--) {
            if (++last_inode >= inode_table + NR_INODE)
                last_inode = inode_table;
            if (!last_inode->i_count) {
                inode = last_inode;
                if (!inode->i_dirt && !inode->i_lock)
                    break;
            }
        }
```

```
        ...
    } while (inode->i_count);
    memset(inode,0,sizeof(*inode));
    inode->i_count = 1;
    return inode;
}
```

然后，以“mnt”这个目录项中提供的 i 节点号为依托，遍历系统中的 i 节点管理表 inode_table[32]，看看 mnt 这个目录文件的 i 节点之前是不是已经存在于该管理表中了，如果存在，就不用再读取了。显然，我们在第 3 章中安装文件系统时，mnt 这个目录文件的 i 节点被用于和硬盘进行挂接，这就意味着这个 i 节点早已被加载至 i 节点管理表 inode_table[32] 中了。如图 4-8 中的 i 节点表所示，图中用一个绿色竖线形象地表示了 mnt 目录文件 i 节点占用的位置。所以，用不着从设备上再读取一遍了，可以直接使用，执行代码如下：

```
//代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ...
    if (inode->i_dev != dev || inode->i_num != nr)
    ...
}
```

到这里为止，我们以“mnt”这个目录文件名为例，把 4.1.3 节开始部分提到的解析技术路线“寻找 i 节点→通过 i 节点找到目录文件→通过目录文件找到目录项→通过目录项找到后续路径名的 i 节点号”的整个过程都介绍完了。

一般情况下，路径名中的每一个文件名都是以这个技术路线进行解析的，直到把最终指定的文件的 i 节点找到。查找方法都一样，通过这种查找方法，系统就可以根据用户提供的路径名查找到任意文件的 i 节点，并将结果返回。只要能得到 i 节点，就能得到对这个 i 节点所对应的文件的操作权。

4.1.3.5 完成从虚拟盘到硬盘的转换

但是，现在出现了特殊情况，我们先来回忆一下第 3 章安装硬盘文件系统的内容，最终硬盘超级块上的安装 i 节点与虚拟盘 mnt 目录文件的 i 节点相挂接，从而进程能够依托系统对硬盘上的文件进行访问。

如果 mnt 目录文件的 i 节点不是挂接点，那么 4.1.3.4 节最后在系统 i 节点管理表中得到这个 i 节点后，就会继续返回到 get_dir 函数中。通过这个 i 节点，继续查找 mnt 这个目录文件，然后再从这个目录文件中找到路径名中的下一个目录项，再找 i 节点号，再得到 i 节点……以此类推，不断重复地执行解析的技术路线，直到最终达到目的。

但现在的情况是，mnt 目录文件的 i 节点是硬盘文件系统的“挂接点”，这就意味着，要到硬盘中去查找路径名中的下一个要找的文件名“user”，而非仍旧从虚拟盘中去查找。mnt 目录文件的 i 节点是一个转折点，任何路径名，只要其中出现了“mnt”，就要转而去硬盘上解析后续的文件名。

当然，mnt 目录文件本身还是存储在虚拟盘上的，但就因为它是挂接点，所以，原来在 mnt

目录下存储的所有目录项的内容，现在“全部作废”，即在安装于 mnt 目录下的硬盘文件系统卸载前，这些目录项的内容无法被用户获得。在安装了硬盘文件系统的前提下，mnt 目录文件的主要功能就是作为访问硬盘数据的挂接点，至于 mnt 目录文件里面原来究竟有多少目录项，它们都是什么，与目前的路径名解析（即访问硬盘文件系统）一点关系都没有。直到将来把硬盘的文件系统卸载了，这些内容才能重新发挥作用，参与路径名的解析。

下面我们来看一下操作系统中处理这种特殊情况的代码：

```
// 代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ...
    if (inode->i_mount)
{
    ...
    for (i = 0 ; i<NR_SUPER ; i++)
                    if (super_block[i].s_imount==inode)
                        break;
    ...
    dev = super_block[i].s_dev;
              nr = ROOT_INO;
              inode = inode_table;
              continue;
}
    ...
}
```

inode->i_mount 这个标志为真，意味着 mnt 这个目录文件的 i 节点就是挂接点，既然是挂接点，就能在系统的超级块管理表 super_block[8] 中找到具体是哪个设备的超级块中的 s_imount 与这个挂接点相挂接，进而找到相应的超级块，现在这个超级块就是硬盘的超级块。这部分内容如图 4-9 所示：

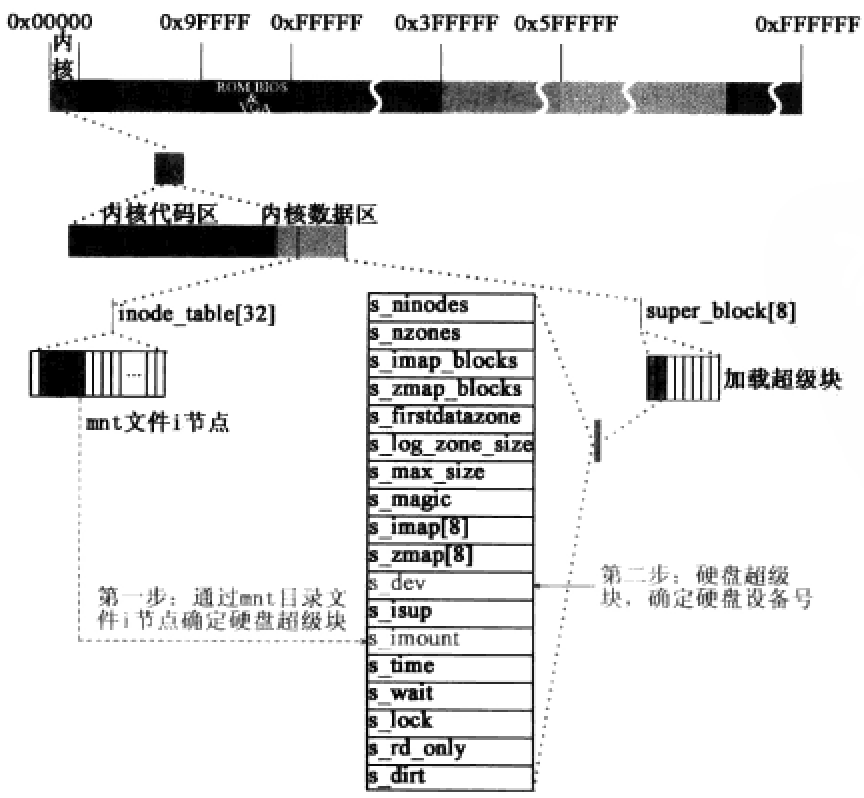


图 4-9 获得硬盘设备的设备号

在本章的开始部分我们介绍过，超级块是文件系统中处于顶端的管理结构，找到了超级块，就具备了与硬盘交互的能力，从 iget 函数中的以下两行代码（见加粗内容）：

```
// 代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ...
                dev = super_block[i].s_dev;
                nr = ROOT_INO;
}
    ...
}
```

我们可以看出，现在已经根据硬盘超级块中的信息得到了硬盘的设备号，同时利用根目录文件 i 节点定位识别的特性得到了硬盘根目录文件的 i 节点号，即 ROOT_INO。系统接下来准备将硬盘的根目录文件 i 节点提取出来，然后再从这个 i 节点开始进一步解析路径名“/mnt/user/helloc.txt”中 user 以后的部分。

现在我们来看一下提取的过程，先要执行 continue; 这一行，表示接下来要按照“硬盘”的设备号和“硬盘根目录文件 i 节点”的 i 节点号继续在位于内核数据区的 i 节点表中查找硬盘根目录文件的 i 节点，执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ...
    while (inode < NR_INODE+inode_table){
    if (inode->i_dev != dev || inode->i_num != nr) {
            inode++;
            continue;
        }
    ...
    }
    ...
}
```

由于从来没有载入过硬盘根目录文件的 i 节点，所以肯定找不到，最终就会跳出循环，并调用 read_inode 函数，准备从硬盘读出该 i 节点并载入到 4.1.3.4 节所介绍的“在 i 节点管理表 inode_table[32] 中获取的空闲项上”，执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ...
    inode=empty;
    inode->i_dev = dev;
    inode->i_num = nr;
    read_inode(inode);
    return inode;
}
```

4.1.3.6 获得硬盘根目录文件 i 节点

进入 read_inode 函数后，首先将申请到的这个空闲 i 节点表项加锁，以防止此时其他进程使用这一项；然后从硬盘上 i 节点的存储区域中把硬盘根目录文件的 i 节点读出来，并载入到位于内核数据区中的空闲 i 节点表项中；最后，将该 i 节点表项解锁。图 4-10 展示了将访问设备转换到硬盘并提取硬盘根目录文件的 i 节点的主要过程。

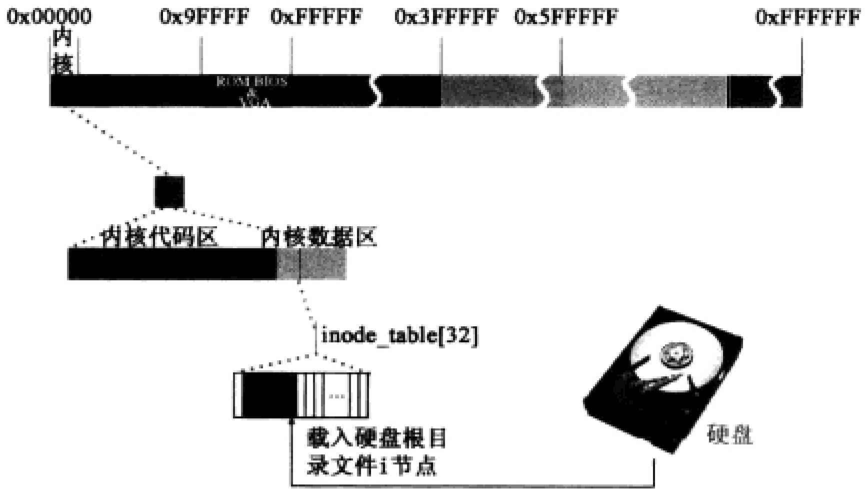


图 4-10 访问设备从虚拟盘转换到硬盘

执行代码如下：

```
// 代码路径：fs/inode.c：
static void read_inode(struct m_inode * inode)
{
     ...
     lock_inode(inode);
     ...
     block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
          (inode->i_num-1)/INODES_PER_BLOCK;
if  (!(bh=bread(inode->i_dev,block)))
    ...
    *(struct d_inode *)inode =
          ((struct d_inode *)bh->b_data)
               [(inode->i_num-1)%INODES_PER_BLOCK];
     ...
     unlock_inode(inode);
}
```

由于 i 节点是定位识别的，因此在得到硬盘设备号和硬盘根目录文件 i 节点号的基础上，就能够将硬盘根目录文件 i 节点所在的数据块从硬盘上读到高速缓冲区了。read_inode 中的

```
block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks
+(inode->i_num-1)/INODES_PER_BLOCK;
```

这行语句就是根据 i 节点号找到 i 节点在硬盘上所在块号的具体方法。

当包含硬盘根目录 i 节点的数据块从硬盘读入到高速缓冲区后，就以 i 节点号为线索，将硬盘根目录 i 节点的信息从缓冲块中提取出来并拷贝到刚找到的空闲 i 节点表项中，即 *(struct d_inode *)inode =((struct d_inode *)bh->b_data)[(inode->i_num-1)%INODES_PER_BLOCK]。

当然，此处拷贝到空闲 i 节点表项中的信息只是硬盘 i 节点的信息。而硬盘 i 节点中包含的字段比内存 i 节点要少，内存 i 节点中多出来的那部分字段的初始化工作正是在 4.1.3.4 节中找空闲 i 节点表项的过程中进行的。

之后 read_inode 函数就执行完了，iget 函数也会将硬盘根目录文件的 i 节点返回，最终回到 get_dir 函数中继续执行，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * get_dir(const char * pathname)
{
    ...
    while (1)
    {
        ...
    }
}
```

4.1.3.7 获得枝梢 i 节点——user 目录文件的 i 节点

在这个 while 循环中，接下来将以硬盘根目录文件 i 节点为基准，继续解析“/mnt/user/helloc.txt”这个路径名，解析的技术路线仍然与解析 mnt 这个文件名的路线相同。我们在 4.1.3.3 节至 4.1.3.5 节中已经详细说明，这里我们只介绍一下解析的结果：最终将把 user 这个目录文件的 i 节点返回，user 目录文件的 i 节点也就是枝梢 i 节点。图 4-11 形象地表示了从硬盘上把硬盘根目录文件的数据读入到缓冲块，在硬盘根目录文件中查找 user 目录项的过程。

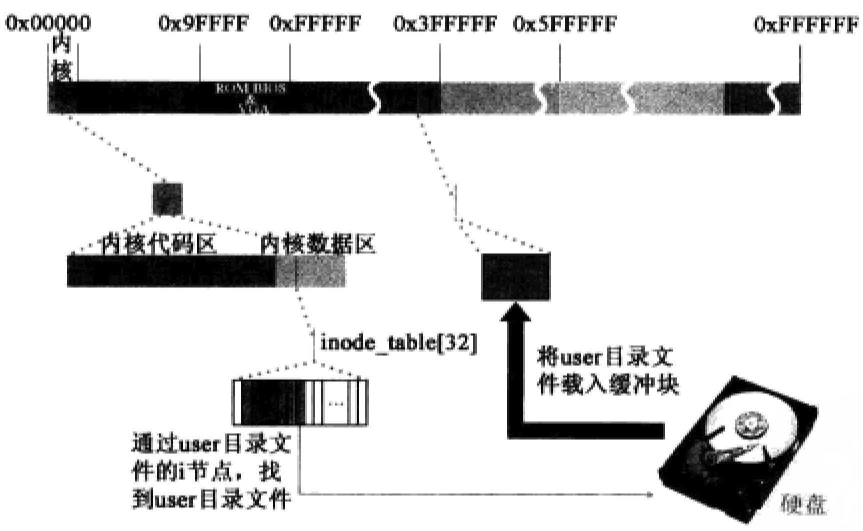


图 4-11 找到 user 目录项

这样，get_dir 函数就执行完了，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * get_dir(const char * pathname)
{
    ...
    while (1)
    {
        if (!c)
            return inode;
    }
}
```

接下来将返回到 dir_namei 函数继续执行，对应代码如下：

```
// 代码路径：fs/namei.c：
static struct m_inode * dir_namei(const char * pathname,
    int * namelen, const char ** name)
{
    ...
    basename = pathname;
    while (c=get_fs_byte(pathname++))
        if (c=='/')
            basename=pathname;
    *namelen = pathname-basename-1;
    *name = basename;
    return dir;
}
```

这段代码的意思是：计算出路径名“/mnt/user/helloc.txt”中“helloc.txt”字符串的长度，以此为最终获取该文件的 i 节点做准备。到此为止，dir_namei 函数就执行完了，将返回到 open_namei 函数中去执行，现在已经获得了枝梢 i 节点，即 user 目录文件的 i 节点，这就为最终获得 helloc.txt 这个文件的 i 节点创造了条件。

4.1.3.8 找到 helloc.txt 文件的目录项

我们现在再来看 4.1.3 节中开始部分的图 4-6，从图中可知接下来就要通过 user 目录文件的 i 节点找到 user 目录文件，并从中提取出名为“helloc.txt”的目录项了，执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,struct m_inode ** res_inode)
{
    ...
    bh = find_entry(&dir,basename,namelen,&de);
    ...
}
```

其中 de 就承载着 helloc.txt 这个目录项。

4.1.3.9 找到 helloc.txt 文件的 i 节点

继续看图 4-6，可知得到了 helloc.txt 这个目录项就可以从这个目录项中获取 helloc.txt 这个文件的 i 节点号，同时，当前设备就是硬盘，也能获取设备号，执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    inr = de->inode;
    dev = dir->i_dev;
    ...
}
```

这样，就可以获取 helloc.txt 文件的 i 节点了，仍然是先把硬盘上存储 helloc.txt 文件的 i 节点的数据块读入到高速缓冲块，然后再从中把 helloc.txt 文件的 i 节点信息复制到在内存 i 节点表

中找到的空闲项中。图 4-12 形象地表示了通过 user 目录文件的内容，获取到 helloc.txt 文件的目录项，再得到 helloc.txt 文件的 i 节点的整个过程。

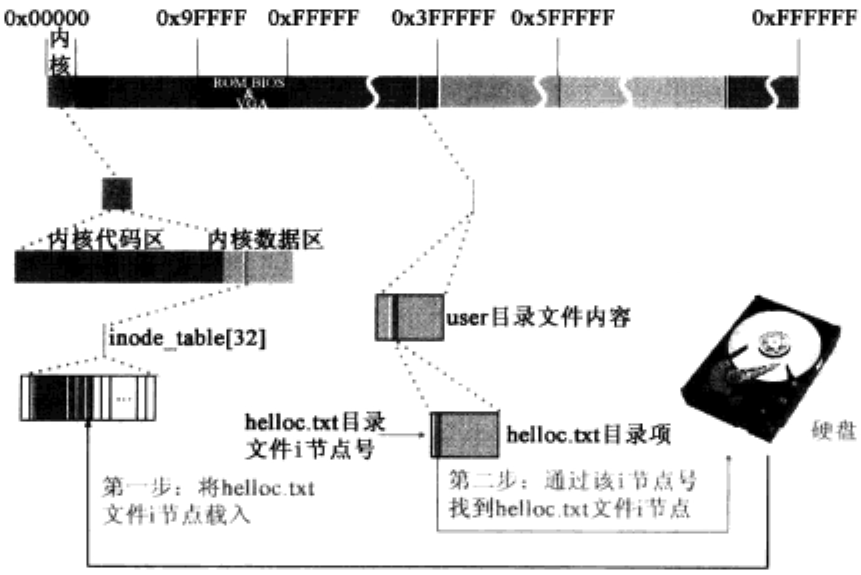


图 4-12　找到 helloc.txt 的 i 节点

执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    if (!(inode=iget(dev,inr)))
    ...
}
```

到这里为止，sys_open 通过调用 open_namei 函数已经得到了 helloc.txt 文件的 i 节点，它的工作也就做完了，继续返回到 sys_open 函数中去执行。这标志着，“打开文件”的第二步，即以用户给定的路径名“/mnt/user/helloc.txt”为线索找到 helloc.txt 文件的 i 节点这项任务已经彻底完成。

4.1.4　将 helloc.txt 文件与文件管理表相挂接

这里开始介绍“打开文件”的第三步，即将 helloc.txt 对应的 i 节点在文件管理表 file_table[64] 中进行登记，从而使进程具备操作 helloc.txt 文件的能力，并最终将 *filp[20] 中，关于“系统文件管理表 file_table[64] 挂接点”的偏移值返回给用户，图 4-13 形象地表示了这个过程。

执行代码如下：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
    ...
    f->f_mode = inode->i_mode;
    f->f_flags = flag;
    f->f_count = 1;
    f->f_inode = inode;
```

```
    f->f_pos = 0;
    return (fd);
}
```

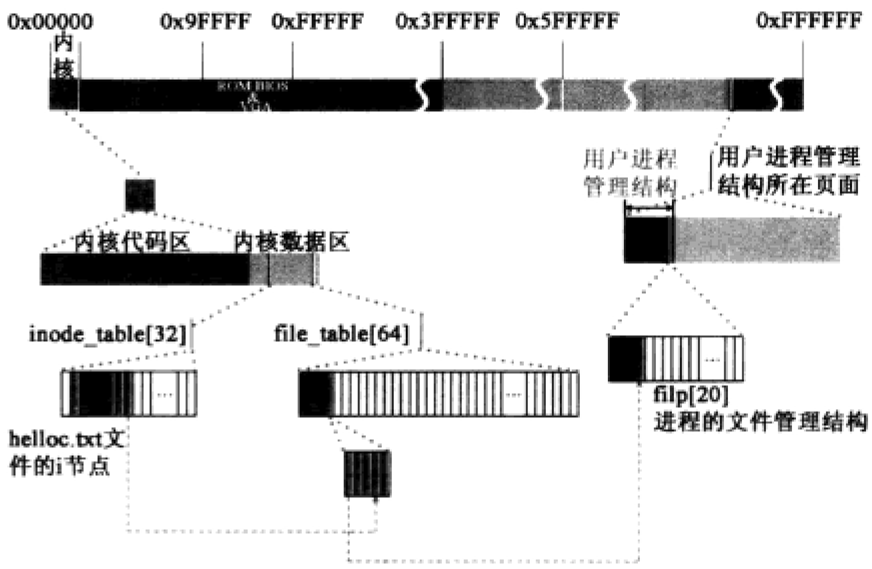


图 4-13　将用户句柄返回给用户

这样，系统中的文件管理表 file_table 与 helloc.txt 文件就建立了关系，又因为在“打开文件”的第一步中当前进程与 file_table 也建立了关系，所以，当前进程就可以通过系统中的文件管理表 file_table 对 helloc.txt 文件进行操作了。本章后续小节所讲述的一切文件操作都是以这个关系的建立为基础的。这一步的完成标志着要实现实例 1 的最终目的所需的第一个要求（要确定读取哪个文件，即“打开文件”）达到了。

4.2　读文件

本节将介绍要实现实例 1 的最终目的所需的第二个要求（要确定读取文件的哪一部分）和第三个要求（把文件读到哪里去）是如何达到的。为达到这两个要求，本节将分为三个步骤进行介绍：

第一步，确定用户读取的数据在硬盘上的“块号”，这一步的目的是为了实现第二个要求，即确定把文件的哪部分内容读出来；

第二步，以“块号”和“设备号”为依据，将指定的数据块从硬盘读入缓冲区中；

第三步，将缓冲区中的数据拷贝到指定的用户程序空间内。

后面两步是为了实现第三个要求，即把读出来的内容存储在用户事先指定的空间内。下面我们来详细地介绍这一过程。本节的程序执行架构如图 4-14 所示：

我们先来看实例 1 中对 read 函数的调用：

```
int size = read(fd,buffer,sizeof(buffer));
```

传递下来的实参将为实现实例 1 最终目的所需的第二个要求和第三个要求提供数据保障。具体表现为：

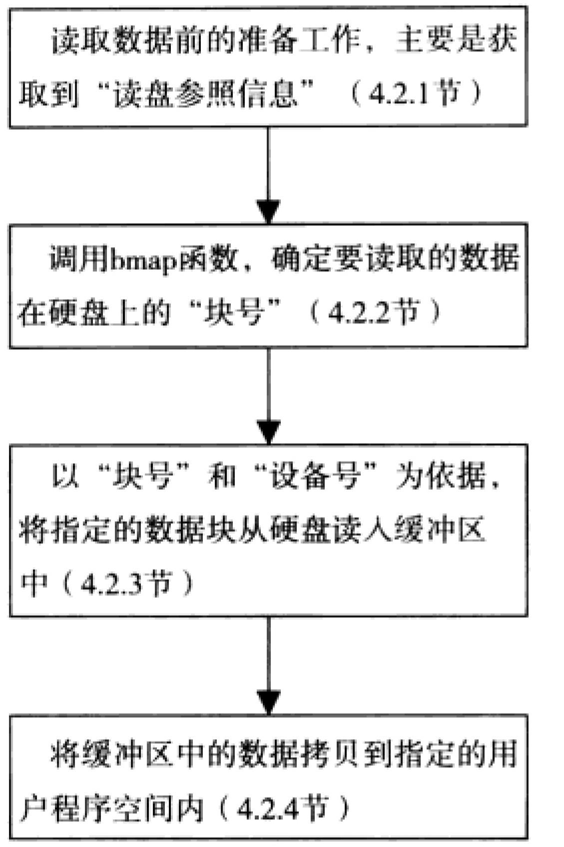


图 4-14　读取文件的程序执行架构图

要实现第二个要求就需要得知用户程序需要从文件的哪个位置开始读取数据，以及读取多少数据。fd 和 sizeof(buffer) 这两个实参为此提供了保障。fd 是文件句柄，在 4.1.1 节和 4.1.4 节中我们已经对文件句柄进行了介绍，通过这个实参可以找到该文件的当前操作指针 f_pos。4.1.4 节中将这个指针的值设置为 0，这就意味着要从文件的起始位置开始读取数据。sizeof(buffer) 是指用户要从文件中读取的字节数。有了这两个参数，第二个要求就可以实现了。当然，fd 还会有其他的用途，我们后面在用到它的时候还会详细介绍。

buffer 这个实参为实现第三个要求提供了保障，buffer 就是实例 1 的程序中 char buffer[12000]; 这个数组的首地址，有了它，第三个要求就可以实现了。

下面我们来具体介绍这些数据都是如何发挥作用的。

4.2.1　为按照用户要求读入文件做准备

这里介绍本节的第一步，即确定用户读取的数据在硬盘上的“块号”。

首先要将库函数 read 最终映射到 sys_read 函数，具体的映射过程与第 4.1.1 节所述的 open 函数映射到 sys_open 函数的过程大体类似。进入 sys_read 函数后，先为实现本节的三个步骤做一些准备工作，即对用户程序中传递参数的“可使用性”进行检测，执行代码如下：

```
// 代码路径：fs/read_write.c：
int sys_read(unsigned int fd,char * buf,int count)
{
    ...
    if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
    ...
}
```

检测的对象包括传递下来的句柄值是否合理，由于 *filp[20] 的数据最多只有 20 项，如果 fd 大于这个数值就不合理了；count 是读取数据的个数，如果是负数就不合理；!(file=current->filp[fd]) 这个代码用以检测当前进程是否真的与某个打开的文件建立了关系，如果这个值是空的，就无法继续操作文件了。

图 4-15 形象地表示了 read 函数库通过引发 int 0x80 软中断最终映射到 sys_read 的过程。

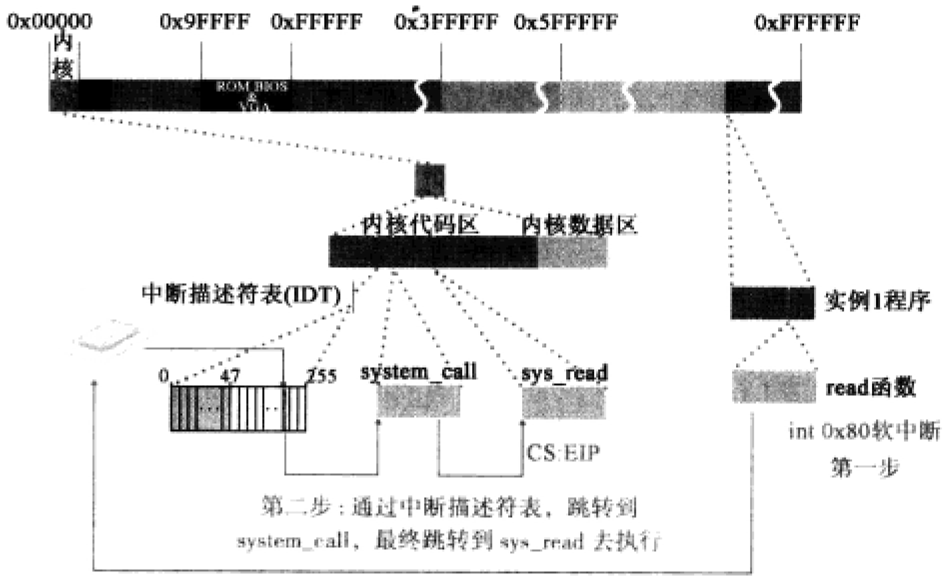


图 4-15　执行文件读取操作前的准备工作

之所以要对这些实参的合理性进行检测，是因为这些实参都是用户程序自己写的，系统事先无法预计用户都写了什么程序，所以只能在这里进行检测。其中，当前进程与 helloc.txt 这个文件的关系如图 4-16 所示：

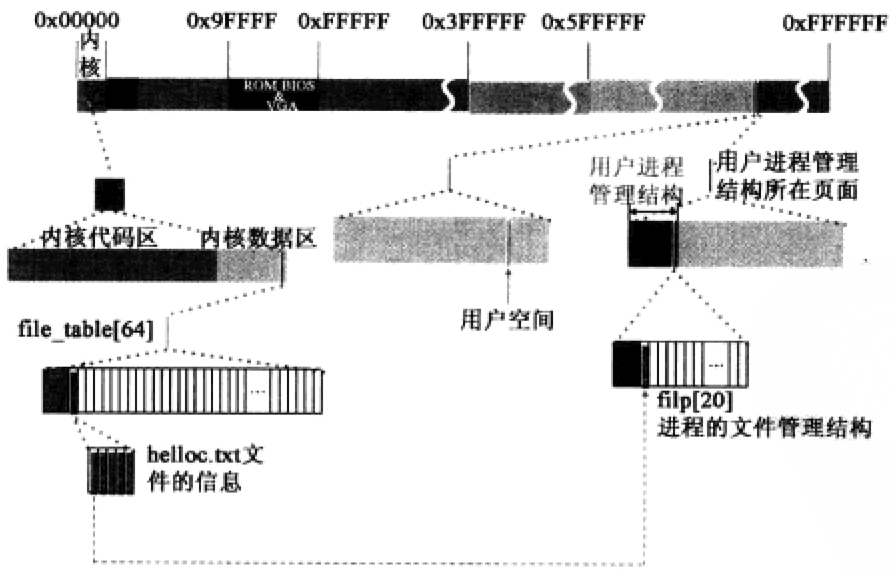


图 4-16　当前进程与 helloc.txt 文件的关系

检测完毕后，就要获取 helloc.txt 这个文件的 i 节点，一个文件只有一个 i 节点，得到了 i 节点就等于掌握了这个文件，执行代码如下：

```
// 代码路径：fs/read_write.c：
int sys_read(unsigned int fd,char * buf,int count)
{
```

```
    ...
    inode = file->f_inode;
    ...
}
```

准备工作完毕了，下面开始调用 file_read 函数来实现本节读取文件内容的三个步骤，执行代码如下：

```
//代码路径：fs/read_write.c：
int sys_read(unsigned int fd,char * buf,int count)
{
    ...
    return file_read(inode,file,buf,count);
    ...
}
```

通过这行代码，我们来看现在系统都得到了哪些信息：

- inode：这是 helloc.txt 的 i 节点指针，有了它就可以掌握 helloc.txt 文件，这个 i 节点就是通过用户程序的实参 fd 获取的；
- file：这是系统中文件管理表 file_table[64] 中 helloc.txt 登记的表项指针，通过这个指针就可以得知这个表项的内容，包括文件的当前指针偏移量 f_pos、文件的引用计数 f_count 等，它也是通过用户程序的实参 fd 获取的；
- buf：这是用户空间的指针，将来在第三步中读取出来的数据就要拷贝到这个空间里面，buf 就是用户程序的实参 buffer；
- count：这是要读取的数据量，count 就是用户程序的实参 sizeof(buffer)。

有了以上这些参数，读取工作完全可以进行了。

4.2.2 确定要读入的数据块的位置

这里将进入 file_read 函数，开始本节的第一步，即确定用户读取的数据在硬盘上的“块号”，执行代码如下：

```
//代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    while (left)
    {
        if (nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE))
        ...
    }
    ...
}
```

本节的三个执行步骤都是在这个循环中完成的，这是因为每次只能处理一个“块”的操作，而用户指定要操作的数据长度往往大于一个块，所以会一直在这个循环中执行，直到把规定的数据长度“left”全部读取出来为止。

第一步的工作是通过 bmap 函数来实现的，在讲解这个函数的工作原理之前，我们先讲解文件的数据在硬盘上的存储方式及管理方式。先来看下面的示意图：图 4-17 至图 4-19。

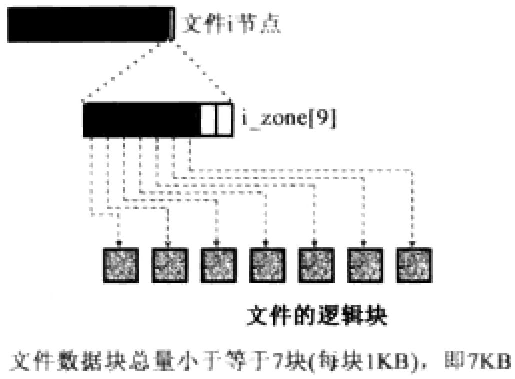


图 4-17　文件数据小于 7 块时 i 节点的管理示意图

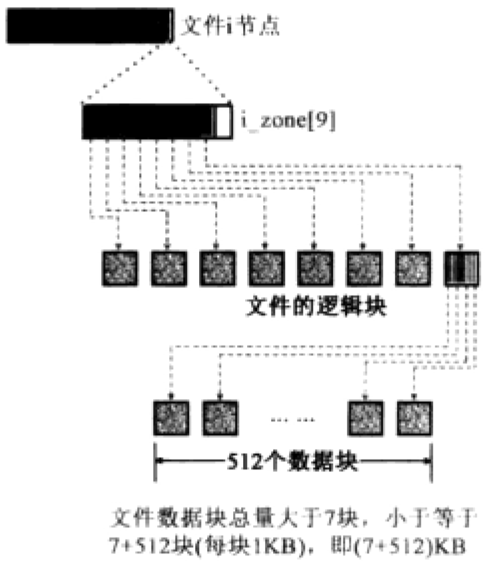


图 4-18　文件数据大于 7，小于 7+512 块时 i 节点的管理示意图

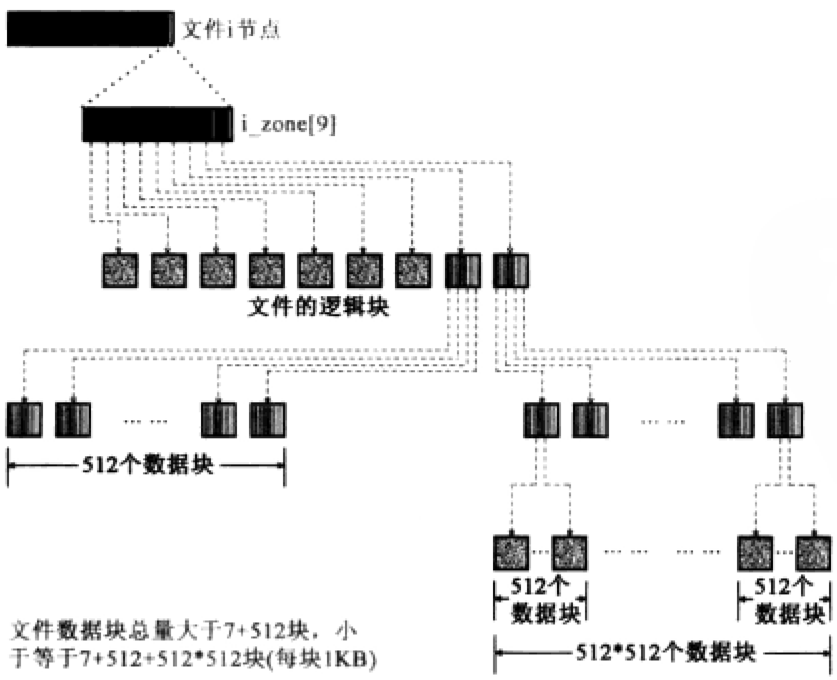


图 4-19　文件数据块大于 7+512，小于 Minix 允许的极限情况的 i 节点的管理示意图

通过图 4-17 至图 4-19 我们可以看出，i 节点的 i_zone[9] 中记录着文件数据块内容的分布情况，但是它毕竟只有 9 个记录空间，随着数据量的增多，肯定是不够用的，所以 Linux 0.11 采取

了一种策略，即在数据区中的数据块内继续存储逻辑块的索引值，以此来分级管理数据块，这样就可以增大管理的数量。

当数据总量小于等于 7KB 时，i_zone[9] 的前七个成员已经足够用了，所以它们就直接记录该文件的这 7 个数据块在数据区的“块号”。

当数据量大于 7KB 时，i_zone[9] 就显得有些捉襟见肘了，于是让其第 8 个成员记录一个数据块的块号，但这个块里面存储的并不是文件内容，而是该文件后续 512 个数据块在数据区中的“块号”，通过这些块号就可以找到相应的数据块，因为一个数据块的大小为 1024 字节，即 1KB，而每个块号需要占用两个字节，所以一个数据块只能存储 512 个“块号”。由此可见，对数据块进行两级管理时，能够管理的极限应该是 7+512 个数据块，即（7+512）KB。

当数据量大于（7+512）KB 时，就要启动二级管理，让其第 9 个成员记录一个数据块的块号。同样，这个块里面存储的并不是文件内容，而是 512 个数据块在数据区中的“块号”。在这 512 个数据块中，存储的仍然不是具体的数据内容，它们还是索引块，每个块里面又都存储着 512 个数据块的块号，这些数据块中存储的才是文件的具体数据内容。由此可见，对数据块进行三级管理时，能够管理的极限应该是 7+512+512*512 个数据块，即（7+512+512*512）KB。

数据块存储的特点我们就介绍完了，现在继续回到代码，看 4.2.2 节开始所描述的第一步是如何实现的。进入 bmap 函数后，间接调用 _bmap 函数，之后就会对 block 的值进行判断。对应的代码如下：

```
//代码路径：fs/inode.c：
static int _bmap(struct m_inode * inode,int block,int create)
{
    ...
    if (block<0)
    ...
    if (block >= 7+512+512*512)
    ...
    if (block<7)
    ...
    if (block<512)
    ...
}
```

这个 block 是调用 bmap 函数时传下来的实参 (filp->f_pos)/BLOCK_SIZE)，在 4.1.4 节中我们已经讲过，文件操作偏移指针 f_pos 被设置为 0，这意味着要从该文件的“第一个块”开始读取，block 现在肯定是 0，于是会执行到 if (block<7) 这个判断条件中来，执行代码如下：

```
//代码路径：fs/inode.c：
static int _bmap(struct m_inode * inode,int block,int create)
{
     ...
     if (block<7)
```

```
        {
                ...
                return inode->i_zone[block];
    }
        ...
    }
```

返回的应该是 helloc.txt 文件的 i 节点中 i_zone[0] 里面存储的"块号"，这样 4.2.2 节第一步完成了。实现实例 1 的最终目的的第二个要求（具体读取文件的哪部分内容），在这里"被完成了一次"。

"被完成了一次"是指：此次只是确定了接下来要从 helloc.txt 文件的起始位置连续读出一个数据块，即 1KB 的数据。但是，仅仅只有 1KB 数据够不够，是否能够满足用户指定的数据量需求，要通过 while 循环对 left 这个变量（count）进行不断的检验才能得知。这有可能需要很多次，从实例 1 的代码中的 buffer[12000] 这个数组的大小来看，肯定不止一次，所以此时只能说"被完成了一次"。

4.2.3　将指定的数据块从硬盘读入到高速缓冲块

这里开始介绍本节第二步，即以"块号"和"设备号"为依据，将指定的数据块从硬盘读入缓冲区中。

实现实例 1 的最终目的的第三个要求（把文件内容读出后，要能够存储在用户事先指定的空间内）是分为两步完成的，第一步就是本节（4.2.3 节）所介绍的，将这指定的数据从硬盘上读进缓冲区；第二步是 4.2.4 节将要介绍的，将数据从缓冲区拷贝到用户事先指定的空间内，即 buffer 这个数组内。所以，这里只是完成了前一半。

我们下面来看具体的实现。

4.2.2 节中最后介绍的 _bmap 函数返回后，经过 bmap 函数，会一直返回到 file_read 函数中继续执行，准备调用 bread 函数，将指定的数据块从硬盘读入到缓冲区中。图 4-20 展示了将 helloc.txt 文件的数据从硬盘上读入到高速缓冲块的过程。

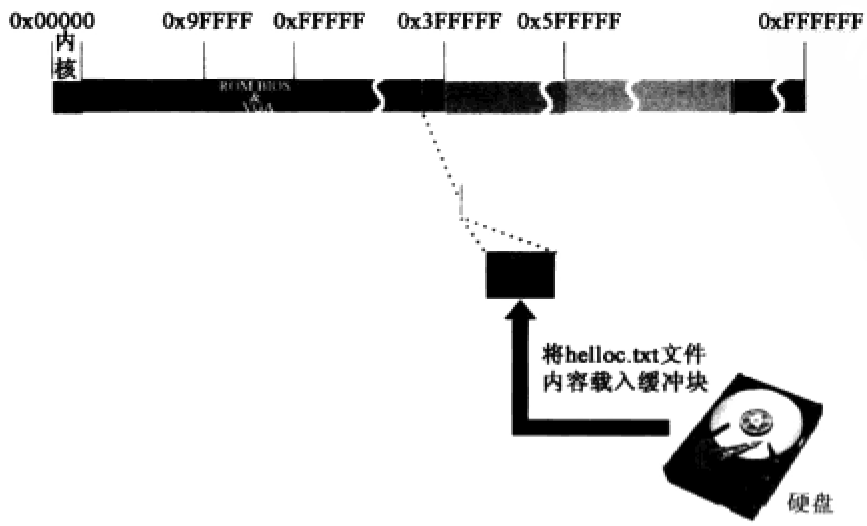


图 4-20　将 helloc.txt 文件的数据读入到高速缓冲区

执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    if (!(bh=bread(inode->i_dev,nr)))
    ...
}
```

这个 bread 函数有两个实参：inode->i_dev 代表硬盘的设备号，nr 代表块号。这两个数据我们都已经获取了，bread 函数开始执行。

bread 函数的执行过程在第 2 章的 2.3 节中讲解读取硬盘引导块的内容时已经详细介绍了，bread 函数执行的步骤与读取引导块执行的步骤完全一致。

这也说明了一个道理，一旦涉及文件系统的底层操作，只要能够指定具体的设备和具体的块号就足够了，具体是哪个文件，文件系统的底层根本不关心。

在发送完读盘指令后，当前进程也会被设置为不可中断等待状态，等数据读入缓冲区后，当前进程再被设置为就绪状态，即被唤醒。因为本节重点讲述的是文件读取的步骤，所以为了突出这个重点，当前进程的状态转换过程我们就不详细介绍了，现在默认执行的数据已经读入缓冲区了。

4.2.4 将数据拷贝到用户指定的内存

这里开始介绍本节的第三步，即将缓冲区中的数据拷贝到指定的用户程序空间内。在 4.2.3 节中只是完成了第三个要求的前一半，现在来完成后一半，即将数据从缓冲区拷贝到 buffer 这个数组中。执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
while (left) {
    ...
            nr = filp->f_pos % BLOCK_SIZE;
                chars = MIN( BLOCK_SIZE-nr , left );
                filp->f_pos += chars;
                left -= chars;
                if (bh) {
                            char * p = nr + bh->b_data;
                            while (chars-->0)
                                put_fs_byte(*(p++),buf++);
                            brelse(bh);
            }
            ...
        }
    }
    ...
}
```

先检测是否把用户程序指定的数据量读完了，执行代码是：

```
// 代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
while (left) {
    ...
        nr = filp->f_pos % BLOCK_SIZE;
        chars = MIN( BLOCK_SIZE-nr , left );
            ...
    }
    ...
}
```

其中 left 是用户程序指定要读出的数据量，从实例 1 的源代码 char buffer[12000]; 中我们可以得知，用户想要读出的数据是 12000 字节，可见现在远远没有读出来，所以 BLOCK_SIZE-nr 一定小于 left，chars 的值此时应该是 BLOCK_SIZE，即 1024。

既然知道了数据还没有读完，接下来就要确定还有多少没读完，执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
while (left) {
    ...
        filp->f_pos += chars;
        left -= chars;
        ...
    }
    ...
}
```

filp->f_pos += chars; 这行语句的意思是，调整当前的文件操作指针 f_pos，以便下次读取后续的数据块，同时 left 的值变为“12000−1024”，说明还差“12000−1024”，即 10976 字节的数据没有读出来。

接下来先把已经读出来的数据拷贝到用户执行空间内，图 4-21 表示了该过程所达到的效果。

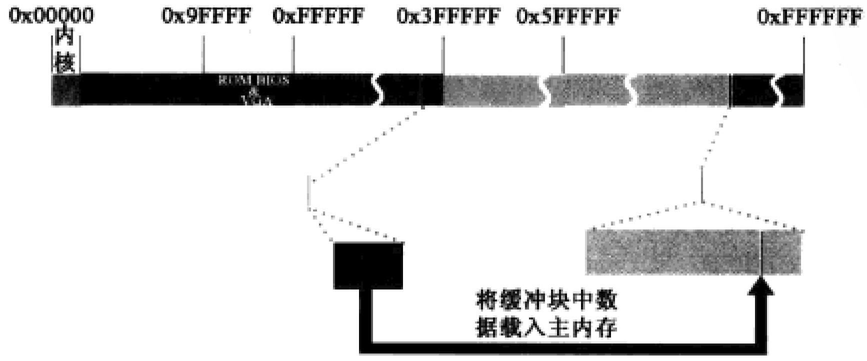


图 4-21　将数据从缓冲块读入到主内存

执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
while (left) {
    ...
        if (bh) {
            char * p = nr + bh->b_data;
            while (chars-->0)
                put_fs_byte(*(p++),buf++);
            brelse(bh);
        }
            ...
    }
    ...
}
```

这样，1KB 大小的数据最终就从硬盘上读入到用户指定的空间内了。实现实例 1 的最终目的的第三个要求（把文件内容读出后，要能够存储在用户事先指定的空间内）已经在这里“被完成了一次”。“被完成了一次”的理由和 4.2.2 节最后部分讲述的理由一样，都是因为仅仅是把 1KB 大小的数据拷贝到了用户指定的空间内，目前尚没有完成用户“读取数据”的全部要求。

接下来，在 while (left) 这个循环中，4.2.2 节至 4.2.4 节中所介绍的内容将被反复地执行，即确定一次要读取的数据，然后把这个数据读出来并拷贝到用户空间内；再次确定要读取的数据，再次读出……直到把用户指定的工作全部完成为止。只有这样，实现实例 1 最终目的的第二个要求和第三个要求才能完成，而且实例 1 的最终目的（将 helloc.txt 这个文件中的内容读出来）才会实现。

值得注意的是，4.2.2 节中介绍的 _bmap 函数执行时会产生变化，因为 block 的值肯定会不断地增大，不可能永远小于 7KB，当它大于 7KB 后，再次进入 _bmap 函数执行时，if (block<512) 这一行下面的代码将会发挥作用，数据块在硬盘上的具体安排方式在 4.2.2 节中的图 4-16 至图 4-18 中已经详细地展示出来了。

到这里为止，实例 1 就全部讲完了，接下来我们详细讲解实例 2 的实现过程。

实例 2：该用户进程在硬盘上新建一个文件，并将内容写入这个文件

本实例的内容分为两部分：“新建文件”和“文件内容写入”，下面我们先来看一下实例 2 对应的用户进程程序使用的代码：

```
void main()
{
char str1[]="Hello, world";
char str2[]="Linux world";
// 新建文件
int fd = creat("/mnt/user/hello.txt",0644));
// 写文件
int size = write(fd,str1,strlen(str1));
}
```

这个进程的程序所要达到的最终目的就是往一个名为 hello.txt 的新建文件中写入内容。而要想实现此目的，就要满足以下要求：

- 要确定在哪里新建哪个文件；
- 要确定把内容写入到文件的什么位置；
- 要确定把哪些内容写入到文件的指定位置。

在 Linux 0.11 中可以向一个已有的文件中写入内容，也可以向一个新建的文件中写入内容，实例 2 所描述的场景就是向一个新建的文件中写入内容。在这个进程程序中对 creat 和 write 这两个库函数的调用就是为了满足以上要求，从而最终实现写入文件内容的目的。

- creat 函数的核心目的就是实现第一个要求，我们将在第 4.3 节中详细介绍。
- write 函数的核心目的就是实现第二个要求和第三个要求，我们将在第 4.4 节中详细介绍。

值得注意的是，实例 2 中介绍的向文件中写入内容的操作场景还仅仅是操作系统中最简单的文件写入场景。除此之外，还有更为复杂的场景，比如说，在一个已有文件的任意位置写入内容等，这种场景的实现过程我们将在第 4.5 节中介绍。

下面，我们来看实例 2，即在操作系统中新建和写入文件的具体实现方法。

4.3 新建文件

本节要实现实例 2 中的第一个要求，即要确定在哪里新建哪个文件。这个要求显然包括了两个方面的内容：一个是“哪个位置”，另一个是“哪个文件”。

其中“哪个文件”这个要求，即便是在实例 1 读文件时也需要事先确定下来。由此可见，在 Linux 0.11 中，无论是读文件还是写文件，只要是操作文件，都要在操作之前确定目标，即确定要操作哪个文件。

其中“哪个位置”是个新的要求，毕竟需要新建一个文件，要为它做一些安排。

通过以上对实例 2 中的第一个要求的分析，我们几乎可以推测出，本节所介绍的内容应该与 4.1 节所介绍的内容在很大程度上是一致的，同时在“新建”文件方面又存在着一些特殊的处理操作。

下面我们来看一下本节的具体操作步骤。新建的步骤和 4.1 节中打开文件的过程大体是一致的，也是分为三步：

第一步：将用户进程管理结构 task_struct 中的文件管理指针表 *filp[20] 与内核中的文件管理表 file_table[64] 进行挂接；

第二步：以用户给定的路径名“/mnt/user/hello.txt”为线索，新建 hello.txt 文件的目录项及其对应的 i 节点；

第三步：将 hello.txt 对应的 i 节点在文件管理表 file_table[64] 中进行登记，从而使进程具备操作 hello.txt 文件的能力。

以上三步操作将能完成“哪个文件”这个要求，即让进程得知要新建一个名为 hello.txt 的文

件。此要求的核心内容是为该文件创建一个新的 i 节点，这是在第二步中完成的。

“哪个位置”这个要求也是在第二步中完成的，即为该文件创建一个目录项。

本节内容的架构图如图 4-22 所示：

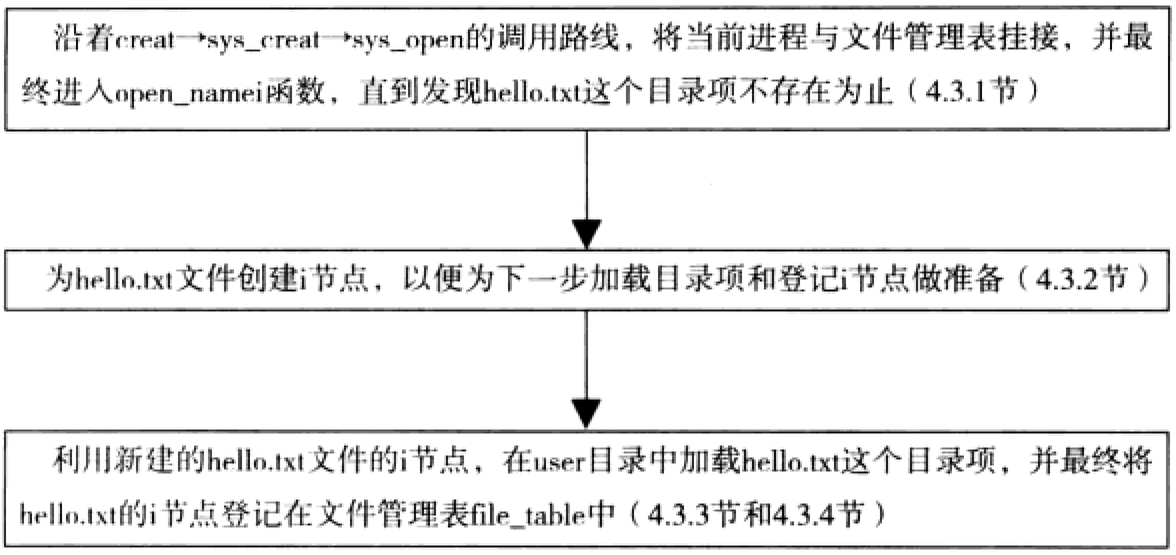


图 4-22　新建文件执行过程架构图

通过该架构图可以看出，本节的内容主要是在 sys_open 这个函数中实现的，与本章第 4.1 节中的内容基本一致，唯一的区别在于第二步中针对“新建文件”进行的特殊处理。因此第二步的操作将成为本节讲解的重点，下面开始详细介绍。

4.3.1　查找路径“/mnt/user/hello.txt”

用户程序是通过调用 creat 这个库函数来新建文件的，这个库函数最终映射到 sys_creat 函数中，映射过程与 4.1 节 open 函数映射到 sys_open 函数的过程大体类似。进入 sys_creat 函数后，调用 sys_open 函数，开始新建文件的第一步，执行代码如下：

```
// 代码路径：fs/open.c:
int sys_creat(const char * pathname, int mode)
{
    return sys_open(pathname, O_CREAT | O_TRUNC, mode);
}
```

值得注意的是“O_CREAT | O_TRUNC”这个实参的传递，这个实参的作用是准备新建路径名中指定的文件，这里指的就是新建 hello.txt 文件。

进入 sys_open 函数后，开始执行第一步，即将用户进程管理结构 task_struct 中的文件管理指针表 *filp[20] 与内核中的文件管理表 file_table[64] 进行挂接，具体步骤我们在本章 4.1.2 节中已经讲解了。

接下来执行第二步，即以用户给定的路径名“/mnt/user/helloc.txt”为线索，查找 hello.txt 文件的的 i 节点。区别就在第二步中，第二步是由 open_namei 函数完成的，进入该函数后，起初的执行与第 4.1.3 节中介绍的步骤完全一致，即通过调用 dir_namei 函数找到“/mnt/user/helloc.txt”路径名中的枝梢 i 节点，即 user 目录文件的 i 节点，对应代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
      ...
      if (!(dir = dir_namei(pathname,&namelen,&basename)))
      ...
}
```

获取后，调用 find_entry 函数，根据 user 目录文件的 i 节点找到 user 目录文件，并将该目录文件中名为 hello.txt 的目录项读取出来，执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
     struct m_inode ** res_inode)
{
      ...
      bh = find_entry(&dir,basename,namelen,&de);
      ...
}
```

区别就从这里产生了，由于 user 目录文件中现在还没有 hello.txt 这个目录项，所以函数的返回值 bh 一定是 NULL，于是执行到下一行中，图 4-23 形象地反映了这个过程。可以看到，图中所示的 user 目录数据中并没有名为 hello.txt 的目录项。

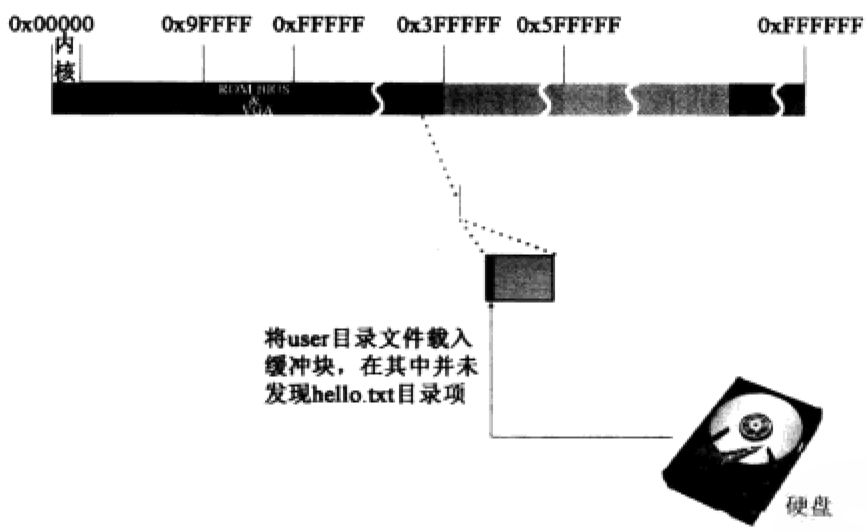


图 4-23　未找到 hello.txt 目录项

对应代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
     struct m_inode ** res_inode)
{
     ...
     if (!bh)
     ...
}
```

接下来，就开始对“hello.txt 这个目录项不存在”的情况进行具体的处理。本节的重点内容“要确定在哪里新建哪个文件”中的“哪里”和“哪个文件”这两个要求的核心内容从下面就开

始实现了。

4.3.2 为 hello.txt 文件新建一个 i 节点

本节的开始部分我们已经介绍过，“哪个文件”这个要求的核心内容就是为该文件创建一个 i 节点。现在就要为 hello.txt 文件创建一个 i 节点。

执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    if (!(flag & O_CREAT))
    ...
}
```

这个 flag 标志指明了此次操作的性质，我们在 4.3.1 节中开始的位置介绍 sys_creat 传参的时候曾经强调了“O_CREAT | O_TRUNC”这个实参的作用就是要创建一个新文件。而 flag 的值就是“O_CREAT | O_TRUNC”，这表明用户确实是要新建一个文件，于是才会开始创建 hello.txt 这个文件。假设此时 flag 的值不是该值，那就说明用户此时就是想打开一个不存在的文件，用户的要求本身就不合理，于是会给用户返回错误信息。

之后，就通过调用 new_inode 函数创建一个新的 i 节点，图 4-24 表示了为 hello.txt 文件的 i 节点在内核 i 节点表和硬盘上分别找到一个存储位置的过程。

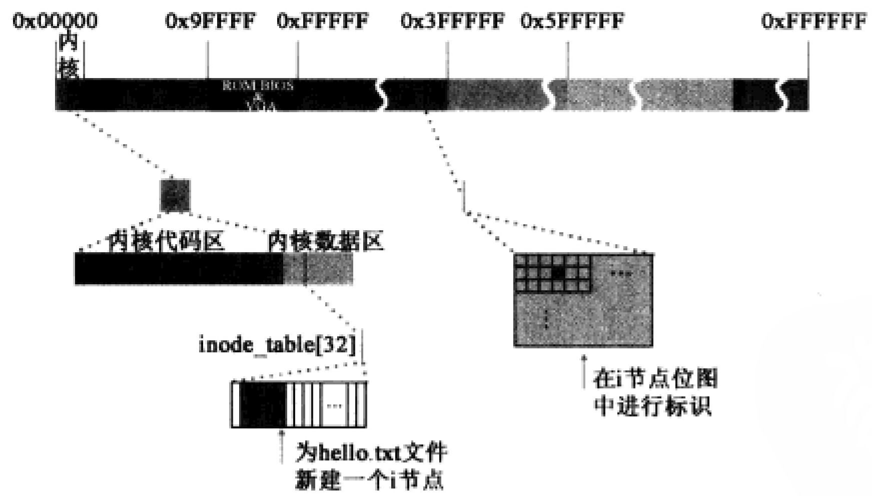


图 4-24　为 hello.txt 文件新建一个 i 节点

执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    inode = new_inode(dir->i_dev);
    ...
}
```

进入 new_inode 函数后，要先在系统的 i 节点管理表 inode_table 中申请一个空闲项，执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * new_inode(int dev)
{
      ...
      if (!(inode=get_empty_inode()))
      ...
}
```

之后，读取硬盘的超级块，由于硬盘的超级块已经在第 3 章加载文件系统时载入到了系统的超级块管理结构 super_block[8] 中，所以可以从该结构中直接获取它，获取超级块的目的，就是要通过它对硬盘的 i 节点位图进行进一步操作。执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * new_inode(int dev)
{
      ...
      if (!(sb = get_super(dev)))
      ...
}
```

我们在第 3 章中已经介绍过，硬盘的 i 节点位图已经载入缓冲区了，所以找到超级块后，通过搜索超级块上 i 节点位图的指针表 s_imap[8] 就可以在缓冲区里面的 i 节点位图上为 hello.txt 文件的 i 节点找到一个空闲位置并将其置位，同时也获得了 i 节点号，执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * new_inode(int dev)
{
    ...
    for (i=0 ; i<8 ; i++)
        if (bh=sb->s_imap[i])
            if ((j=find_first_zero(bh->b_data))<8192)
                break;
    ...
    if (set_bit(j,bh->b_data))
    ...
}
```

最后，对刚刚在系统的 i 节点管理表 inode_table 中申请的空闲项进行设置，这个空闲项将不再空闲，它将作为 hello.txt 目录文件的 i 节点存放于系统中的 i 节点管理表 inode_table 中，执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * new_inode(int dev)
{
    ...
    inode->i_count=1;
    inode->i_nlinks=1;
    inode->i_dev=dev;
```

```
        inode->i_uid=current->euid;
        inode->i_gid=current->egid;
        inode->i_dirt=1;
        inode->i_num = j + i*8192;
        inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
        ...
    }
```

需要注意的是，上述设置过程并没有对内存 i 节点结构的所有字段进行设置，其他还没有被赋值的字段，由于在 get_empty_inode 中已经提前把空闲 i 节点所占的存储区域全部置为 0，因此这些未赋值字段的默认值为 0，保证了 i 节点数据不会出错。

新的 i 节点已经创建完成了，于是将继续返回到 open_namei 函数中去执行。我们得知，这个 i 节点并没有写入到硬盘的 i 节点表中，而是存放在内存中 i 节点的管理表 inode_table 中，待 hello.txt 文件创建完毕后，这个 i 节点将直接在内存中的文件管理表 file_table[64] 中进行登记。

4.3.3 为 hello.txt 文件新建目录项

这里开始实现“要确定在哪里新建哪个文件”中的“哪里”这个要求，即为该文件创建一个目录项。这里通过调用 add_entry 函数开始为 hello.txt 文件创建一个目录项，执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    bh = add_entry(dir,basename,namelen,&de);
    ...
}
```

我们先来看一下图 4-25 中所示的目录文件的示意图。

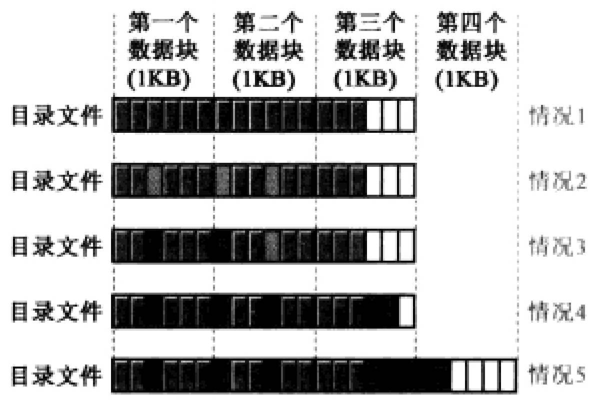


图 4-25　目录文件示意图

图中的情况 1 为一个目录文件的初始状态；情况 2 是将其中的目录项删除（删除的本质就是将目录项中的 i 节点号清 0）；情况 3、情况 4 和情况 5 都是不断地加载目录项时出现的情况。

通过对 4.3.1 节中的路径分析我们得知，hello.txt 的目录项要加载在 user 这个目录文件中。这里我们不妨假设 user 目录文件的初始情况如情况 2 所示，下面详细介绍进入 add_entry 函数后的创建情况。

进入该函数后，先要把 user 目录文件的第一个数据块读出来，看看能不能在里面创建目录项，执行代码如下：

```
// 代码路径：fs/namei.c：
static struct buffer_head * add_entry(struct m_inode * dir,
    const char * name, int namelen, struct dir_entry ** res_dir)
 {
    ...
    if (!(block = dir->i_zone[0]))
    ...
    if (!(bh = bread(dir->i_dev,block)))
    ...
    i = 0;
    de = (struct dir_entry *) bh->b_data;
    ...
}
```

然后进入 while 循环，开始根据不同的情况寻找空闲位置，准备加载目录项，图 4-26 分别展示了在 user 目录数据中寻找一个空闲项的动作。

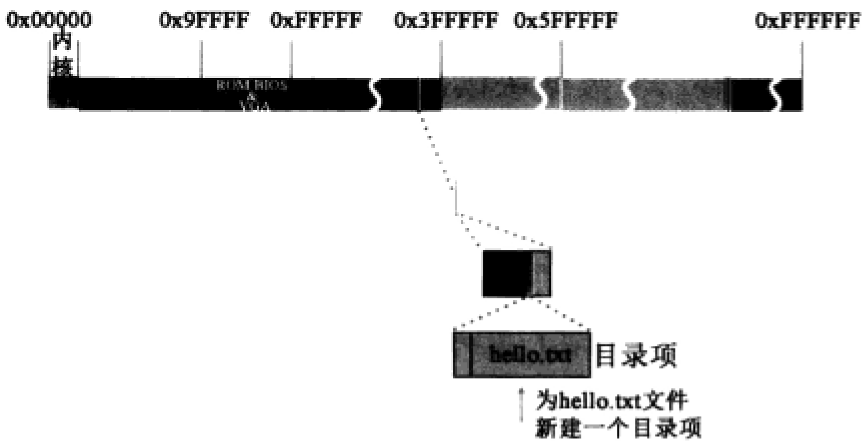


图 4-26　查找空目录项并添加目录数据

执行代码如下：

```
// 代码路径：fs/namei.c：
static struct buffer_head * add_entry(struct m_inode * dir,
    const char * name, int namelen, struct dir_entry ** res_dir)
{
    ...
    while (1) {
        if ((char *)de >= BLOCK_SIZE+bh->b_data) {
            brelse(bh);
            bh = NULL;
            block = create_block(dir,i/DIR_ENTRIES_PER_BLOCK);
            if (!block)
                return NULL;
                if (!(bh = bread(dir->i_dev,block))) {
                    i += DIR_ENTRIES_PER_BLOCK;
                    continue;
                }
                de = (struct dir_entry *) bh->b_data;
         }
```

```
18        if (i*sizeof(struct dir_entry) >= dir->i_size) {
19            de->inode=0;
20            dir->i_size = (i+1)*sizeof(struct dir_entry);
21            dir->i_dirt = 1;
22            dir->i_ctime = CURRENT_TIME;
23        }
24        if (!de->inode) {
25            dir->i_mtime = CURRENT_TIME;
26            for (i=0; i < NAME_LEN ; i++)
27                de->name[i]=(i<namelen)?get_fs_byte(name+i):0;
28            bh->b_dirt = 1;
29            *res_dir = de;
30            return bh;
31        }
32        de++;
33        i++;
34    }
35    ...
36}
```

其中：

这段代码的第 6 行～第 17 行意思是：目录文件（很可能含有多个数据块）第一个数据块中没有空间加载目录项，那就把第二个数据块读出来；如果还不能加载，就把第三个数据块读出来，以此类推。如果最后整个目录文件中的所有数据块都读出来了，还是没有加载该目录项的空间，就新建一个数据块，准备加载目录项。

第 18 行～第 23 行代码适用于图 4-25 的的第 5 种情况，即整个目录文件中都没有如图 4-25 的情况 2 中所示的空位来加载新的目录项，但最后一个数据块尾端还有空间加载该目录项，那就准备在尾端加载一个目录项。

第 24 行～第 31 行代码是要执行加载目录项的具体动作。在遍历目录文件的目录项过程中，只要发现哪个目录项的 i 节点号为 0（在 Linux 0.11 中，只要目录文件的目录项中 i 节点号为 0，就被认定为“空目录项”），就将新的目录项加载到这个空目录项所在的位置处。

我们现在假设 user 目录文件是图 4-25 中的第 2 种情况，那么在循环两次后，执行到 if (!de->inode) 这里时，开始在目录文件的第三项创建 hello.txt 这个目录项，执行代码如下：

```
//代码路径：fs/namei.c：
static struct buffer_head * add_entry(struct m_inode * dir,
    const char * name, int namelen, struct dir_entry ** res_dir)
{
    ...
    while (1) {
    ...
        if (!de->inode) {
            dir->i_mtime = CURRENT_TIME;
            for (i=0; i < NAME_LEN ; i++)
                de->name[i]=(i<namelen)?get_fs_byte(name+i):0;
            bh->b_dirt = 1;
            *res_dir = de;
            return bh;
```

```
        }
    ...
    }
    ...
}
```

其作用是先将 hello.txt 这个字符串载入目录项，以此作为该目录项的名字，如图 4-26 中的目录结构所示。然后，将该目录项所在的缓冲块的 b_dirt 标记设置为 1，以便将来从缓冲块中写到硬盘上（写文件的具体步骤我们会在 4.4 节中介绍），之后该函数就返回了。

4.3.4 完成 hello.txt 新建操作并返回给用户进程

这里继续完成创建目录项的工作。hello.txt 这个目录项已经定位，字符串也设置了，但还差 i 节点号没有设置，目录项还不完整。因此，在 add_entry 函数返回后，还要将 hello.txt 文件创建的 i 节点的节点号载入到 hello.txt 目录项中，图 4-27 左下方表示了用 hello.txt 对应的 i 节点号对目录项进行设置的动作。

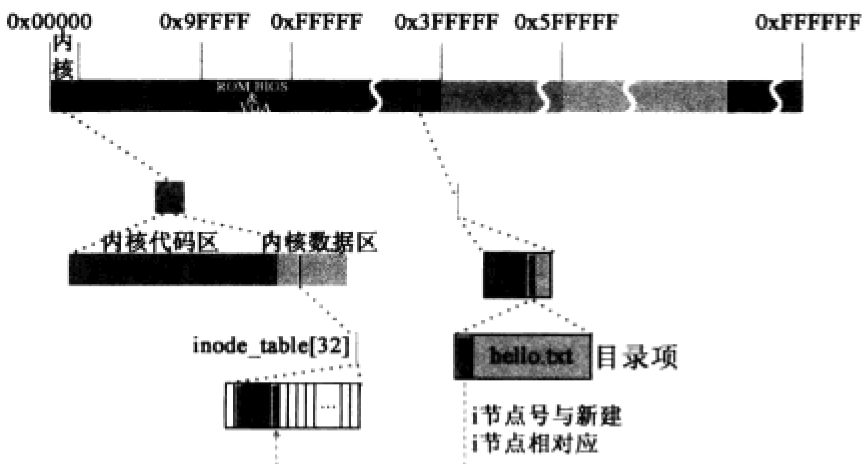


图 4-27 完成对 hello.txt 目录项的设置

执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    de->inode = inode->i_num;
    ...
}
```

这样，通过 4.3.3 节和 4.3.4 节的工作，“要确定在哪里新建哪个文件”中的“哪里”这个要求（为该文件创建一个目录项）就实现了，这个目录项被加载到了 user 这个目录文件中。今后，通过 hello.txt 这个目录项就能够找到该文件的 i 节点，最终就能够找到这个文件的具体位置。

之后，open_namei 函数就返回。到这里为止，本节第二步工作就完成了，实例 2 中的第一个要求（要确定在哪里新建哪个文件）也就实现了。通过代码我们也可以看出，新建一个 hello.txt 文件和打开一个已有的 hello.txt 文件的最大区别就是第二步中创建新的 i 节点和创建新的目录项

这两项工作，其余的与 4.1 节中介绍的打开文件的步骤基本一致。

返回到 sys_open 函数中继续完成第三步，即将 hello.txt 对应的 i 节点在文件管理表 file_table[64] 中进行登记，从而使进程具备操作 hello.txt 文件的能力，图 4-28 表示了将 hello.txt 文件的 i 节点与文件描述符表进行挂接并返回给用户进程的方式。

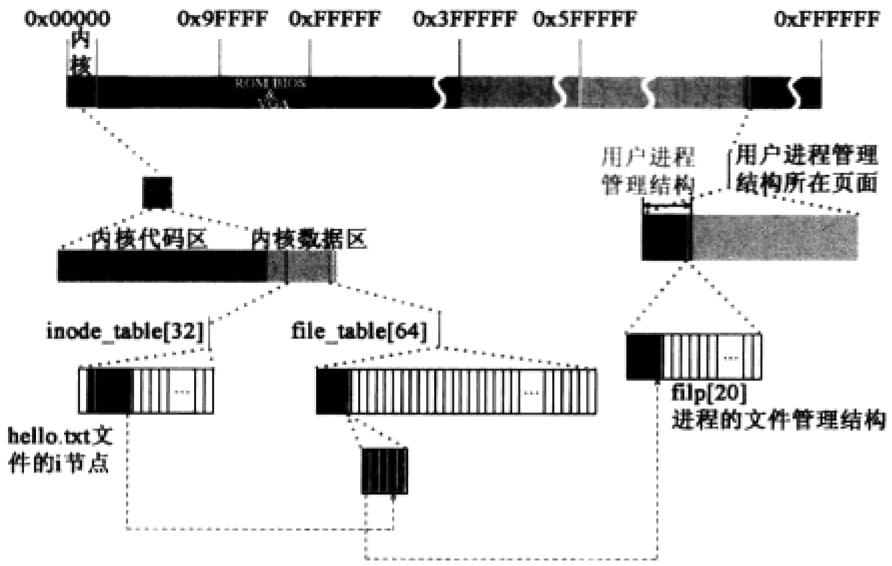


图 4-28　将文件句柄返回给用户进程

执行代码如下：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
    ...
    f->f_mode = inode->i_mode;
    f->f_flags = flag;
    f->f_count = 1;
    f->f_inode = inode;
    f->f_pos = 0;
    return (fd);
}
```

这个步骤我们在 4.1 节中已经讲解过。

最后，通过 sys_creat 函数返回 hello.txt 文件的句柄，文件创建的工作就全部结束了。

实例 2 中的进程不仅确定了要把 hello.txt 创建在什么位置，而且还具备了操作该文件的能力，这就为实现实例 2 中的第二个要求和第三个要求创造了条件。

值得注意的是，新创建的 i 节点和新创建的目录项还在内存中存放着，并没有写入到硬盘上。那么，文件的内容又是怎样写入硬盘的呢？带着这个问题，我们来看下一节的内容——写文件。

4.4　写文件

本节要实现实例 2 中的第二个要求和第三个要求，即要确定把内容写入到文件的什么位置，以及要确定把哪些内容写入到文件的指定位置。写入过程分为三个步骤。

第一步：确定在 hello.txt 文件中写入的位置。这一步是为了实现第二个要求，即要确定把内容写入到文件的什么位置。

第二步：将第一步确定的位置所在的数据块读取到缓冲区中。

第三步：将程序规定的数据写入到指定的缓冲块中。这一步是为了实现第三个要求，即把哪些内容写入到文件的指定位置处。

下面我们开始详细介绍这些步骤的实现过程。

本节的程序架构图如图 4-29 所示：

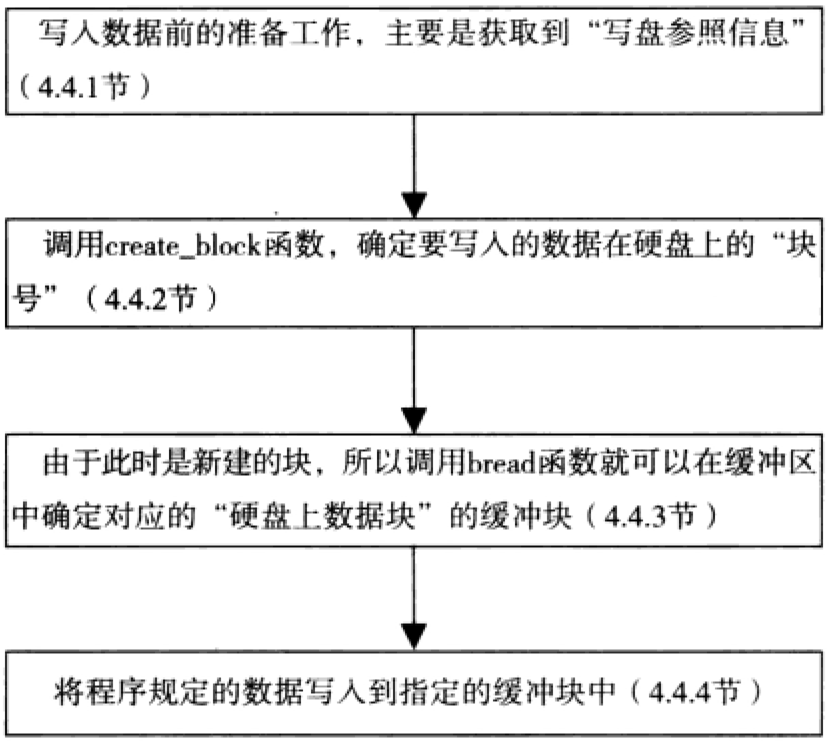


图 4-29　写文件操作的程序架构图

我们先来看实例 2 中对 write 函数的调用：

```
int size = write(fd,str1,strlen(str1));
```

这条语句传递下来的实参将为实现实例 2 的最终目的所需第二个要求和第三个要求提供数据保障，具体表现为：

要实现第二个要求，就需要得知用户程序需要从文件的哪个位置开始写入数据，以及写入多少数据。fd 和 strlen(str1) 这两个实参为此提供了保障。fd 是文件句柄，在 4.1.4 节中我们已经对文件句柄进行了介绍，通过这个实参，可以找到该文件的当前操作指针 f_pos，4.3.4 节中已经将这个指针值设置为 0，这就意味着要从文件的起始位置开始写入数据。（注意：我们在介绍实例 2 时已经说明，这个实例仅仅是文件写入的最基本案例，因为 hello.txt 是新建的文件，所以实例 2 的环境默认从文件的起始位置开始写入，其实这个指针是可以由用户进行设置的，以便完成更为复杂的文件写入任务，这一点我们将在 4.5 节中详细介绍）。strlen(str1)：是指用户要写入的字节数。有了这两个参数，第二个要求就可以实现了。当然，fd 还会有其他的用途，我们后面在用到它的时候还会详细介绍。

str1 这个实参为实现第三个要求提供了保障。str1 就是实例 2 的程序中 char str1[] 这个数组的首地址，有了它，第三个要求就可以实现了，即把“Hello, world”这个字符串从文件的起始位置开始写入。

下面我们来具体介绍这些数据都是如何发挥作用的。

4.4.1 文件写入前的准备工作

先来关注一下写入的准备工作。在写入文件前，当前已经已经与 hello.txt 文件建立了关联，它们之间的关系如图 4-30 所示：

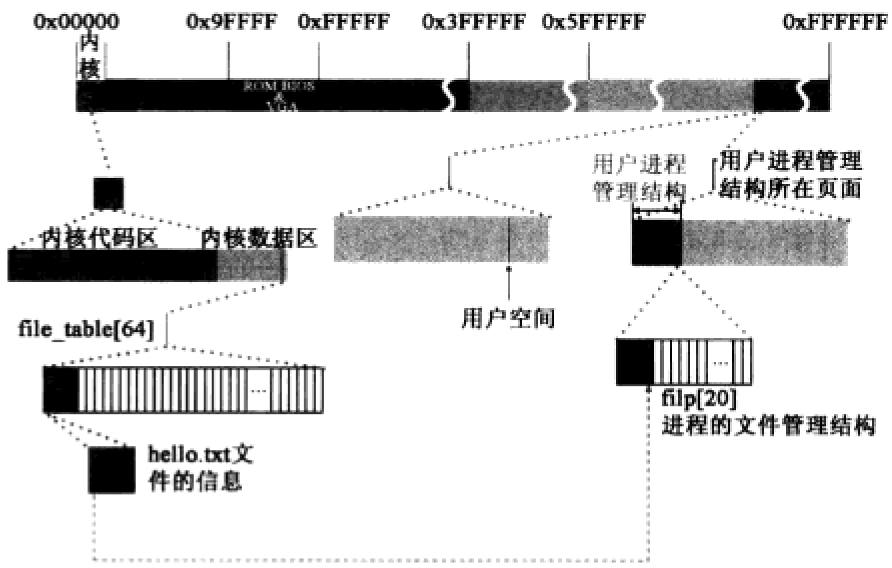


图 4-30 当前进程与 hello.txt 文件的关系

写入工作是通过调用 write 库函数完成的，调用后产生软中断，最终映射到 sys_write 函数中去执行，映射的过程与 4.2 节中 read 函数映射到 sys_read 函数中的步骤大体类似。进入 sys_write 函数后，要检测用户程序传递参数的合理性，然后获取 hello.txt 文件的 i 节点，并最终到 file_write 函数中去执行，整个写入的过程都是在 file_write 函数中完成的，执行代码如下：

```
// 代码路径：fs/read_write.c：
int sys_write(unsigned int fd,char * buf,int count)
{
    ...
    if (fd>=NR_OPEN || count <0 || !(file=current->filp[fd]))
...
inode=file->f_inode;
...
return file_write(inode,file,buf,count);
...
}
```

通过“file_write(inode,file,buf,count)”这行代码，我们来看现在系统都得到了哪些信息。

- inode：这是 hello.txt 的 i 节点指针，有了它就可以掌握 hello.txt 文件，这个 i 节点最终就是通过用户程序的实参 fd 获取的；
- file：这是系统中的文件管理表 file_table[64] 中 hello.txt 登记的表项指针，通过这个指针就可以得知这个表项的内容，包括文件的当前指针偏移量 f_pos、文件引用计数 f_count 等，

这个 file 最终也是通过用户程序的实参 fd 获取的；

- buf：这是用户空间的指针，将来第三步向文件中写入数据时，就从这个空间里面拷贝，buf 就是用户程序的实参 str1；
- count：这是要写入的数据量，count 就是用户程序的实参 strlen(str1)。

这些步骤和进入 sys_read 后执行的情况大体类似，已经在 4.2.1 节中说明。

4.4.2 确定 hello.txt 文件的写入位置

这里我们开始介绍本节的第一步工作，即确定在 hello.txt 文件中写入的位置，这是为了完成实例 2 中的第二个要求，即要确定把内容写入到文件的什么位置。

接下来，就根据 f_pos 的值来确定文件在磁盘上的写入位置。

进入 file_write 函数后，通过调用 create_block 函数来完成第一步的工作，执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
      ...
      if (!(block = create_block(inode,pos/BLOCK_SIZE)))
      ...
}
```

create_block 函数展开后，我们不难发现，它里面调用的函数是 _bmap，这与 4.2.2 节中 bmap 函数内部的情况完全一致，bmap 函数也是调用 _bmap 函数，_bmap 函数的功能就是通过分析文件的 i 节点中的信息以及文件的当前操作指针来确定需要操作的数据块。它的执行步骤我们在 4.2.2 中已经说明了。

这里值得关注的是，create_block 函数和 bmap 函数在调用 _bmap 函数时传递的实参是不一样的。我们分别来看一下这两个函数调用 _bmap 函数时的传参情况，代码如下：

```
// 代码路径：fs/inode.c：
int create_block(struct m_inode * inode, int block)
{
    return _bmap(inode,block,1);
}

int bmap(struct m_inode * inode,int block)
{
    return _bmap(inode,block,0);
}
```

很显然，最后一个参数不一样，这个参数是一个“新建数据块”标识。

现在执行的是写操作，create_block 函数是用来确定将要进行“写入操作”的数据块的，以便实现实例 2 中的第二个要求。

从实例 2 的场景实际情况来看，hello.txt 这个文件是一个新建的文件，这个文件目前还没有任何内容，即在硬盘上还没有任何数据块是属于它的，所以就需要新建一个数据块。这样一来，create_block 函数在调用 _bmap 函数时，最后一个参数，即“新建数据块”标识被设置为 1，这

就意味着进入 _bmap 函数后可能要新建一个数据块。

再看实例 1 的场景，很显然是读文件的操作，任何情况下，读操作都不可能需要新建数据块。这样一来，bmap 函数调用 _bmap 函数时，最后一个参数，即“新建数据块”标识被设置为 0，这就意味着进入 _bmap 函数后，绝对不可能需要新建一个数据块。

这就是 create_block 函数和 bmap 函数的根本区别。

当然，create_block 函数调用 _bmap 函数并进入 _bmap 后，“可能”要新建一个数据块，而非“一定”要新建一个数据块。这是因为，实例 2 的场景说到底还是一个比较简单的写盘场景，这一点我们在介绍实例 2 时已经说明了。如果场景稍微复杂一些，就有可能不需要新建一个数据块了。比如说，往一个已经存在的文件中写入数据，而且文件的当前操作指针 f_pos 并不是文件的起始位置，而是处于某个已经存在的数据块上，在这种情况下，就显然不能通过新建数据块来为内容的写入服务了，直接锁定这个已有数据块就可以了，这种较为复杂的情况，我们将在 4.5 节中介绍。

这样，create_block 函数就采取了将“新建数据块”标识设置为 1 的策略，即先把这种可能的需求通知给 _bmap 函数，_bmap 函数根据具体情况进行处理，需要新建数据块就新建，不需要就不新建。无论如何，只要 _bmap 函数能够确定一个数据块，能够完成实例 2 中的第二个要求（要确定把内容写入到文件的什么位置）就可以了。

我们来看具体的执行情况：由于 hello.txt 文件是一个新建的文件，所以它没有任何数据块，这时候“新建数据块”标识就发挥作用了，_bmap 函数将新建一个数据块，执行代码如下：

```
//代码路径：fs/inode.c:
static int _bmap(struct m_inode * inode,int block,int create)
{
    ...
    if (block<7) {
        if (create && !inode->i_zone[block])
            if (inode->i_zone[block]=new_block(inode->i_dev)) {
                inode->i_ctime=CURRENT_TIME;
                inode->i_dirt=1;
            }
        return inode->i_zone[block];
    }
    ...
}
```

其中 new_block 函数将负责新建一个数据块。

进入 new_block 函数后，先要通过缓冲区中常驻的逻辑块位图结构（这个结构我们在第 3 章中已经介绍，通过它就能够了解到硬盘上现在都有哪些数据块是空闲的，哪些数据块已被占用），找到一个空闲的逻辑块，在硬盘上找到一个空闲逻辑块的操作如图 4-31 所示。

执行代码如下：

```
//代码路径：fs/bitmap.c:
int new_block(int dev)
{
     ...
```

```
        j += i*8192 + sb->s_firstdatazone-1;
        ...
}
```

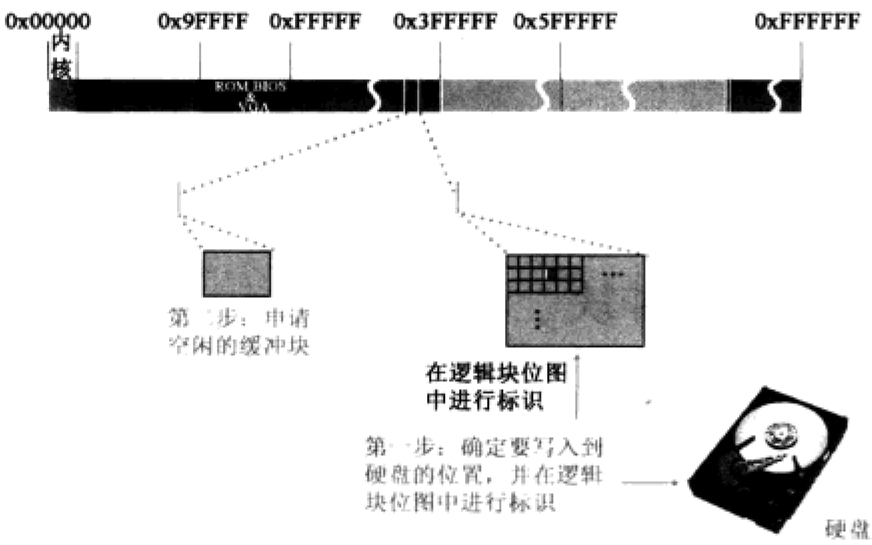


图 4-31　在逻辑块位图中找到空闲的数据块

之后，通过调用 getblk 函数，将这个空闲的逻辑块在缓冲区管理结构中注册（注册的方式我们在 2.3 节中介绍读取硬盘引导块的内容时已经讲过，这个管理结构对每一个缓冲块进行管理，记载着缓冲块具体都对应着哪个外设的哪个数据块），并进行相应设置，执行代码如下：

```
// 代码路径：fs/bitmap.c：
int new_block(int dev)
{
        ...
        if (!(bh=getblk(dev,j)))
        ...
}
```

最后，将缓冲块所在的内存中的 1KB 空间清 0，然后逻辑块的块号返回，执行代码如下：

```
// 代码路径：fs/bitmap.c：
int new_block(int dev)
{
      ...
      clear_block(bh->b_data);
      bh->b_uptodate = 1;
      bh->b_dirt = 1;
      return j;
}
```

这个返回的块号会存储在 hello.txt 文件的 i 节点表中的 inode->i_zone[block] 这一项中，由于 _bmap 函数下传的实参 block 为 0（实参是从 create_block 函数一直传递下来的，通过 4.3.4 节中的介绍我们得知，文件的当前操作指针 f_pos 为 0，到这里一直没有变化过，所以，pos/BLOCK_SIZE 为 0，block 也为 0），所以返回的块号会存储在 hello.txt 文件的 i 节点表中的 inode->i_zone[0] 这一项中。

这样，本节的第一步就完成了，实例 2 中的第二个要求（要确定把内容写入到文件的什么位置）就实现了。写入的时候，就把内容写入到最新申请的这个缓冲块中，将来缓冲块同步到硬盘的时候，就会根据这个缓冲块管理结构中记录的设备号和块号，准确无误地映射到 inode->i_zone[block] 这一项所记录的数据块上。

之后，_bmap 函数返回了，create_block 函数也返回了，继续回到 sys_write 函数去执行，准备实现实例 2 中的第三个要求，即把哪些内容写入到文件的指定位置处。

4.4.3 为数据的写入申请缓冲块

这里开始执行本节的第二步工作，即将第一步确定的位置所在的数据块读取到缓冲区中，这是为了完成实例 2 中的第三个要求，即“要确定把内容写入到文件的什么位置”所做的准备。

这一步是通过调用 bread 函数来实现的，该函数的执行与 4.2.3 节中的 bread 函数的执行步骤有所不同。由于本章 4.4.2 节中已经介绍到，该逻辑块已经被注册在缓冲区管理结构中了，所以此次无须读盘，直接在缓冲区中就可以得到，执行代码是：

```
// 代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    if (!(bh=bread(inode->i_dev,block)))
    ...
}
// 代码路径：fs/buffer.c：
struct buffer_head * bread(int dev,int block)
{
    ...
    if (!(bh=getblk(dev,block)))
    ...
}
// 代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
{
    ...
    if (bh = get_hash_table(dev,block))
        return bh;
    ...
}
```

这里产生了一个问题，本节是要介绍文件的写入，为什么要将第一步确定的位置所在的数据块读取到缓冲区中呢？其中原因还是要从实例 2 的场景说起：实例 2 是一个比较简单的文件写入场景，hello.txt 文件是一个新建文件，而且又是要从文件的起始位置开始写入数据的，因此在 4.2.2 节中调用 create_block 函数后，也就自然而然地会在缓冲区中申请一个缓冲块，将其与新的硬盘数据块相对应，为写入做准备，这就导致此次调用 bread 函数时，完全没必要再从硬盘上读取出来，缓冲区中有现成的。如图 4-32 所示：

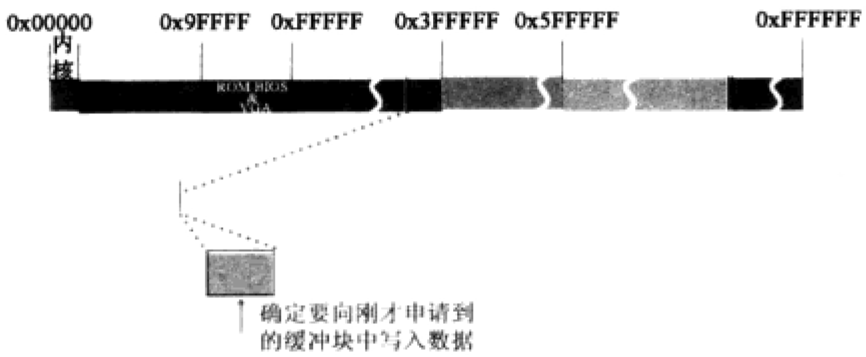


图 4-32　确定向刚才申请的缓冲块写入数据

但是，file_write 在设计时要考虑到通用性，如果情况稍微复杂一些，调用 bread 函数就很有可能需要从硬盘上读取数据块了。我们在 4.4.2 节中介绍 create_block 函数时，假设过一个场景，即“往一个已经存在的文件中写入数据，而且文件的当前操作指针 f_pos 并不是文件起始位置，而是处于某个已经存在的数据块上”，在这种情况下，就要先把这个数据块读入缓冲区，然后进一步计算要具体把数据写入到这个数据块中的哪个位置，并根据结果写入数据，以便将来同步时将该缓冲块的数据准确地对应到硬盘的相应数据块上。这种较为复杂的写盘情况，我们将在 4.5 节进一步介绍。

file_write 函数为了实现通用性，决定无论如何都调用 bread 函数去读取数据块，bread 函数内部会根据具体的情况进行处理。如果缓冲区中已经申请过能够对应硬盘上数据块的缓冲块了，直接将结果返回；如果确实需要从硬盘上读取数据块，再根据 4.4.2 节中 inode->i_zone[block] 这一项所记录的数据块块号把数据读出来。

这个缓冲块的确定就为完成实例 2 中的第三个要求（要确定把内容写入到文件的什么位置）创造了条件，下面该进行正式的写入工作了。

4.4.4　将指定的写入数据从用户数据区拷贝到缓冲块

这里开始执行本节的第三步工作，即将程序规定的数据写入到指定的缓冲块中。这是为了完成实例 2 中的第三个要求（要确定把内容写入到文件的什么位置）所做的具体执行工作，如图 4-33 所示。

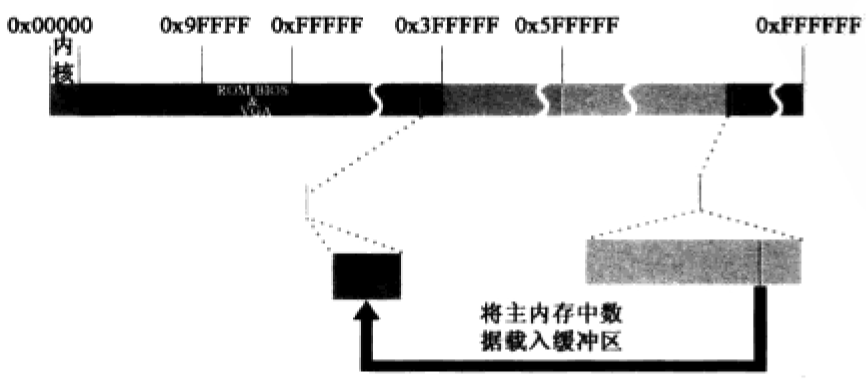


图 4-33　将数据从主内存区拷贝到高速缓冲块

执行代码如下：

```
// 代码路径：fs/file_dev.c：
```

```
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    while (i<count)
    {
        c = pos % BLOCK_SIZE;
        p = c + bh->b_data;
        bh->b_dirt = 1;
        c = BLOCK_SIZE-c;
        if (c > count-i) c = count-i;
        pos += c;
        if (pos > inode->i_size) {
            inode->i_size = pos;
            inode->i_dirt = 1;
        }
        i += c;
        while (c-->0)
            *(p++) = get_fs_byte(buf++);
        brelse(bh);
}
...
}
```

首先要根据文件的当前操作指针 pos 在这个逻辑块中测算出写入的精确位置，由于现在 pos 值为 0，所以此次写数据就要从这个数据块的起始部分开始写，执行代码如下：

```
//代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    while (i<count)
    {
        ...
        c = pos % BLOCK_SIZE;
        p = c + bh->b_data;
        ...
}
...
}
```

之后将该缓冲块的 b_dirt 标志位设置为 1，将来把数据往硬盘上同步的时候，只要见到 b_dirt 标志位是 1 的缓冲块，一律同步。执行代码如下：

```
//代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    while (i<count)
    {
        ...
        bh->b_dirt = 1;
        ...
}
...
}
```

接下来要统计还剩多少数据没有写入，并将 i 节点的属性进行调整，比如文件的大小等。由于上述操作改变了 hello.txt 文件的 i 节点，因此也需要将 i 节点的“脏标志”（即 inode->i_dirt）置 1。

执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    while (i<count)
    {
        ...
            c = BLOCK_SIZE-c;
            if (c > count-i) c = count-i;
            pos += c;
            if (pos > inode->i_size) {
                inode->i_size = pos;
                inode->i_dirt = 1;
            }
            i += c;
            ...
}
...
}
```

最后，将用户指定的数据写入到缓冲块中，并释放该缓冲块，执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    while (i<count)
    {
        ...
            while (c-->0)
                   *(p++) = get_fs_byte(buf++);
            brelse(bh);
}
...
}
```

这样，本次写文件的操作就完成了，实例 2 中指定的字符“Hello, world”被写入到了缓冲块上。从实例 2 的代码上看，一次写入操作就可以把指定的写入工作完成了。当然，如果情况更复杂一些，比如还有更多的数据需要写入，都将在这个 while (i<count) 循环中完成。

到此为止，实例 2 中的三个要求已经全部实现了，这意味着它所要达到的最终目的（往一个名为 hello.txt 的新建文件中写入内容）已经实现。

不过我们发现，需要写入的数据并没有写入到硬盘上，而是先写入到缓冲区中了，下面我们来具体介绍这些数据是如何从缓冲区同步到硬盘上的。

4.4.5 数据同步到硬盘的方法 1

在 Linux 0.11 中，将数据从缓冲区同步到硬盘上有两种方法，这里介绍将缓冲区中的数据同

步到硬盘上的一种方法。

在第 2 章第 2.8 节中我们介绍到，shell 进程在第一次执行时，启动了一个名叫 update 的进程，这个进程常驻于内存，这个进程的功能，就是将缓冲区中的数据同步到外设上（外设包括软盘、硬盘、虚拟盘等，我们这里仅以硬盘为例）。

该进程会调用 pause 函数，这个函数会映射到 sys_pause 函数中，使该进程被设置为可中断等待状态，每隔一段时间，系统就将 update 进程唤醒，并切换到该进程，由它来调用 sync 函数，并将缓冲区中的数据同步到设备上，然后 update 进程就挂起。

sync 函数执行后，会产生软中断，并最终映射到 sys_sync 函数去执行，同步的工作就是 sys_sync 这个函数来实现的，对应的代码如下：

```
// 代码路径：fs/buffer.c：
int sys_sync(void)
{
        int i;
        struct buffer_head * bh;

        sync_inodes();          /* write out inodes into buffers */
        bh = start_buffer;
        for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
            wait_on_buffer(bh);
            if (bh->b_dirt)
                ll_rw_block(WRITE,bh);
        }
        return 0;
}
```

很显然，同步工作分为两步走，第一步，是将需要同步的 i 节点载入缓冲区中，这一步是通过调用 sync_inodes 函数来实现的；第二步，再将缓冲区中需要同步的数据，同步到硬盘上，这时候缓冲区中就包括了逻辑块位图的数据、i 节点位图的数据、文件 i 节点的数据，以及文件本身的内容。这些数据，在文件操作的过程中，都有可能发生变化，这必将导致文件内容及其管理信息的改变，把这些数据同步，就是要让用户对文件的操作，在外设上得以落实，接下来，我们先来看 i 节点，是如何从 i 节点表 inode_table[32] 载入到缓冲区中的。

进入 sync_inodes 函数后，开始遍历 i 节点表，只要发现哪个 i 节点被改动过，即，i_dirt 为 1，就说明该 i 节点需要同步，于是调用 write_inode 函数，开始将该 i 节点载入缓冲区，执行代码如下：

```
// 代码路径：fs/inode.c：
void sync_inodes(void)
{
        ......
        for(i=0 ; i<NR_INODE ; i++,inode++) {
            wait_on_inode(inode);
            if (inode->i_dirt && !inode->i_pipe)
                write_inode(inode);
    }
        ......
}
```

进入 write_inode 函数后，先测算出该i节点在硬盘上逻辑块的位置，之后，就将该i节点所在的逻辑块读出来，执行代码如下：

```
// 代码路径：fs/inode.c:
static void write_inode(struct m_inode * inode)
{
        ......
        block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
            (inode->i_num-1)/INODES_PER_BLOCK;
        if (!(bh=bread(inode->i_dev,block)))
        ......
}
```

之后，将该i节点写入这个逻辑块对应的缓冲块中，并将该缓冲块设置为脏，以此表示该缓冲块是需要同步到硬盘的，这个缓冲块一同步，i节点也就被写入到硬盘了，最后，再将该i节点的“脏”标志设置为0，以表示这个i节点已经同步过了，不需要同步了，执行代码如下：

```
// 代码路径：fs/inode.c:
static void write_inode(struct m_inode * inode)
{
        ......
        ((struct d_inode *)bh->b_data)
            [(inode->i_num-1)%INODES_PER_BLOCK] =
                *(struct d_inode *)inode;
        bh->b_dirt=1;
        inode->i_dirt=0;
        ......
}
```

sync_inodes 函数将会循环调用 write_inode 函数，将所有需要同步的i节点全都载入缓冲区中。

现在要进行同步工作的第二步，要将缓冲区中所有需要同步的缓冲块同步到硬盘中。

由 ll_rw_block 函数来完成具体的写盘任务。这个函数的操作过程，我们在第2章第2.3节中已经详细介绍了，唯一的不同是，当时执行的是读操作，此次执行的是写操作，但技术路线大体是一致的，ll_rw_block 函数将被反复调用，实例2中操作的文件，其i节点是新建的，文件内容是新写的，在i节点位图和逻辑块位图中，与此相关的管理信息，都发生了变化，这些变化了的数据信息，都在缓冲块中，它们都将通过 ll_rw_block 函数，被写入到硬盘上，这就是文件在缓冲区中的内容被同步到硬盘上的第一种情况。

4.4.6 将文件写入硬盘的情况 2

这里介绍将缓冲区数据同步到硬盘上的另一种情况。我们在4.4.5节中已经介绍到，update 进程每隔一段时间，会将缓冲区中的数据同步到硬盘上，之后，它将被挂起。那么如果在它被挂起的这段时间内，写盘进程不断地将数据写入到缓冲区中，导致缓冲区被写满了，而且还并不满足写盘进程的需求，它还要写入数据，这时候，就需要强行先将缓冲区中的数据同步到硬盘上，以便腾出空间，继续为写盘进程的数据写入提供服务。

由于实例 2 的场景比较简单，所以它写入缓冲区的数据比较少，还没有遇到这种情况，但如果场景稍微复杂一些，就有可能遇到这种情况了，我们不妨对实例 2 稍加改动，来看下面的源代码：

```
void main()
{
char str1[]="Hello, world";
int i;

//新建文件
int fd = creat("/mnt/user/hello.txt",0644));
//写文件
    for(i=0;i<1000000;i++)
    {
        int size = write(fd,str1,strlen(str1));
    }
}
```

从这个改动的源代码可以看出，要写入的数据将达到 10MB 以上，而 Linux 0.11 中的缓冲区，肯定不可能超过 10MB，所以，当前进程如果要写入数据的话，很可能在 update 进程被唤醒之前，就已经将缓冲区写满了，而且它还要继续写入数据，这样的话，就需要强行将缓冲区中的数据同步到硬盘，以便为后续的写入工作提供空间了。

通过本节前面对 write 的介绍得知，该进程要在缓冲区中不断地申请空闲的缓冲块，当缓冲区中所有空闲的缓冲块都被写入数据，即，b_dirt 都为 1 后，仍然需要继续调用 getblk 函数来申请缓冲块，进入 getblk 函数后，会执行到如下代码：

```
//代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
{
    ......
    tmp = free_list;
    do {
        if (tmp->b_count)
            continue;
        if (!bh || BADNESS(tmp)<BADNESS(bh)) {
            bh = tmp;
            if (!BADNESS(tmp))
                break;
        }
/* and repeat until we find something good */
    } while ((tmp = tmp->b_next_free) != free_list);
    ......
}
```

这段代码的意思是：要在缓冲区中找到一个空闲的缓冲块。tmp->b_count 这个条件很容易满足，这是因为，虽然所有空闲的缓冲块都被设置为“脏”了，但它们仍然空闲着，即 b_count 仍然为 0，在当前的情况下，一定会找到一个“脏”的“空闲”缓冲块。

接下来就要执行以下代码：

```
//代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
```

```
{
    ......
    while (bh->b_dirt) {
        sync_dev(bh->b_dev);
    }
    ......
}
```

注意 sync_dev 这个函数，这个函数就是此时强行同步缓冲块到硬盘的函数，进入该函数后，执行代码如下：

```
//代码路径：fs/buffer.c：
int sync_dev(int dev)
 {
    int i;
    struct buffer_head * bh;

    bh = start_buffer;
    for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
        if (bh->b_dev != dev)
            continue;
        wait_on_buffer(bh);
        if (bh->b_dev == dev && bh->b_dirt)
            ll_rw_block(WRITE,bh);
    }
    sync_inodes();
    bh = start_buffer;
    for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
        if (bh->b_dev != dev)
            continue;
        wait_on_buffer(bh);
        if (bh->b_dev == dev && bh->b_dirt)
            ll_rw_block(WRITE,bh);
    }
    return 0;
}
```

很显然，这个函数中的代码和 4.4.5 节中介绍的 sys_sync 的代码很类似，也包括了将 i 节点同步到缓冲区，以及将缓冲区中需要同步的缓冲块同步到硬盘，具体的执行过程，我们在 4.4.5 节中已经说明了。

注意看 for (i=0 ; i<NR_BUFFERS ; i++,bh++) 这个循环，这表明，一旦进入这个函数，实例 2 的写入操作将会被暂时搁置，转而“一心一意”地将缓冲区中所有需要同步的缓冲块全部同步到硬盘，然后，再返回到 getblk 函数中，继续完成实例 2 的写盘工作。

这就是缓冲区中数据同步到硬盘的另一种情况，在 Linux 0.11 中，就是通过 4.4.5 节和 4.4.6 节中介绍的这两种方法，将缓冲区中数据同步到硬盘上的。

本节的内容已经介绍完了，最后还有一个问题值得注意。

我们来看 4.4.4 节中最后一段代码：

```
//代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
```

```
{
    ...
    while (i<count)
    {
        ...
            while (c-->0)
                *(p++) = get_fs_byte(buf++);
            brelse(bh);
    }
    ...
}
```

指针 p 很显然是从指定数据块的起始位置开始写入数据的。这个数据块之前是没有关于 hello.txt 的数据的，所以谈不上覆盖数据块中的后续数据。但是，实例 2 毕竟是一个比较简单和比较基本的案例，如果 hello.txt 不是一个新建文件，而是一个已有文件，那么情况就全变了，指针 p 无论往哪个数据块写入数据都会把这个数据块中的“写入点”后面已有的数据覆盖掉，除非是在尾端加写。

这样就导致了一个问题，用户只能在文件尾端“加写”数据，如果想在文件中间“改写”数据，单纯依靠 sys_write 函数就无能为力了，那么面对“改写”数据这种更为复杂的文件写入情况，系统又该如何处理呢？下一节的内容将告诉我们答案。

4.5 修改文件

本节就解答 4.4 节中最后提出的疑问，即如何修改文件。修改文件的本质就是可以在文件的任意位置插入数据和删除数据，而且不会影响到文件已有的数据，本节我们就以插入数据为例来对修改文件进行具体的讲解。

相对实例 2 来讲，这是一个较为复杂的案例。我们从 4.4 节中对 sys_write 函数的介绍中可以看出，这个函数是不可能独自完成此项任务的，它一旦从文件的当前操作指针 f_pos 处开始写入数据，就肯定会把后面的数据覆盖掉。面对这种情况，Linux 0.11 给出的解决方案是：将 sys_read、sys_write 以及 sys_lseek 几个函数组合使用，从而解决“修改文件”问题。sys_read 和 sys_write 函数的执行，我们在 4.2 节和 4.4 节中已经介绍过了。在本节中，我们先来介绍 sys_lseek 函数的功能，然后再具体介绍当面对较为复杂的文件写入情况时应该如何处理。

本节的程序架构图如图 4-34 所示：

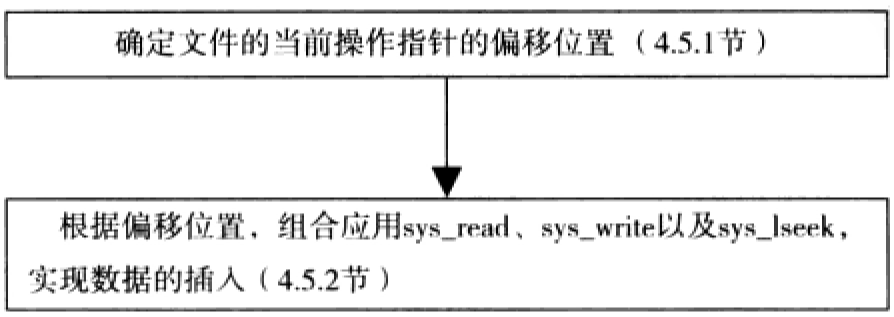


图 4-34 文件修改操作的程序架构图

4.5.1 对文件的当前操作指针进行重定位

这里我们先来介绍 sys_lseek 的执行情况。这个系统调用的功能是：对文件的当前操作指针 f_pos 进行重新定位。lseek 这个库函数被调用后，产生软中断，之后就映射到 sys_lseek 函数中去执行，进入该函数后，代码如下：

```
//代码路径：fs/read_write.c：
int sys_lseek(unsigned int fd,off_t offset, int origin)
{
    ...
    switch (origin) {
        case 0:
            if (offset<0) return -EINVAL;
            file->f_pos=offset;
            break;
        case 1:
            if (file->f_pos+offset<0) return -EINVAL;
            file->f_pos += offset;
            break;
        case 2:
            if ((tmp=file->f_inode->i_size+offset) < 0)
                return -EINVAL;
            file->f_pos = tmp;
            break;
        default:
            return -EINVAL;
    }
    return file->f_pos;
}
```

通过参数 offset 可以了解到用户希望将文件的当前操作指针 f_pos 的位置向文件尾端移动多少个字节；实参 SEEK_SET 是一个宏定义，sys_lseek 函数只能对三种宏进行有效操作，具体介绍如下：

- SEEK_SET 代表 0，表明从文件起始处开始偏移；
- SEEK_CUR 代表 1，表明从文件的当前读写位置处开始偏移；
- SEEK_END 代表 2，表明从文件的尾端开始偏移。

4.5.2 对文件进行修改

现在，假设 hello.txt 是硬盘上已有的一个文件，而且内容为“hello，world”，在文件的当前操作指针设置完毕后，我们来介绍 sys_read、sys_write 和 sys_lseek 如何联合使用才能把数据插入到 hello.txt 文件中。

可以通过如下方式对它们进行组合应用，应用程序的代码如下：

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

#define LOCATION 6
```

```
int main(char argc, char **argv)
    {
        char str1[] = "Linux";
        char str2[1024];
        int fd, size;

        memset(str2, 0, sizeof(str2));
        fd = open("hello.txt", O_RDWR, 0644);
        lseek(fd, LOCATION, SEEK_SET);
        strcpy(str2, str1);
        size = read(fd, str2+5, 6);

        lseek(fd, LOCATION, SEEK_SET);
        size = write(fd, str2, strlen(str2));

       close(fd);
       return 0;
   }
```

这是一段用户进程的程序，通过这样一段代码就能将“Linux”这个字符串插入到 hello.txt 文件中了，最终 hello.txt 文件中的内容应该是："hello, Linuxworld”。

这段代码几乎把本章前面的几节讲到的所有系统调用函数都用到了，下面我们来分析一下这些代码的作用。

- fd = open("hello.txt", O_RDWR, 0644);

  open 函数将对应 sys_open 函数，很明显，在操作文件之前先要确定要操作哪个文件。

- lseek(fd, LOCATION, SEEK_SET);

  lseek 函数将对应 sys_lseek 函数，由于参数中选择了 SEEK_SET，表明要将文件的当前操作指针从文件的起始位置向文件尾端偏移 6 个字节。

- strcpy(str2, str1);

  这一行是将“Linux”这个字符串拷贝到 str2[1024] 这个数组的起始位置处。

- size = read(fd, str2+5, 6);

  read 函数将对应 sys_read 函数，现在要读取 hello.txt 这个文件的内容了，实参“str2+5”表示要把从 hello.txt 这个文件读出来的内容拷贝到 str2 这个数组的第 6 个元素之后，相当于是与之前的 Linux 这个字符串进行了拼接；实参“6”表明要将该文件的 6 个字符读出，由于 lseek(fd, LOCATION, SEEK_SET); 这一行已经将文件的当前操作指针从文件的起始位置向文件尾端偏移了 6 个字节，所以此次读出的内容就应该是 world，最后拼接的结果是：Linuxworld。

- lseek(fd, LOCATION, SEEK_SET);

  这行的效果和前面调用的效果一样，都是要将文件的当前操作指针，即文件的起始位置，向文件尾端偏移 6 个字节，这时候就确定了下面文件的准确写入位置；

- size = write(fd, str2, strlen(str2));

  write 函数将对应 sys_ write 函数，现在要将 str2 这个数组中的“Linuxworld”字符串写入到 hello.txt 文件中，而且写入位置刚刚确定，就是从文件的起始点向尾端偏移 6 个字节的位置，于

是最终的写入结果就是："hello, Linuxworld"。

这就是将 sys_read、sys_write 和 sys_lseek 组合应用，从而实现文件修改的全过程。

由此可见，对于"插入文件"这种较为复杂的操作的写入操作，Linux 0.11 采取的策略是：通过调整文件的当前操作指针，将插入点后续的数据先读出来，然后拼接，最后再写入，使新写入的数据自然而然地覆盖插入点原来的后续内容。这种组合操作的策略可以解决 Linux 0.11 中所有的文件写入问题，包括改写等都可以采取类似的策略。

到此为止，实例 1 和实例 2 的实现过程就全部完成了，下面将讲解实例 3 的实现过程。

实例 3：关闭此文件，之后将其从文件系统中删除。

此处，我们对 4.5.2 节的实例程序进行改写，代码如下所示：

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

    #define LOCATION 6

int main(char argc, char **argv)
    {
        char str1[] = "Linux";
        char str2[1024];
        int fd, size;

        memset(str2, 0, sizeof(str2));
        fd = open("/mnt/user/hello.txt", O_RDWR, 0644);
        lseek(fd, LOCATION, SEEK_SET);
        strcpy(str2, str1);
        size = read(fd, str2+5, 6);

        lseek(fd, LOCATION, SEEK_SET);
        size = write(fd, str2, strlen(str2));

// 关闭文件
  close(fd);
// 删除文件
  unlink("/mnt/user/hello.txt");
       return 0;
   }
```

我们通过实例 3 来说明文件的两类"解除关系"：

- 一类是通过调用 close 函数来关闭一个当前进程打开着的文件，这是为了解除该文件与当前进程的关系，关系解除后，当前进程将不再具备操作文件的能力。
- 另一类是通过调用 unlink 函数来删除文件，这是为了解除该文件与整个文件系统的关系，关系解除后，该文件在文件系统中的全部管理结构将不复存在，它将永远不能被找到。

4.6 关闭文件

在前面的几节中，当前进程已经对 hello.txt 文件进行了多种操作，本节将要介绍如何关闭该

文件，即执行实例 3 中的 close(fd); 这行代码。关闭文件的本质是进程和文件切断一切关系。

具体分为三个步骤：

第一步：将当前进程管理结构 task_struct 中的文件管理指针表 *filp[20] 与系统中的文件管理表 file_table[64] 解除关系；

第二步：将系统中的文件管理表 file_table[64] 中的 hello.txt 文件的引用次数减 1；

第三步：释放 hello.txt 文件的 i 节点。

下面我们将具体介绍这三个步骤是如何实现的。

既然新建和打开操作是将用户进程与内核中的管理文件的数据结构相挂钩，那么关闭操作自然就是把它们脱钩了。脱钩之后，用户进程就找不到该文件的相关信息了，因此在用户进程看来，文件关闭了。但是，文件本身仍然是存在的，无论是其他进程，还是当前这个进程，都可以在文件关闭之后，再次执行打开操作来建立与这个文件的联系。

因此从这个角度说，关闭文件有两层含义：第一层是针对当前进程的，通过释放文件描述符指定的指针，使得当前进程与内核文件表中存储的文件信息脱钩；第二层是针对内核而言的，首先释放当前进程对文件的引用，然后，如果没有其他进程引用该文件，就释放文件对应的 i 节点。

本节内容的程序架构图如图 4-35 所示：

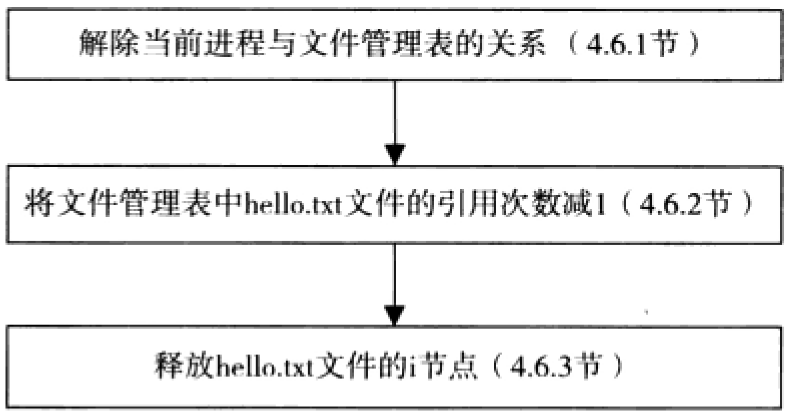


图 4-35　文件关闭操作的程序架构图

4.6.1　当前进程与文件管理表“脱钩”

close 这个库函数开始执行后会产生软中断，然后映射到 sys_close 函数中去执行。映射的过程与 4.1 节中 open 函数映射到 sys_open 函数的执行过程大体类似。进入 sys_close 函数后，开始执行关闭 hello.txt 文件的第一步，即将当前进程的管理结构 task_struct 中的文件管理指针表 *filp[20] 与系统中的文件管理表 file_table[64] 解除关系，如图 4-36 所示：

执行代码如下：

```
// 代码路径：fs/open.c：
int sys_close(unsigned int fd)
{
    ...
```

```
    current->filp[fd] = NULL;
    ...
}
```

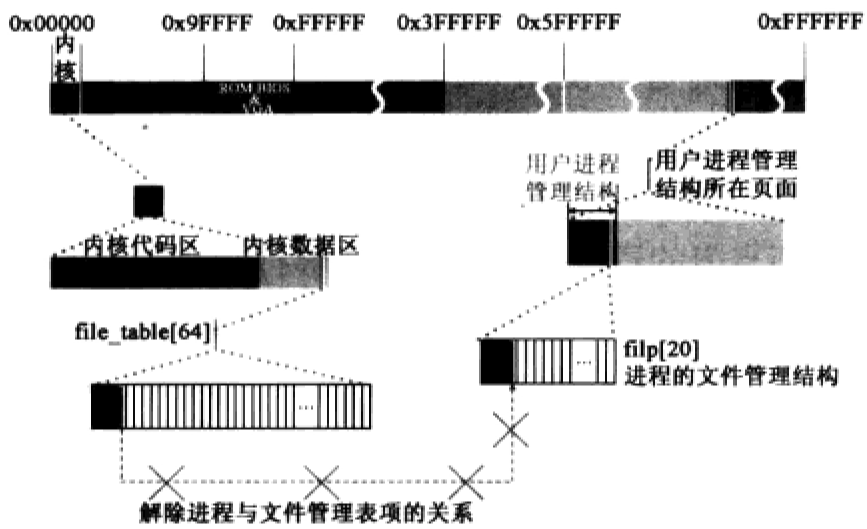


图 4-36 断开当前进程与 hello.txt 文件的关联

4.6.2 将文件管理表中 hello.txt 对应的引用次数减 1

这里介绍关闭 hello.txt 文件的第二步，即将系统中的文件管理表 file_table[64] 中 hello.txt 文件的引用次数减 1，如图 4-37 所示：

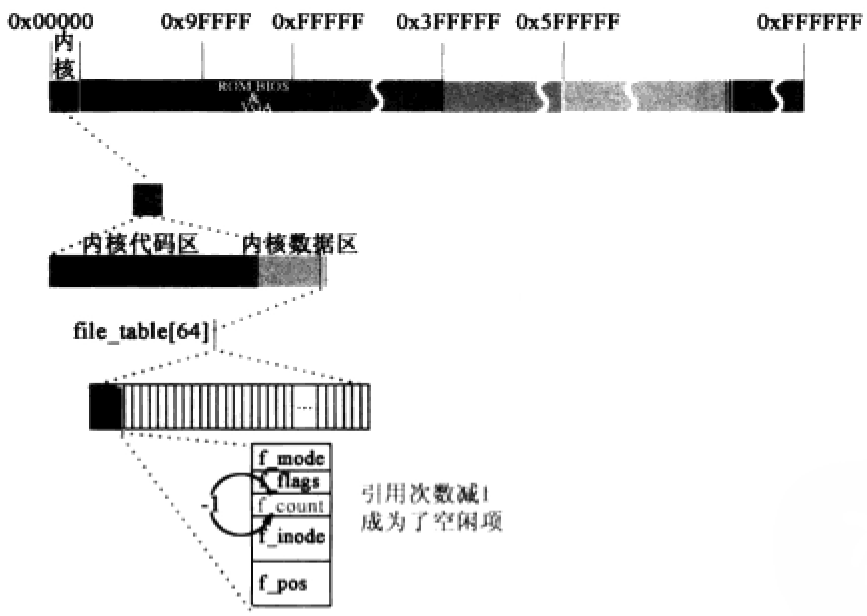


图 4-37 将 hello.txt 文件的引用计数减 1

执行代码如下：

```
// 代码路径：fs/open.c：
int sys_close(unsigned int fd)
{
    ...
    if (--filp->f_count)
    ...
}
```

之所以只是把引用计数减 1 而不是把这个表项清空，是因为 Linux 0.11 是一个支持多进程的

操作系统，其他进程也有可能正在操作 hello.txt 文件，而且用的就是这个表项（这种情况我们将在第 6 章详细介绍）。所以，对于多进程很有可能共用的资源，只能用引用计数来表示其使用状况，而不能简单地“置位”或“清空”。

由于本实例中假定只有当前进程使用 hello.txt 文件，因此 hello.txt 文件对应的文件槽的 f_count 字段就变成了 0。这标志着，已经没有进程使用 hello.txt 文件了。对于操作系统来说，f_count 值为 0 具有重要意义，意味着操作系统将把 hello.txt 文件对应的 i 节点从内核 i 节点管理表中删除，以回收宝贵的内核 i 节点表的资源，使得内核能够打开另一个进程请求进行操作的文件。

从 4.6.3 节开始，我们将展示这一过程，即关闭 hello.txt 文件的第三步。

4.6.3 hello.txt 文件与文件管理表“脱钩”

这里介绍关闭 hello.txt 文件的第三步，即释放 hello.txt 文件的 i 节点，执行代码如下：

```
//代码路径：fs/open.c：
 int sys_close(unsigned int fd)
{
     ...
     iput(filp->f_inode);
     ...
}
//代码路径：fs/inode.c：
void iput(struct m_inode * inode)
{
    ...
repeat:
     if (inode->i_count>1) {
             inode->i_count--;
             return;
    }
    if (!inode->i_nlinks) {
             truncate(inode);
             free_inode(inode);
             return;
    }
    if (inode->i_dirt) {
             write_inode(inode);   /* we can sleep - so do again */
             wait_on_inode(inode);
             goto repeat;
    }
    inode->i_count--;
    return;
}
```

由于假定只有当前进程打开 hello.txt 文件，inode->i_count 是 1，因此进入 iput 函数后，执行如下代码解除 hello.txt 文件的 i 节点和系统 i 节点表 inode_table[32] 的关系：

```
//代码路径：fs/inode.c：
void iput(struct m_inode * inode)
{
     ...
```

```
        if (inode->i_dirt) {
                write_inode(inode); /* we can sleep - so do again */
                ...
        }
        ...
}
```

由于本实例的应用程序对 hello.txt 文件进行了改写，导致 hello.txt 文件对应的 i 节点的脏标志置位，理由在 4.4.4 节中已经做了介绍。所以该 i 节点的信息需要和硬盘上的数据信息进行同步，即写盘，write_inode 函数将完成此功能。当然，我们在 4.4 节中已经讲过，Linux 0.11 中任何形式的写盘，都只是把数据写入到缓冲块中，然后再往设备上同步，hello.txt 的 i 节点的数据也不例外，所以 write_inode 函数会将其写入到缓冲区中。

至于 write_inode 的具体执行过程，与 read_inode 的执行大同小异，不难理解。i 节点回写到高速缓冲区的效果如图 4-38 所示：

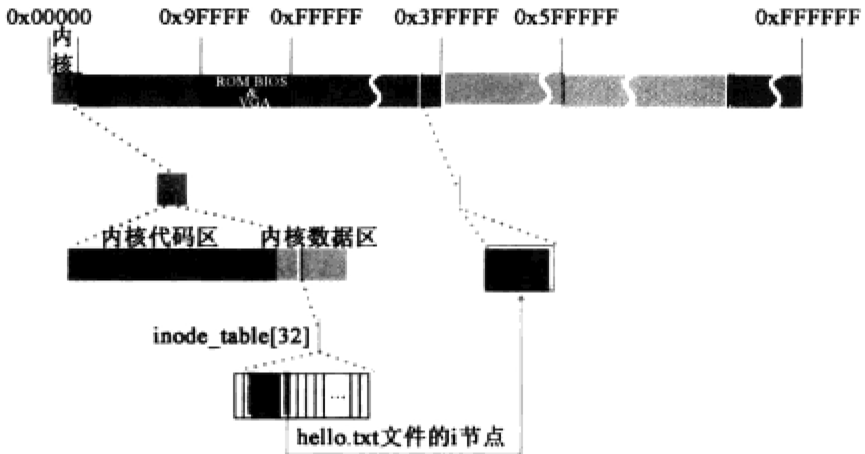


图 4-38　i 节点数据回写到硬盘

write_inode 函数返回后，将 hello.txt 对应的 i 节点的引用次数减 1，执行代码如下：

```
// 代码路径：fs/inode.c：
void iput(struct m_inode * inode)
{
     ...
     inode->i_count--;
     ...
}
```

减 1 的理由和 4.6.2 节中的理由是一样的，hello.txt 文件很可能被其他进程所引用，它是公共资源，所以只能减 1，而不能删除。当然，现在的情况是，该 i 节点仅仅被当前进程引用着，所以此次减 1，inode->i_count 就变成 0 了，系统将认定它为“空闲 i 节点”。该过程如图 4-39 所示。

我们现在回顾一下 4.1 节中打开文件部分的内容，不难发现，其实关闭文件和打开文件操作的技术路线是相反的。打开文件就是以系统中文件管理表 file_table[64] 为中转站，让当前进程管理结构中的 *filp[20] 和 hello.txt 文件的 i 节点分别与该中转站相挂接，从而使当前进程具备操作 hello.txt 文件的能力。关闭文件要解除的关系也恰恰就是这两个方面，分别是让当前进程管理结构中的 *filp[20] 和 hello.txt 文件的 i 节点与这个中转站解除关系，从而使当前进程与 hello.txt 彻

底解除关系。

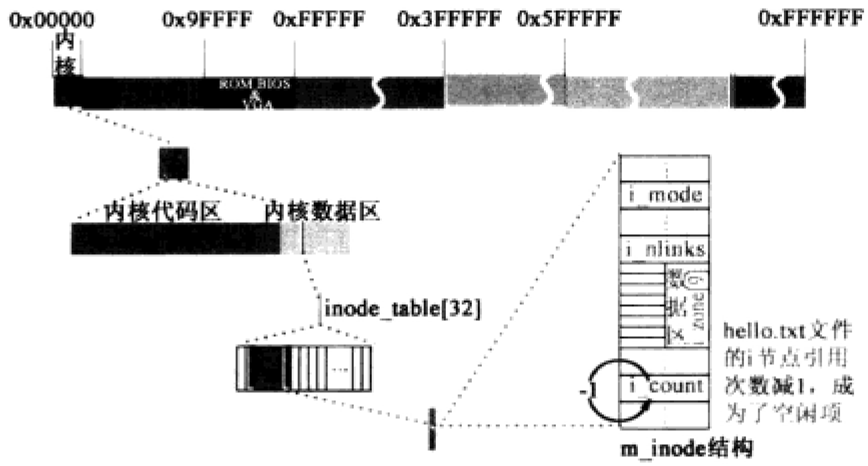


图 4-39　完成文件关闭过程

4.7　删除文件

本节将介绍删除 hello.txt 文件的具体步骤，删除文件与 4.6 节中的关闭文件有所不同，关闭文件只是解除当前进程与 hello.txt 文件的关系，而删除操作虽然也是由当前进程发起的，但是它不仅仅是断开当前进程与这个文件的关联，同时也会导致这个文件“彻底”被删除。删除操作完成以后，无论是当前进程，还是其他进程，都无法再通过路径名访问到与这个路径名对应的文件了。本实例中，删除“/mnt/user/hello.txt”路径名指定的“hello.txt”文件的工作是通过执行实例 3 中的 unlink("/mnt/user/hello.txt"); 这行代码实现的。

具体的删除过程分为两步：

第一步：全方位地检测 hello.txt 文件是否具备删除条件；

第二步，得知可以删除后，对该文件的管理结构进行设置，使之永远无法通过“路径名”找到。下面我们就来介绍具体的执行步骤。

本节的程序架构如图 4-40 所示：

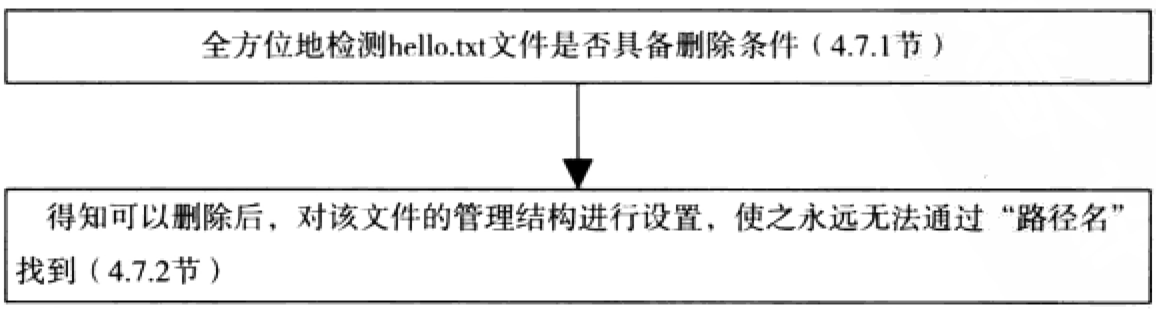


图 4-40　文件删除操作的程序架构图

4.7.1　系统准备删除 hello.txt 文件

unlink 这个库函数开始执行后，会产生软中断，然后映射到 sys_unlink 函数中去执行。映射的过程与本章第 1 节中 open 映射到 sys_open 函数的执行过程大体类似。进入 sys_ unlink 函数后，开始执行删除 hello.txt 文件的第一步，即全方位地检测 hello.txt 文件是否具备删除条件。

现在系统得到的仅仅是路径名“/mnt/user/hello.txt”，所以先要通过这个路径名把 hello.txt 文件的管理信息读出来，才能针对该文件进行后续的操作，于是先要获取枝梢 i 节点，即 user 目录文件的 i 节点，具体操作步骤我们在 4.1.3 节中已经讲过，执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    if (!(dir = dir_namei(name,&namelen,&basename)))
    ...
}
```

获取 user 目录文件的 i 节点后，接下来就要检测是否具备往这个目录文件中写入数据的权限，因为 hello.txt 这个目录项就在这个目录文件里面，如果没有写入权限，也就意味着无法把 hello.txt 的目录项从 user 目录中删除了，后续的工作都无从谈起，执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    if (!permission(dir,MAY_WRITE))
    ...
}
```

得知可以操作 user 这个目录文件后，就从这个目录文件中把 hello.txt 这个目录项读出来，并最终提取到 hello.txt 文件的 i 节点，执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    bh = find_entry(&dir,basename,namelen,&de);
    ...
    if (!(inode = iget(dir->i_dev, de->inode)))
    ...
}
```

接下来将针对 user 目录文件的 i 节点和 hello.txt 文件的 i 节点进行检测，先要检测用户是否有删除该文件的权限，执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    if ((dir->i_mode & S_ISVTX) && !suser() &&
            current->euid != inode->i_uid &&
            current->euid != dir->i_uid)
    ...
}
```

之后判断，如果 hello.txt 文件是目录文件，则不可以删除。这是因为，一个目录文件里面会有其他目录项，如果目录文件删除了，其他目录项也就找不到了，这些目录项所对应的文件，将会永远消失。当然，Linux 0.11 是允许删除目录的，内核会有其他的函数专门处理这个问题，但在这里不可以删除，执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    if (S_ISDIR(inode->i_mode))
    ...
}
```

然后判断 hello.txt 文件的 i 节点的链接数是否为 0，如果为 0，就发出警告，并设置为 1。这是因为，在整个系统中，至少当前进程还是在操作该 i 节点，所以它的链接数至少是 1，肯定不能是 0，所以要进行这样的调整，执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    if (!inode->i_nlinks)
    ...
}
```

到此为止，本节的第一步工作（全方位的检测 hello.txt 文件是否具备删除条件）就做完了。hello.txt 文件通过了全部检测，下面就开始执行具体的删除工作了。

4.7.2 删除 hello.txt 文件在硬盘上对应的数据和 i 节点

这里开始介绍本节的第二步工作，即得知可以删除后对该文件的管理结构进行设置，使之永远无法通过“路径名”找到。

“通过路径名无法找到”有两层含义：第一，通过删除路径名指定的目录项，断开从指定路径名到所要删除的文件对应的 i 节点的关联，所有的删除操作都要执行这个操作，在本实例中即是断开“/mnt/user/hello.txt”的路径名与 hello.txt 文件对应的 i 节点的关联；第二，如果已经没有其他路径名指向这个 i 节点，就把 i 节点及其对应的文件实体从硬盘上删除，对本实例来说，就是把 hello.txt 文件的 i 节点及其管理的硬盘上的文件删除。

删除工作开始后，先要将 hello.txt 目录项中对应的 i 节点号置 0，我们在 4.3.3 节中介绍过，只要目录项中的 i 节点号置 0，系统就认定该目录项为空闲项，即为空，这样 hello.txt 的目录项就失效了；然后再将这个目录项所在的缓冲块的 b_dirt 标志置 1，在 4.4.4 节中也介绍过，只要 b_dirt 的标志置 1，就表示这个缓冲块将来要同步到硬盘上。这就意味着，hello.txt 这个目录项失效的信息也会传达到硬盘上，从而真正让删除操作在硬盘上落实，如图 4-41 所示：

执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    de->inode = 0;
        bh->b_dirt = 1;
    ...
}
```

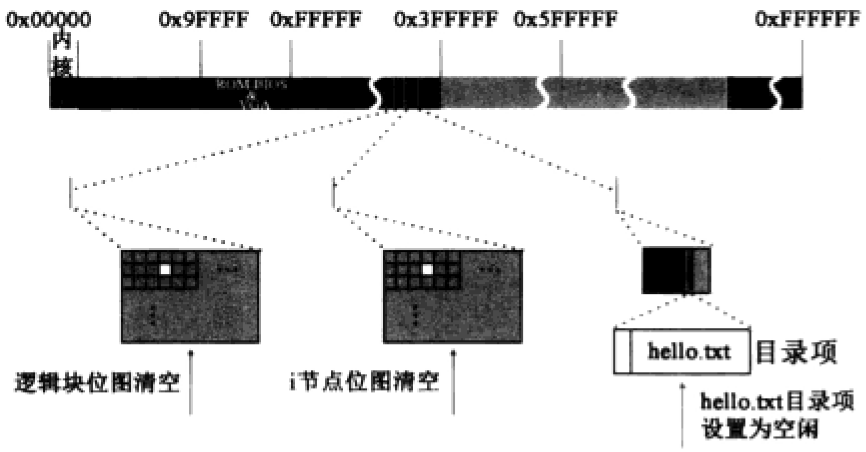


图 4-41　删除 hello.txt 文件

由于当前进程删除了 hello.txt 文件，因此 hello.txt 文件对应的 i 节点的链接数 inode->i_nlinks 应该减少 1，从而完成“通过路径名无法找到”的第一层含义；之后，再对 hello.txt 文件的 i 节点进行一些设置，包括将该 i 节点的 b_dirt 标志位置 1，表示该 i 节点已经改动过，需要同步到硬盘上了；最后，再将当前时间认定为该 i 节点的改动时间，执行代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    inode->i_nlinks--;
    inode->i_dirt = 1;
    inode->i_ctime = CURRENT_TIME;
    ...
}
```

可见，在 Linux 0.11 中，删除文件并不一定意味着将文件实体从硬盘上删除，只有当该文件对应的 i 节点的引用数 i_nlinks 为 0，表示已经没有其他路径名引用该文件时，才会将 hello.txt 文件的其他管理信息清零。

在本实例中，假定没有其他用户执行 sys_link 的操作，hello.txt 文件对应的 i 节点的 inode->i_nlinks 为 0，因此 hello.txt 文件将从硬盘上真正地被删除。

因此，接下来就要把 hello.txt 文件的 i 节点，以及文件在硬盘中的数据全部清除。这些工作都要通过调用 iput 函数来实现，效果如图 4-42 所示：

执行代码是：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
```

```
    ...
    iput(inode);
    ...
}
```

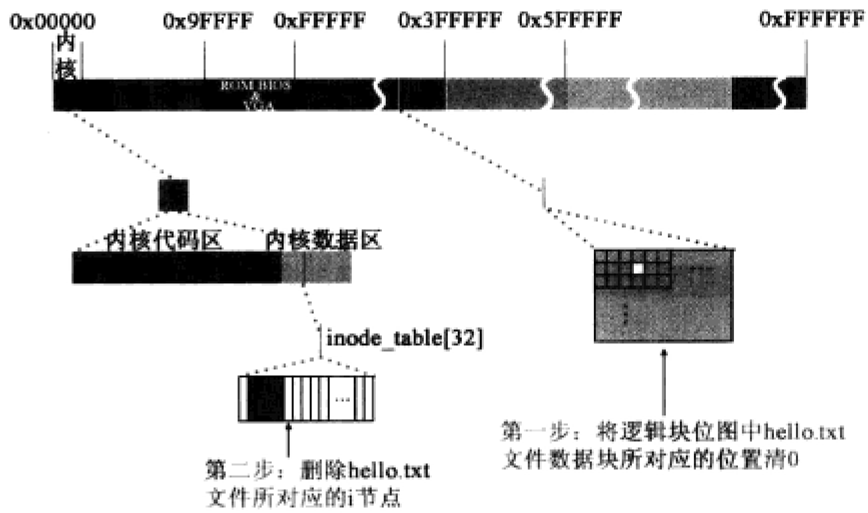


图 4-42　删除 hello.txt 文件对应的数据块和 i 节点

由于 hello.txt 文件的 i 节点的 i_nlinks 已经被削减为 0，所以进入该函数后，将会执行以下代码：

```
// 代码路径：fs/inode.c：
void iput(struct m_inode * inode)
{
    ...
    if (!inode->i_nlinks){
           truncate(inode);
           free_inode(inode);
           return;
     }
    ...
}
```

删除的顺序必须是先删除 hello.txt 文件在硬盘中对应的数据块，然后才能删除 hello.txt 文件对应的 i 节点，因为 hello.txt 文件究竟存储在硬盘的哪些块上，需要通过 i 节点才能找到。因此，如果先删除 i 节点，数据块就找不到了，也就无法正常删除 hello.txt 文件对应的数据块了。

执行从硬盘上删除 hello.txt 文件对应的数据块的代码是 truncate 函数。要删除某个数据块的内容也需要设备号和块号，这两个参数都可以从 hello.txt 的 i 节点中得到。由于在写入过程中只写入了小于 1KB 的数据，因此 hello.txt 对应的 i 节点只用到了一个直接块。i_zone[0] 字段所存储的就是硬盘上存储 hello.txt 文件的数据块的块号。对应代码如下：

```
// 代码路径：fs/truncate.c：
void truncate(struct m_inode * inode)
{
    int i;

    if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode)))
        return;
```

```
    for (i=0;i<7;i++)
        if (inode->i_zone[i]) {
              free_block(inode->i_dev,inode->i_zone[i]);
              inode->i_zone[i]=0;
        }
    ...
    inode->i_size = 0;
    inode->i_dirt = 1;
    inode->i_mtime = inode->i_ctime = CURRENT_TIME;
}
```

在得到设备号和块号之后，接下来就能将数据块从硬盘上删除了。在 Linux 0.11 中，从硬盘上删除文件并不是真正将存储文件的数据块全部清 0，而只是在位于高速缓冲区的逻辑块位图上将存储该文件的数据块标识为未使用即可。执行该操作的是 free_block 函数，代码如下：

```
//代码路径：fs/bitmap.c：
void free_block(int dev, int block)
{
    struct super_block * sb;
    struct buffer_head * bh;

    if (!(sb = get_super(dev)))
        panic("trying to free block on nonexistent device");
    if (block < sb->s_firstdatazone || block >= sb->s_nzones)
        panic("trying to free block not in datazone");
    bh = get_hash_table(dev,block);
    if (bh) {
        if (bh->b_count != 1) {
            printk("trying to free block (%04x:%d), count=%d\n",
                dev,block,bh->b_count);
            return;
        }
        bh->b_dirt=0;
        bh->b_uptodate=0;
        brelse(bh);
    }
    block -= sb->s_firstdatazone - 1 ;
    if (clear_bit(block&8191,sb->s_zmap[block/8192]->b_data)) {
        printk("block (%04x:%d) ",dev,block+sb->s_firstdatazone-1);
        panic("free_block: bit already cleared");
    }
   sb->s_zmap[block/8192]->b_dirt = 1;
}
```

由于超级块是对硬盘上的数据进行操作的起点，因此先要得到超级块的数据，之后在缓冲区中查找是否存在与即将删除的数据块有关的缓冲块，如果有，并且它的 bh->b_count=1，就将它的脏标志 bh->b_dirt 和有效标志 bh->b_uptodate 都清 0。这样做的原因是 bh->b_count=1 表明该缓冲块仅被当前进程引用，既然当前进程已经要删除它了，自然也就没有必要将其回写到硬盘或作为有效数据。

然后，开始执行删除硬盘上对应的数据块的操作。先要依据超级块中所存储的信息计算出

要删除的数据块在硬盘数据区中的实际位置，即 block -= sb->s_firstdatazone −1。在得到了要删除的数据块在硬盘数据区中的真实位置后，接下来就只需要在逻辑块位图中将该数据块对应的位置置 0 即可。由于这个操作改变了某个逻辑块位图对应的缓冲块，因此缓冲块的脏标志 sb->s_zmap[block/8192]->b_dirt 也需要置 1。

在利用 i 节点信息将 hello.txt 文件在硬盘上对应的数据删除之后，开始执行删除 i 节点自身的操作。不要忘了，此时，hello.txt 文件对应的 i 节点有两份：一份在内存中，用于支持运算操作；一份在硬盘中，用于支持 hello.txt 文件在硬盘上的存储。由于删除 i 节点本身也是一个运算过程，因此首先要根据内存中 i 节点的信息删除硬盘上 hello.txt 文件对应的 i 节点。方式也同样是将高速缓冲区中该 i 节点对应的 i 节点位图清 0。执行这一操作的函数是 free_inode，对应代码如下：

```
//代码路径：fs/bitmap.c：
void free_inode(struct m_inode * inode)
{
    struct super_block * sb;
    struct buffer_head * bh;

    if (!inode)
        return;
    if (!inode->i_dev) {
        memset(inode,0,sizeof(*inode));
        return;
    }
    if (inode->i_count>1) {
        printk("trying to free inode with count=%d\n",inode->i_count);
        panic("free_inode");
    }
    if (inode->i_nlinks)
        panic("trying to free inode with links");
    if (!(sb = get_super(inode->i_dev)))
        panic("trying to free inode on nonexistent device");
    if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
        panic("trying to free inode 0 or nonexistant inode");
    if (!(bh=sb->s_imap[inode->i_num>>13]))
        panic("nonexistent imap in superblock");
    if (clear_bit(inode->i_num&8191,bh->b_data))
        printk("free_inode: bit already cleared.\n\r");
    bh->b_dirt = 1;
    memset(inode,0,sizeof(*inode));
}
```

由于删除 i 节点的操作也使得 i 节点位图所在的高速缓冲块中的内容发生了改变，因此将缓冲块的“脏标志”置 1，以在合适的时机与硬盘上的数据进行同步。

总之，被更改了内容的逻辑块位图和 i 节点位图所占用的缓冲块的 b_dirt 标志都会被置 1，这样，当它们同步到硬盘上的时候，刚才的这些设置就会生效，hello.txt 文件将永远地从设备上消失。

硬盘上的 i 节点删除之后，hello.txt 文件在内存 i 节点表中的 i 节点也就没有意义了。因此需要执行 memset(inode, 0, sizeof(* inode))，将这个记录全部在物理上清 0。到此为止，hello.txt 对应的 i 节点已经删除完毕，并返回到调用 iput(inode) 的 sys_unlink 函数中。

4.7.3 对 hello.txt 文件所在的 user 目录做处理

由于 hello.txt 文件在 user 目录下，因此最后还需要对 user 目录对应的 i 节点进行处理。即通过调用 iput(dir) 释放 user 目录对应的 i 节点。如图 4-43 所示：

图 4-43　user 目录对应的 i 节点的处理

对应代码如下：

```
// 代码路径：fs/namei.c：
int sys_unlink(const char * name)
{
    ...
    iput(dir);
    ...
}
```

由于本实例开始时就假定没有其他的进程引用 /mnt/user/hello.txt 文件，并且 user 目录也没有其他进程引用，因此 user 目录对应的 i 节点的引用计数会变为 0。对应的代码为：

```
// 代码路径：fs/inode.c：
void iput(struct m_inode * inode)
{
    ...
   inode->i_count--;
   return;
}
```

user 目录对应的 i 节点的 i_count 字段置 0，表示该 i 节点已被释放，能够作为空闲 i 节点再次使用。

至此，文件删除操作完成。

4.8 本章小结

操作系统对文件的一切操作，都可以分为两个方面：对文件管理信息的操作以及对文件自身

的操作。新建、打开、关闭、删除文件属于对文件管理信息的操作，读文件属于对文件自身的操作，写文件和修改文件则涉及两个方面。

操作系统在操作文件管理信息时，先确定进程要操作的文件以及明确进程与该文件的关系。操作系统在 i 节点表中登记文件的 i 节点（如打开已有文件）或新建 i 节点（如新建文件）。通过 i 节点，使进程与文件建立关系或解除关系（如打开或关闭文件）。

操作系统在操作文件自身时，凭借缓冲区与硬盘进行数据交互（如读、写或修改文件）。写、修改文件操作会改变文件管理信息，改变将被记录在文件系统 i 节点管理表、i 节点位图、逻辑块位图等数据结构上，以保证文件自身的内容与自身的管理信息相匹配。

第 5 章

用户进程与内存管理

操作系统在本章以前所做的一切准备工作，都是为了对用户进程的运行提供支持，以及对多个用户进程的执行进行组织管理和协调。因此，操作系统中最核心的概念就是“进程”。本章将详细介绍用户进程在操作系统的支持、组织、管理和协调下是如何执行的，掌握了本章的内容，就可以对操作系统最核心的运作机制有一个非常清晰的了解。

本章内容分为两个部分：第一部分包括 5.1 节至 5.4 节，以一个用户进程 str1 的“创建”、“程序加载”、“运行”和“退出”为例，对用户进程的工作方式进行了完整的分析和阐述，为第一阶段；第二部分包括 5.5 节至 5.8 节，详细讲述了多个用户进程在内存中如何“同时”工作，以及系统如何对用户进程进行保护，为第二阶段。

5.1　用户进程的创建

在第 2 章中，我们已经对进程 1 和进程 2 的创建进行了详细的介绍。从技术的角度来讲，用户进程与这两个进程在创建方式上大体相同，具体操作步骤可以参考 2.2 节的内容。但是，每个进程都有自己的特征，表现为这个新建用户进程的数据信息也会有不同之处。本节将以创建 str1 这个用户进程为例，对创建 str1 进程的原理及其独有的数据信息特征进行详细讲解。

5.1.1　为创建进程 str1 准备条件

硬盘上现在有一个叫做 str1 的可执行文件，用户在 shell 界面上输入一条指令 ./str1，于是 shell 程序便会响应并解析这条指令，本章第一阶段的内容就由此开始了。

经解析得知，用户现在要执行 str1 这个程序，于是 shell 将调用 fork 函数开始创建进程，接下来会产生 int 0x80 软中断，执行代码（include/unistd.h）如下：

```
// 代码路径：include/unistd.h：
#define _syscall0(type,name) \
type name(void) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
```

```
    : "=a" (__res) \
    : "0" (__NR_##name)); \
if (__res >= 0) \
    return (type) __res; \
errno = -__res; \
return -1; \
}
```

展开后表现为：

```
int fork(void)
{
long __res;
__asm__ volatile ("int $0x80"
    : "=a" (__res)
    : "0" (__NR_ fork));
if (__res >= 0)
    return (int) __res;
errno = -__res;
return -1;
}
```

软中断产生后，system_call 函数会对此响应，并最终通过 call _sys_call_table(,%eax, 4) 这行代码映射到 sys_fork 这个函数中，具体的映射方式在 2.2.1 节中已有详细的讲解。执行代码（kernel/system_call.s）如下：

```
//代码路径：kernel/system_call.s：
_system_call:
    cmpl $nr_system_calls-1,%eax
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx              # push %ebx,%ecx,%edx as parameters
    pushl %ebx              # to the system call
    movl $0x10,%edx                 # set up ds,es to kernel space
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx                 # fs points to local data space
    mov %dx,%fs
    call _sys_call_table(,%eax,4)
 ...
 _sys_fork:
 ...
```

以上代码具体的执行步骤在 2.2.1 节中也已经讲解过。

之后，调用 find_empty_process 函数，分别为 str1 进程申请一个可用的进程号和在进程槽 task[64] 中为该进程申请一个空闲位置。我们这里假设 str1 这个进程就是怠速以后第一个申请的用户进程，那么通过 2.8.3 节中的介绍可以得知，申请到的进程号是 5，在进程槽 task[64] 中找到的空闲位置是第 5 项。

执行代码如下：

```
// 代码路径：kernel/system_call.s：
...
   _sys_fork:
call _find_empty_process
...

// 代码路径：kernel/fork.c：
int find_empty_process(void)
{
    int i;

    repeat:
         if ((++last_pid)<0) last_pid=1;
         for(i=0 ; i<NR_TASKS ; i++)
              if (task[i] && task[i]->pid == last_pid) goto repeat;
         for(i=1 ; i<NR_TASKS ; i++)
         if (!task[i])
              return i;
     return -EAGAIN;
}
```

获取进程号和获取进程槽中空闲项的情况，以及 str1 在进程槽中所占的位置，如图 5-1 所示：

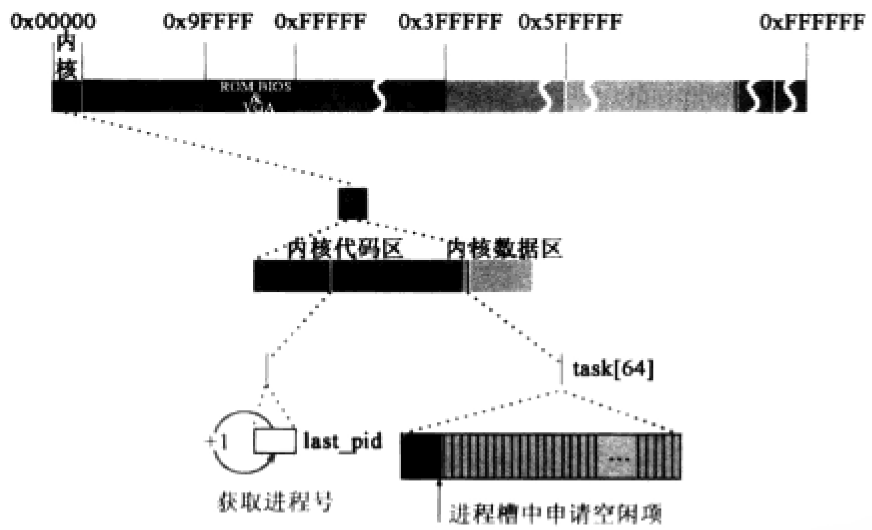


图 5-1　获取 str1 进程占用的进程号和进程槽

具体的执行步骤在 2.1.2 节中已经说明。这种申请进程号和寻找空闲项的方式不仅仅适用于进程 str1，也适用于其他所有用户进程。

5.1.2　为 str1 进程管理结构找到存储空间

继续完成第一阶段工作中创建用户进程 str1 的任务。通过调用 copy_process 函数，准备将 shell 进程的管理结构复制给 str1，执行代码如下：

```
// 代码路径：kernel/system_call.s：
_sys_fork:
...
testl %eax,%eax
```

```
    js 1f
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call _copy_process
...
```

在主内存中为用户进程 str1 申请一个页面的空间，在主内存区中申请到的空闲页面如图 5-2 右上部分所示，并在图 5-2 左下部分的位于内核数据区的 mem_map 管理结构中进行登记，在图中用绿色标记了出来。有关 mem_map 管理结构的更多知识，在 2.1.4 节中已详细讲解。然后，将这个空闲页面与 5.1.1 节中在 task[64] 中申请到的第 5 个空闲项相挂接，执行代码如下：

```
//代码路径：kernel/fork.c：
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    p = (struct task_struct *) get_free_page();
    ...
    task[nr] = p;
    ...
}
```

task[64] 与用做 str1 进程管理结构的空闲页面的挂接情况如图 5-2 右下部分所示：

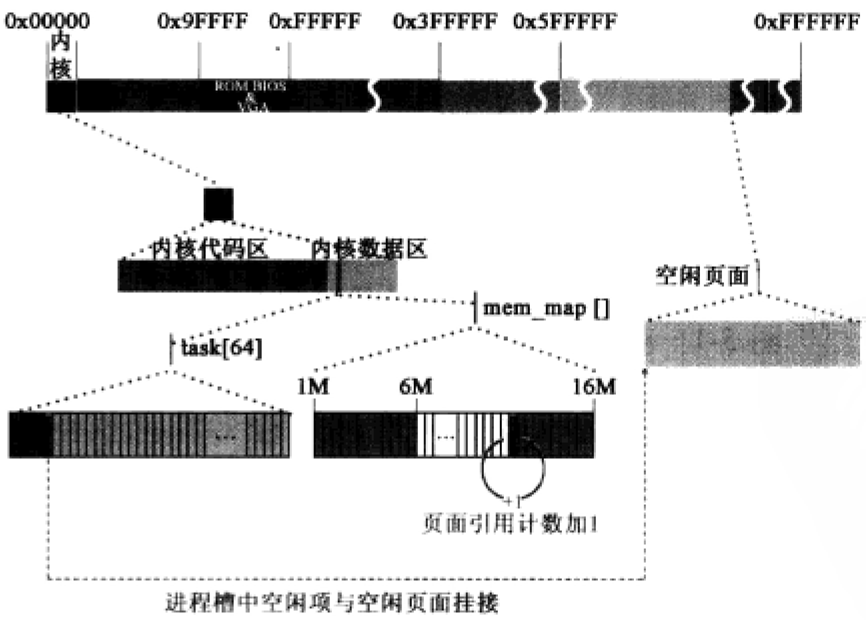


图 5-2　为 str1 进程管理结构申请页面并与操作系统挂接

具体执行步骤在 2.2.3 节和 2.2.4 节中已经说明。这里需要强调的是：这个内存页面除了用来存储 str1 自身的管理结构之外，还有一个重要作用，就是用来存储 str1 进程的“内核栈”数据。

在 Linux 0.11 的程序（包括用户程序和内核程序）执行过程中，各个段寄存器（如 cs、ds、ss、es 等）中表示优先级的“两位”数据只有两种可能：“0”或“3”。如果是 0，则主机将视此

时的程序执行状态为“内核态”；如果是 3，则主机将视此时的程序执行状态为“用户态”，计算机的硬件只“认”段寄存器中表示优先级的“两位”数据，以此来判断优先级，至于当前是在执行用户的程序还是内核的程序，硬件完全不关心。

一般情况下，用户程序的优先级为“3”，它运行时需要通过调用库函数产生软中断 int 0x80，这样就可以使优先级转变为“0”，于是内核程序开始执行，等到执行完毕后，需要中断返回，通过执行 iret 这个指令，又会将优先级的值转变为“3”，但是，这些都是现象，是设计操作系统时人为规定的，并不是本质，段寄存器优先级的数值才是本质。

系统之所以对“用户态”和“内核态”区别对待是有原因的。我们还是以这个页面的应用为例，用户程序执行时，通常是用户态，这时候用户程序会致使多少数据需要压栈，操作系统的设计者是不可能预知的。所以，就根据程序执行时的具体需要，在主内存中分配页面，以此来承载栈数据。但是，一旦内核程序开始执行，即处于内核态，那么内核程序运行时会有多少数据需要压栈，系统的设计者是可以预知的，尤其可以预知最多会有多少数据需要压栈。所以，设计者敢于将内核栈的数据与进程管理结构的数据放在同一个页面内，这些内核栈的数据将与进程管理结构的数据分别处于同一页面的两端，将来用户进程 str1 进入内核态的时候，压栈的数据将向着 str1 管理结构所在的位置不断地积累，但可以预计，绝对不可能覆盖掉进程管理结构的数据，这样就节省了页面的开销。

5.1.3 复制 str1 进程管理结构

继续完成第一阶段工作中创建用户进程 str1 的任务。由于当前进程是 shell，所以，现在调用 copy_process 函数，以当前进程 shell 的管理结构为母本，将其复制到 5.1.2 节中刚刚申请到的页面的起始端。这就是 str1 进程的管理结构雏形，之后，由于 str1 进程现在还没有创建完毕，所以要把它设置为不可中断等待状态，执行代码如下：

```
// 代码路径：kernel/fork.c：
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    *p = *current;
    p->state = TASK_UNINTERRUPTIBLE;
    ...
}
```

复制的过程如图 5-3 所示：

之后以这个雏形为母本，对 str1 的管理结构进行有针对性的调整，这个调整就包括重新设置 5.1.2 节中介绍过的内核栈的栈顶指针，执行代码如下：

```
// 代码路径：kernel/fork.c：
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
```

```
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    p->tss.esp0 = PAGE_SIZE + (long) p;
    ...
}
```

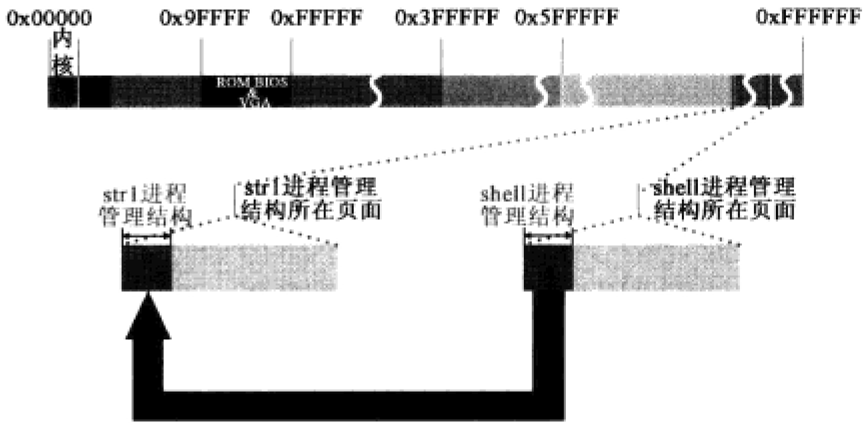


图 5-3　将 shell 进程的管理结构复制给 str1 进程

str1 进程管理结构在其他方面的调整已经在第 2 章的 2.2.7 节中详细讲过。

从执行代码可以看出，由于此时 str1 进程还没有运行过，所以此时栈顶指针就指向 5.1.2 节中申请到的页面末端。

这种进程管理信息的复制方式适用于所有的用户进程，比如，如果由当前进程 shell 创建 str1，就将 shell 的进程管理信息复制给 str1 作为雏形；如果 str1 是当前进程，由它来创建 str2，那么就把 str1 的进程管理信息复制给 str2 作为雏形，以此类推。

5.1.4　确定 str1 进程在线性空间中的位置

继续完成第一阶段工作中创建用户进程 str1 的任务。下面将调用 copy_mem 函数为新建的用户进程 str1 在线性地址空间中指定一个位置，并将该位置与 str1 进程管理结构中的 ldt[3] 挂接，执行代码如下：

```
// 代码路径：kernel/fork.c：
int copy_mem(int nr,struct task_struct * p)
{
    ...
    new_data_base = new_code_base = nr * 0x4000000;
    p->start_code = new_code_base;
    set_base(p->ldt[1],new_code_base);
    set_base(p->ldt[2],new_data_base);
    ...
}
```

这个过程如图 5-4 所示：

这里对 str1 进程设置的线性地址范围是专供内核使用的，对普通用户程序而言，能够接触到的只有逻辑地址，即在所有用户程序看来，它们的地址范围都是 0 ～ 64MB 这个空间内；而内核根据不同的用户程序所在的进程将其对应成线性地址，并将此线性地址解析成“页目录项”、“页

表项”、“页内偏移”，最终映射到物理页面内。在整个过程中，用户进程完全不知道自己的程序的线性地址是多少，更不可能知道物理地址是多少。

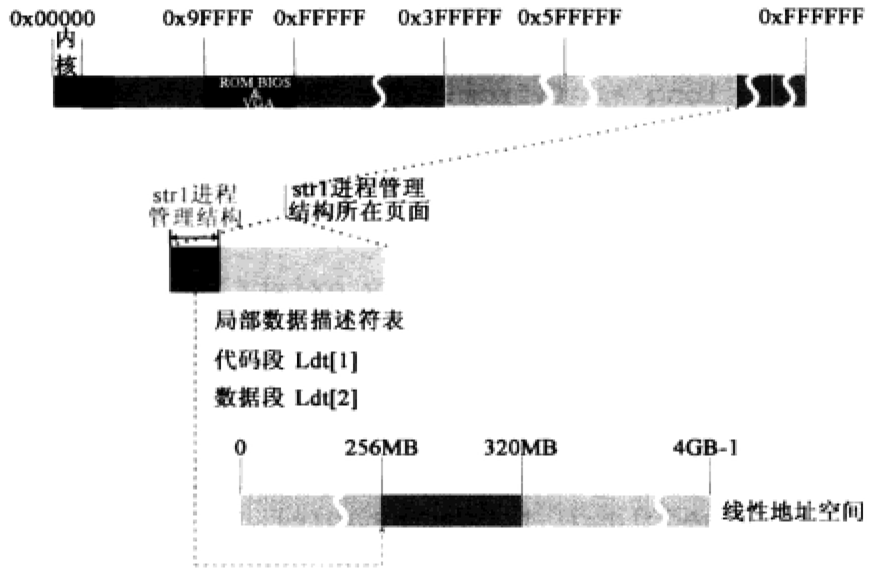


图 5-4 确定 str1 进程在线性空间中的位置

我们在 5.1.1 节中已经介绍过，现在在 task[64] 中申请到的空闲项是第 5 项，则在线性地址空闲中对应的基地址位置就应该是：4×64MB（进程槽各项编号依次为 0、1、2、3、4……所以第 5 项应该是 4×64）。这个位置就是进程 5 的基地址，每一个用户进程创建时，都要在线性地址空间中指定基地址，而且最大寻址范围都是 64MB，所以在 Linux 0.11 中，4GB 的线性地址空间，总共可以容纳 64 个进程。

另外，还要为复制 str1 进程的页表和设置它的页目录项做准备，准备工作包括通过采集 shell 进程的局部数据描述符表中的代码段和数据段的信息以及段限长来设置 str1 全局描述符表的数据，等等，这部分内容在 2.2.8 节中已详细讲解过。之后，就可以为 str1 进程复制页表和新建页目录项了。

5.1.5 复制 str1 进程页表并设置其对应的页目录项

继续完成第一阶段工作中创建用户进程 str1 的任务。接下来将调用 copy_page_tables 函数把 shell 进程对应页表的全部表项复制给 str1 进程，并为此新建一套页目录项，执行代码如下：

```
// 代码路径：kernel/fork.c：
int copy_mem(int nr,struct task_struct * p)
{
    ...
if (copy_page_tables(old_data_base,new_data_base,data_limit))
    ...
}
```

正因为 str1 进程的页表信息是由其父进程 shell 复制过来的，所以 str1 进程现在暂时先与其父进程 shell 共享相同的内存页面，将来 str1 有了属于自己的程序后，再另行调整。有了新的页表，就要为此新增添一套页目录项，因为每个进程都要有一套属于自己的页目录项，这样才能确保将来线性地址到物理地址的映射的唯一性。执行复制页表和设置页目录项的具体代码如下：

```
// 代码路径：mm/memory.c：
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
    ...
    if ((from&0x3fffff) || (to&0x3fffff))
        panic("copy_page_tables called with wrong alignment");
    from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
    to_dir = (unsigned long *) ((to>>20) & 0xffc);
    size = ((unsigned) (size+0x3fffff)) >> 22;
    for( ; size-->0 ; from_dir++,to_dir++) {
        if (1 & *to_dir)
            panic("copy_page_tables: already exist");
        if (!(1 & *from_dir))
            continue;
        from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
        if (!(to_page_table = (unsigned long *) get_free_page()))
            return -1;    /* Out of memory, see freeing */
        *to_dir = ((unsigned long) to_page_table) | 7;
        nr = (from==0)?0xA0:1024;
        for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
            this_page = *from_page_table;
            if (!(1 & this_page))
                continue;
            this_page &= ~2;
            *to_page_table = this_page;
            if (this_page > LOW_MEM) {
                *from_page_table = this_page;
                this_page -= LOW_MEM;
                this_page >>= 12;
                mem_map[this_page]++;
            }
        }
    }
    invalidate();
    return 0;
}
```

具体的复制页表和设置页目录项的动作在第 2 章 2.2.8 节中已详细说明。

图 5-5 形象地展示了页表复制和页目录项设置的过程，读者可以通过图中直观的展示加深对这一过程的理解。

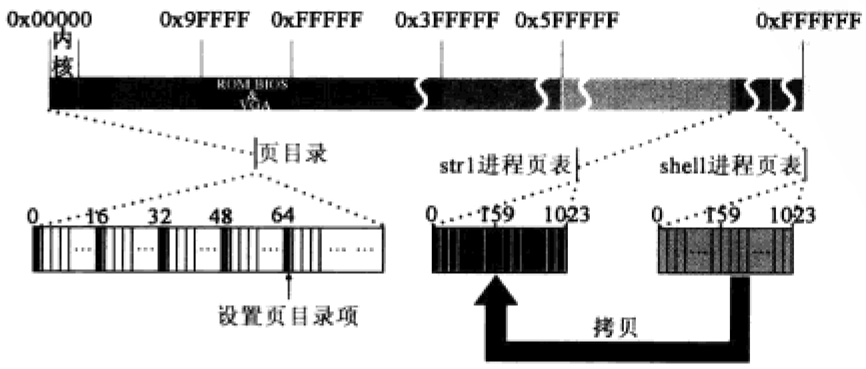


图 5-5　为 str1 进程复制页表并设置 str1 进程的页目录项

以下将对该动作的普遍意义和深刻内涵进行深入讲解。以后，每个用户进程在创建的时候，

其页表的内容都是从其父进程的页表中复制过来的，而且都要新添加一套页目录项，这个规则适用于所有用户进程。比如，用 str1 作为父进程创建 str2，那么，在创建 str2 的页表时，就把 str1 的页表内容拷贝过去，并重新为 str2 制定一套页目录项，以此类推。

那么这里就出现了一个问题，shell 进程如此费周折地为 str1 进程复制页表，并创建页目录表，等到 str1 进程将来有了自己的程序时，又要将这些关系解除，那么现在所做的这些工作还有什么意义呢？

这个意义是重大的，因为有一个对 str1 进程独立运行来说非常关键的步骤需要基于上述复制页表和设置页目录项的操作来解决。即，str1 进程开始执行后并没有对应的程序，这就需要加载程序，于是就要调用 execve 函数。然而，调用 execve 函数的这行代码又在哪里呢？很显然，只能在它的父进程——shell 程序中，这就意味着，str1 进程一旦开始执行，必须有能力执行 shell 程序里面的代码，这样才能开展以后的工作，直到它自己的子程序加载完毕。

要想具备这种能力，只有一个办法，就是让 str1 进程在创建之初就能够共享 shell 程序所占用的页面，一旦执行，共享能力马上生效。而这里的复制页表和设置页目录项的工作就是为 str1 能够具备这个能力所做的准备工作。str1 和 shell 的线性地址肯定不同，但是通过复制页表和设置页目录项，就可以使不同的线性地址映射到相同的物理地址上，以此实现共享。

5.1.6 调整 str1 进程中与文件相关的结构

继续完成第一阶段工作中创建用户进程 str1 的任务。下面继续调整 str1 进程管理结构中与文件相关的部分。由于 shell 进程中管理文件结构 *filp[20] 中的内容已经全部都复制给了 str1，现在相当于 str1 也对应着这些文件，因此就需要将文件管理结构 file_table[64] 中的这些文件的引用次数加 1。同理，“当前工作目录 i 节点引用次数”、“根目录 i 节点引用次数”以及“执行文件 i 节点引用次数”等内容都拷贝给了 str1 进程，所以引用次数都要加 1，执行代码如下：

```
//代码路径：kernel/fork.c：
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    for (i=0; i<NR_OPEN;i++)
        if (f=p->filp[i])
            f->f_count++;
    if (current->pwd)
        current->pwd->i_count++;
    if (current->root)
        current->root->i_count++;
    if (current->executable)
        current->executable->i_count++;
    ...
}
```

设置过程如图 5-6 所示：

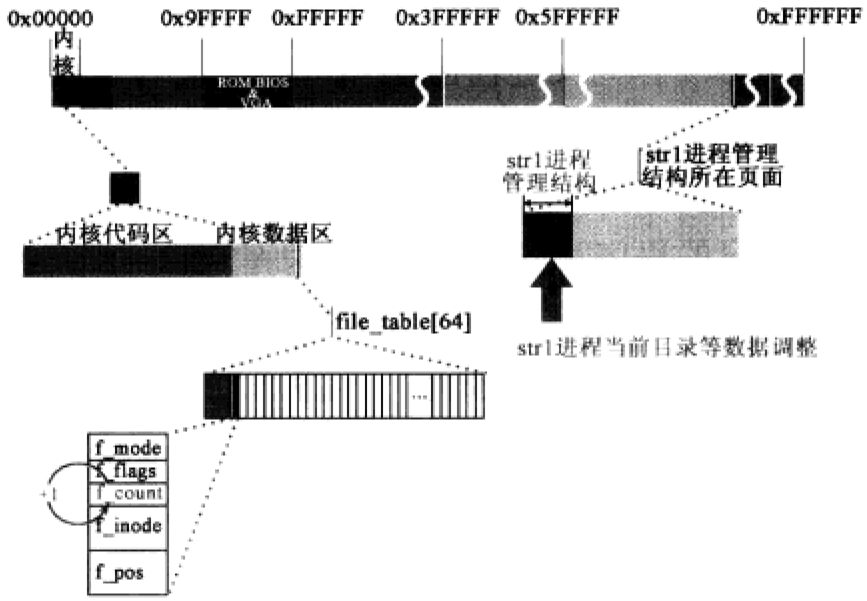


图 5-6　调整 str1 进程管理结构中与文件相关的内容

在 Linux 0.11 中，是允许多个进程操作同一个文件的。由于 i 节点是文件自身全部属性的唯一载体，所以多个进程可通过对应到同一个 i 节点的方式来操作同一个文件。这也带来一个问题，如果某个进程与一个 i 节点的关系解除了，就不能把这个 i 节点从 i 节点管理表 inode_table 中彻底删除掉，因为其他进程与此 i 节点的关系并没有解除，于是系统的设计者设计了一套方案，即对 i 节点的引用进行计数，解除一组关系，就将引用计数减少；反之，增加一组关系，就将引用计数增加，以此来判别 i 节点是否被引用。

由于 shell 进程的管理结构信息全部都复制给了 str1 这个用户进程，所以就自然导致 str1 进程的“当前工作目录 i 节点”、“根目录 i 节点”以及“执行文件 i 节点”与 shell 进程相同，这意味着，现在又多了一组对这些 i 节点的引用关系，所以，这些 i 节点的引用次数都要累加 1，以此表示 i 节点的使用情况。

这一点对于所有的用户进程都适用，如果将来 str1 进程又创建了 str2，则这些文件的引用次数会继续加 1。

5.1.7　建立 str1 进程与全局描述符表 GDT 的关联

继续完成第一阶段工作中创建用户进程 str1 的任务。将用户进程 str1 的任务状态描述符表 TSS，以及局部数据描述符表 LDT 挂接在全局描述符表的指定位置。所有的进程都要把这两套描述符表挂接在全局描述符表中，这样在进程切换时，系统就可以通过全局描述符表找到当前进程的 LDT 和 TSS，并把相关的信息保存在当前进程的 LDT 和 TSS 中。同时，还可以找到即将切换到的进程的 LDT 和 TSS，并用它们里面存储的数据信息来设置局部数据寄存器和任务状态寄存器。可见，现在所做的挂接工作是系统能够进行进程间切换的最根本的保障。将 str1 进程的 LDT 和 TSS 挂接到全局描述符表的代码如下：

```
// 代码路径：kernel/fork.c：
 int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
```

```
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
    ...
}
```

设置过程如图 5-7 所示：

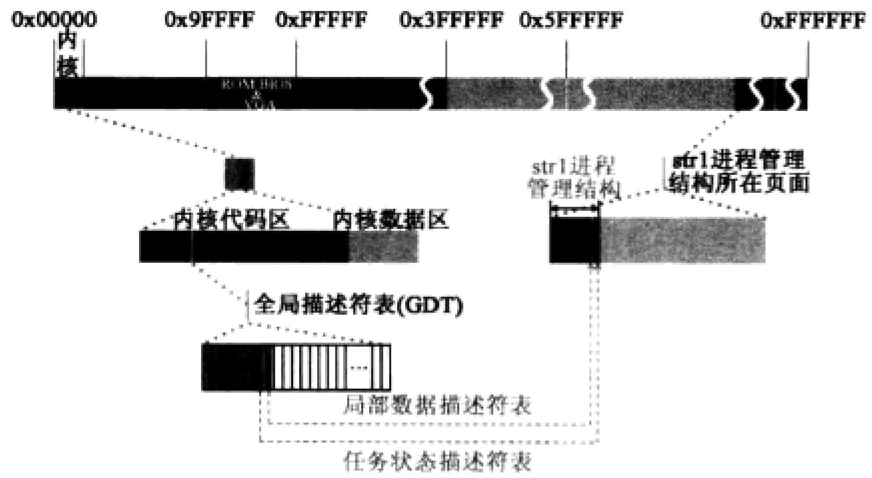


图 5-7 str1 进程的 TSS 和 LDT 与全局描述符表 GDT 挂接

5.1.8 将 str1 进程设为就绪态

继续完成第一阶段工作中创建用户进程 str1 的任务。现在用户进程 str1 的管理信息已经调整完成，可以参与多进程的轮询了，所以将它的状态设置为“就绪态”。至此，这个用户进程的管理结构就创建完毕了，执行代码如下：

```
// 代码路径：kernel/fork.c：
 int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
{
    ...
    p->state = TASK_RUNNING;
    ...
}
```

这一过程如图 5-8 所示，其中的 state 被设置为 TASK_RUNNING：

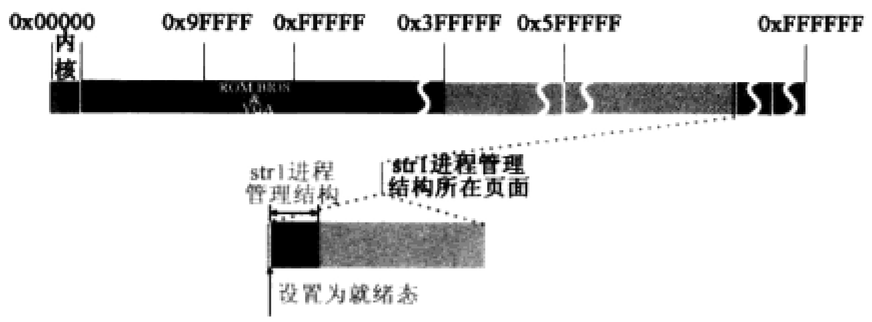


图 5-8 将 str1 设置为就绪态

到此为止，本章第一阶段的工作中，创建用户进程 str1 的任务就完成了。接下来，要为这个用户进程加载程序，也就是从硬盘上加载 str1 这个可执行文件所对应的程序。

5.2 为用户进程 str1 的加载做准备

在 2.7 节中，已经对 shell 程序的加载进行了详细的讲解，其实为用户进程 str1 的加载做准备与为 shell 程序的加载做准备的方式是大体相同的，同样包括对外围的准备、对 str1 可执行文件的检测、对 str1 进程管理结构的针对性调整，以及最终设置 EIP 使之开始执行这 4 个部分。但是，str1 程序自身与 shell 程序肯定是不同的，所以也将导致最终的加载结果必然有所不同，本节我们将对加载 str1 的准备工作进行详细介绍。

5.2.1 为 str1 进程加载自身对应的程序做准备

shell 创建完 str1 进程的管理结构之后，通过 shell 自身的代码继续执行，最终会切换到 str1 进程去执行，由 str1 进程来加载自己对应的程序。str1 进程开始执行后，会调用 execve 函数，产生软中断，并映射到 sys_execve 函数中，最终调用 do_execve 函数，为用户进程 str1 的加载做准备。在此过程中，参数、环境变量、EIP 在栈空间中的地址都会被压栈，此处执行过程与 2.7.3 节所介绍的执行过程完全一致，执行代码如下：

```
// 代码路径：shell 中的代码：
execve(...);

// 代码路径：kernel/system_call.s：
_sys_execve:
    lea EIP(%esp),%eax
    pushl %eax
    ...
_sys_execve:
    ...
    call _do_execve
    ...
```

唯一区别就是，创建 shell 进程时需要的参数和环境变量的信息是由 main.c 中设置的数据来提供的；但是，创建 str1 用户进程时，参数和环境变量的具体信息就要由 shell 程序自身的代码来提供了，将来任何一个用户进程所对应的参数和环境变量都要由创建它的父进程来提供。

进入 do_execve 函数后，先要做外围准备工作，即为管理 str1 进程参数和环境变量所占用的页面做准备，设置 page 管理结构并初始化清 0，以备使用。执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
   char ** argv, char ** envp)
{
    ...
    unsigned long page[MAX_ARG_PAGES];
    ...
```

```
    for (i=0 ; i<MAX_ARG_PAGES ; i++)
        page[i]=0;
    ...
}
```

设置过程如图 5-9 所示：

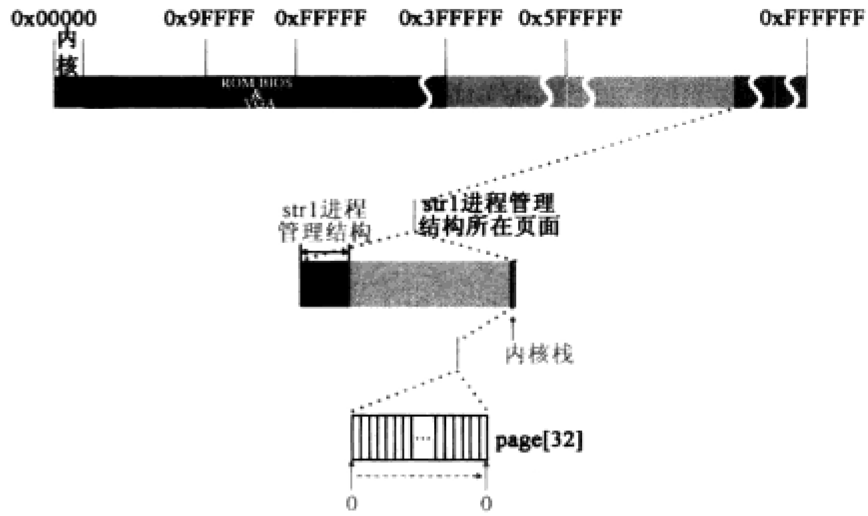


图 5-9　设置 str1 进程参数和环境变量所占的页面管理结构

5.2.2　读取 str1 可执行文件的 i 节点并统计参数和环境变量

这里要将参数和环境变量的设置等外围准备工作暂停，转而通过调用 namei 函数从硬盘上获取 str1 这个可执行文件的 i 节点。在 2.7.5 节中我们介绍过，读取这个 i 节点的目的就是为了对这个 str1 文件进行检测，看其是否具备载入条件，执行代码如下：

```
//代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    if (!(inode=namei(filename)))
    ...
}
```

之后，再继续回去做外围准备工作，把参数和环境变量的个数统计出来，这已经在 2.7.6 节中讲过，执行代码如下：

```
//代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    argc = count(argv);
    envc = count(envp);
    ...
}
```

读取 i 节点和统计个数的过程如图 5-10 所示：

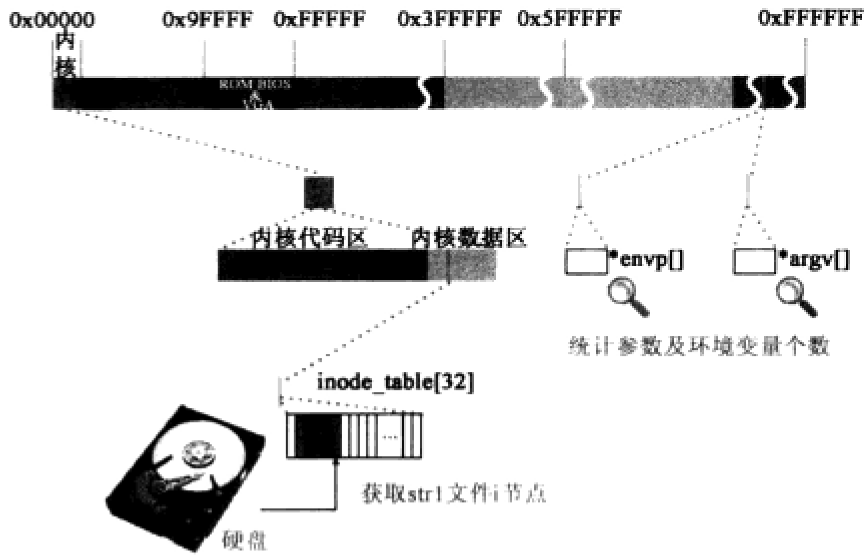


图 5-10 读取 str1 可执行文件的 i 节点并统计参数和环境变量的个数

接下来就要对这个 i 节点进行分析了，包括分析它是不是常规文件，以及对一些与用户有关的属性信息进行分析，理由在 2.7.7 节中已经说明，执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    if (!S_ISREG(inode->i_mode)) {        /* must be regular file */
        retval = -EACCES;
        goto exec_error2;
    }
    i = inode->i_mode;
    e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
    e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
    if (current->euid == inode->i_uid)
        i >>= 6;
    else if (current->egid == inode->i_gid)
        i >>= 3;
    if (!(i & 1) &&
        !((inode->i_mode & 0111) && suser())) {
            retval = -ENOEXEC;
            goto exec_error2;
    }
    ...
}
```

5.2.3 读取 str1 可执行文件的文件头

继续为 str1 程序的加载做准备，在对 str1 文件的 i 节点进行分析以后，这里要将这个文件的文件头从硬盘上载入缓冲块中，以便进一步分析，理由在 2.7.8 节中已经说明，执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
```

```
    if (!(bh = bread(inode->i_dev,inode->i_zone[0])))
    ...
}
```

读取 str1 可执行程序文件头的主要过程如图 5-11 所示：

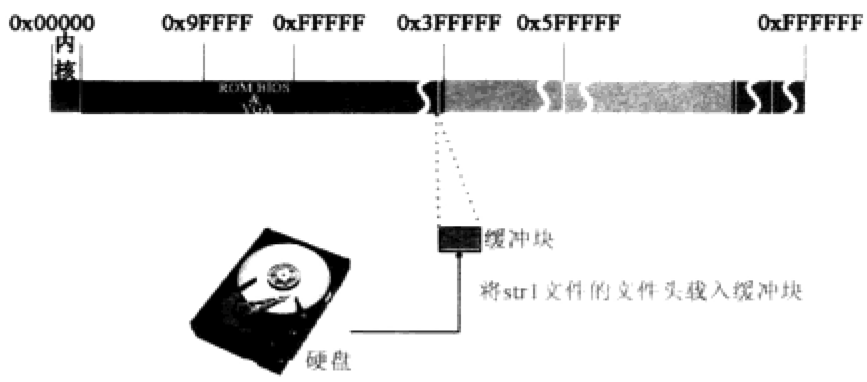


图 5-11　读取 str1 可执行程序文件头

5.2.4　对 str1 可执行程序文件头进行分析

继续为 str1 程序的加载做准备，文件头都已经进入了缓冲区中，现在要对这个文件头进行备份。之后，对这个文件头进行分析，检查 str1 文件是不是脚本文件，当然，这个文件肯定不是脚本文件，于是 “if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) ” 这行语句后面大段的代码又被跳过了，直接执行下面释放承载 str1 内容的缓冲块。最后，对文件头中提供的信息进行进一步的分析，包括代码段、数据段等的长度是否符合 Linux 0.11 的规定，以及 str1 文件自身是不是有错误，执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    ex = *((struct exec *) bh->b_data);
    if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang))
    {
    ...
    }
    brelse(bh);
    if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
        ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
        inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex))
{
        retval = -ENOEXEC;
        goto exec_error2;
    }
    if (N_TXTOFF(ex) != BLOCK_SIZE) {
        printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename);
        retval = -ENOEXEC;
        goto exec_error2;
    }
    ...
}
```

这些备份和检测的理由在 2.7.10 节至 2.7.11 节中已经详细讲解了。

备份及分析文件头的过程如图 5-12 所示：

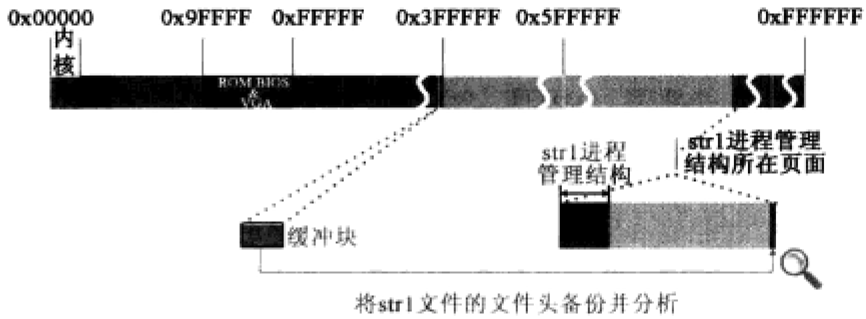


图 5-12　备份 str1 可执行文件的文件头并进行分析

5.2.5　拷贝 str1 可执行程序的参数和环境变量

继续为 str1 程序的加载做准备。现在对可执行文件 str1 的检测已经完毕了，得知这个文件确实是一个可以正常使用的文件。接下来，又要做一些外围方面的准备，要把 str1 的参数和环境变量拷贝到主内存区中指定的页面内，执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
    p = copy_strings(envc,envp,page,p,0);
    p = copy_strings(argc,argv,page,p,0);
    ...
}
```

拷贝环境变量和参数的过程如图 5-13 所示：

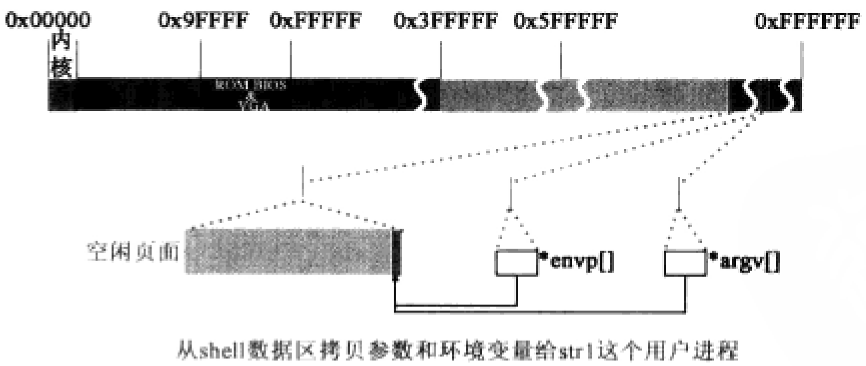


图 5-13　拷贝 str1 可执行程序的环境变量和参数

到这里，加载 str1 程序的外围准备工作和对 str1 这个可执行文件的检测工作就全部结束了。接下来，就要根据实际情况，对 str1 进程的管理结构进行调整，这也是 str1 程序加载的重要准备工作之一。

5.2.6　调整 str1 进程管理结构中可执行程序对应的 i 节点

继续为 str1 程序的加载做准备。这里对 str1 自身的进程管理结构进行调整，先要更改该文件

的 executable 字段，这个字段就是 str1 进程对应的可执行文件的 i 节点。这段内容虽然在 2.7.12 节中已经提到过，但是现在的情况和当时完全不同了。

在 2.7.12 节中，shell 进程的进程管理结构中的 executable 字段是从进程 1 拷贝过来的，进程 1 当时并没有对应任何可执行文件。但现在的情况是，str1 进程管理结构中的 executable 字段是从 shell 进程拷贝过来的，shell 进程是实实在在地对应着 shell 这个可执行文件的，所以现在 str1 进程也对应着 shell 这个可执行文件。

在 str1 进程创建时，由于 shell 管理机构中的信息全部都复制给了 str1 进程，所以这里面也一定包括 shell 进程与其可执行文件的关系，现在 str1 进程就要拥有自己的程序了，也一定会对应一个属于自己的可执行文件的 i 节点，所以原来这个关系已经没有必要继续维护下去了，这就要先把它与 shell 可执行文件的关系解除掉，这里具体表现为释放这个文件的 i 节点，并把 str1 这个可执行文件的 i 节点与 str1 进程挂接上。执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
        if (current->executable)
            iput(current->executable);
        current->executable = inode;
    ...
}
```

调整 str1 进程管理结构中与可执行文件 i 节点对应的 executable 字段的过程如图 5-14 所示：

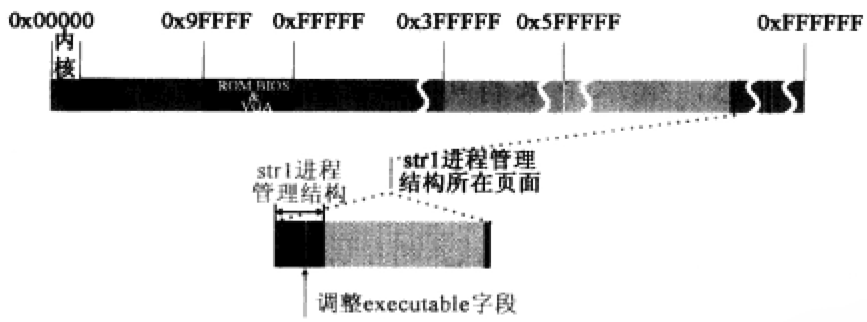


图 5-14　调整 str1 进程管理结构中的 executable 字段

值得注意的是，并不是当 str1 要加载新程序时就会把所有之前与文件的关系都解除掉。比如说，shell 进程与终端设备文件 tty0 的关系也全部遗传给了 str1 进程，str1 进程加载子程序后也有可能用到这些关系，所以这些关系并不会被解除。

5.2.7　继续调整 str1 进程管理结构——文件和信号相关的字段

继续为 str1 程序的加载做准备。这里接着调整 str1 进程管理结构，将与文件和信号相关的数据信息进行调整，调整的具体理由在第 2 章 2.7.3 节中已经讲解，执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
```

```
{
    ...
        for (i=0 ; i<32 ; i++)
        current->sigaction[i].sa_handler = NULL;
        for (i=0 ; i<NR_OPEN ; i++)
            if ((current->close_on_exec>>i)&1)
                sys_close(i);
        current->close_on_exec = 0;
    ...
}
```

其中，清空 str1 进程管理结构中与信号相关的数据、关闭已打开的文件的过程如图 5-15 所示：

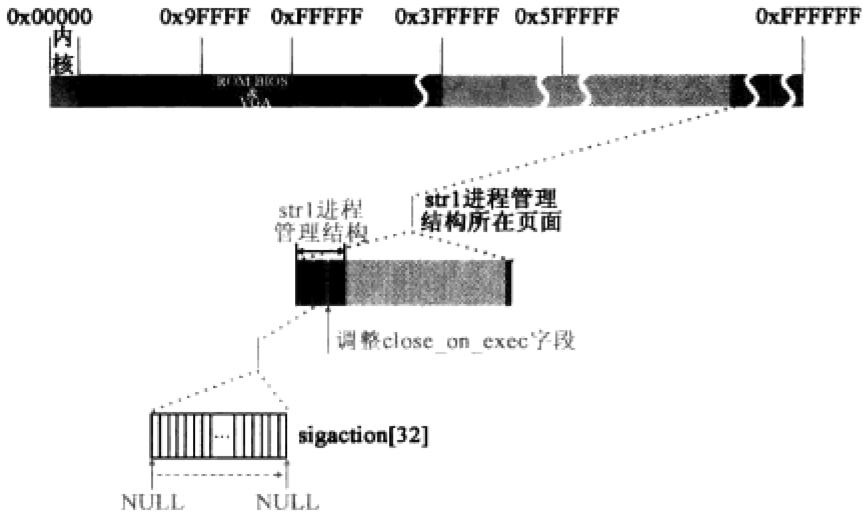


图 5-15 进一步调整 str1 进程管理结构

与信号相关的信息全部清 0，这是因为每个进程都会接收各自的信号，这些都是进程私有的信息，彼此之间不能交叉使用，所以 str1 把 shell 进程的信号信息继承下来毫无意义，而且由于信号还会影响到进程的执行状态，继承下来有可能造成混乱，因此全部清空。

5.2.8 释放 str1 进程的页表

继续为 str1 程序的加载做准备。这里要释放 str1 进程原先的代码段和数据段所占用的物理页面。str1 将要对应的程序与 shell 程序肯定有所不同，但在 5.1.5 节中已经介绍过，str1 进程现在与 shell 进程正在共享着相同的页面和管理页表，所以，要把这个共享关系解除掉，于是就调用 free_page_tables 函数来实施，执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    ...
}
```

释放页表的过程如图 5-16 所示，请读者注意其中页目录项的变化：

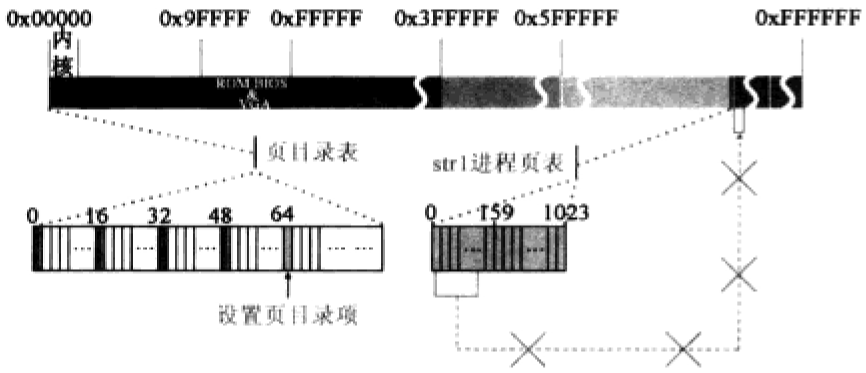


图 5-16　释放 str1 进程继承的页面

值得注意的是，原来 str1 与 shell 进程是共享页面的，共享就意味着这些页面必须是“只读”的，现在 str1 由于加载新程序不再需要这些页面中的程序了，如果还共享着原来 shell 程序的页面，一来毫无意义，二来也会影响到 shell 程序对这些页面的应用，所以关系也需要解除，具体会造成哪些影响，我们将在 5.8 节关于“页写保护”的内容中详细介绍。

5.2.9　重新设置 str1 的程序代码段和数据段

继续为 str1 程序的加载做准备。这里调用 change_ldt 函数，将根据 str1 文件头提供的信息重新设置 str1 程序代码段的描述符以及重新设置数据段描述符，执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
        p += change_ldt(ex.a_text,page)-MAX_ARG_PAGES*PAGE_SIZE;
    ...
}
```

具体表现为，代码段限长被设置为与 str1.c 文件的代码段长度相同。数据段长度设置为 0x4000000，即 64MB，之后再将参数与环境变量所在的物理页面映射到 str1 这个进程的线性地址空间内，执行代码如下：

```
// 代码路径：fs/exec.c：
static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
{
    unsigned long code_limit,data_limit,code_base,data_base;
    int i;

    code_limit = text_size+PAGE_SIZE -1;
    code_limit &= 0xFFFFF000;
    data_limit = 0x4000000;
    code_base = get_base(current->ldt[1]);
    data_base = code_base;
    set_base(current->ldt[1],code_base);
    set_limit(current->ldt[1],code_limit);
    set_base(current->ldt[2],data_base);
    set_limit(current->ldt[2],data_limit);
    __asm__("pushl $0x17\n\tpop %%fs"::);
    data_base += data_limit;
```

```
    for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
        data_base -= PAGE_SIZE;
        if (page[i])
            put_page(page[i],data_base);
    }
    return data_limit;
}
```

设置过程如图 5-17 所示：

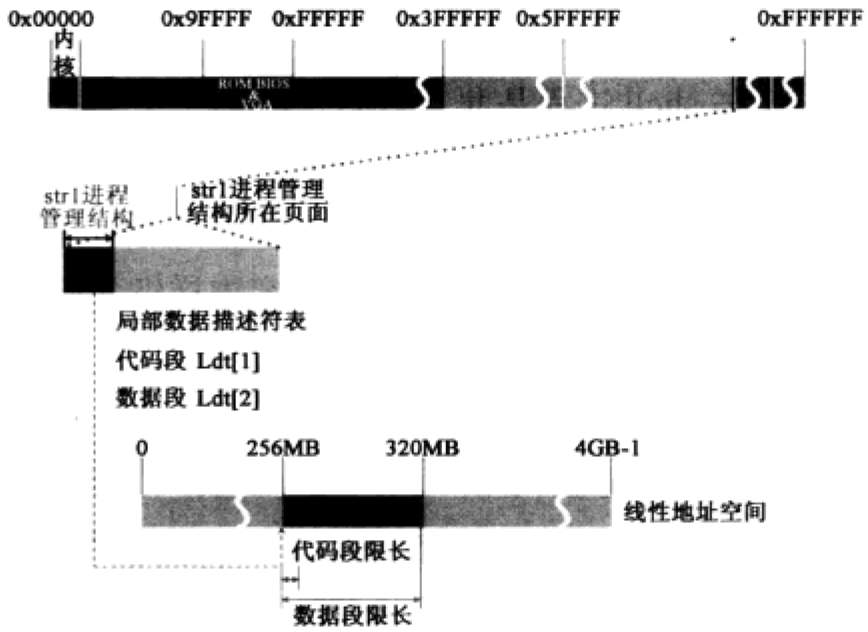


图 5-17　重新设置 str1 程序的代码段和数据段

5.2.10　创建环境变量和参数指针表

继续为 str1 程序的加载做准备。这里调用 create_tables 函数，创建环境变量和参数的指针表，并将其地址存放到“环境变量和参数所在的”主内存区的页面中，理由和详细步骤在第 2 章 2.7.18 节中已经说明。执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
p = (unsigned long) create_tables((char *)p,argc,envc);
    ...
}
```

上述过程如图 5-18 所示：

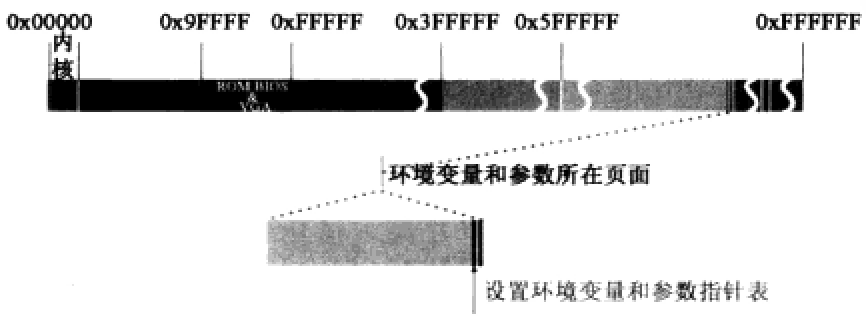


图 5-18　创建环境变量和指针表

5.2.11　继续根据 str1 可执行程序情况调整 str1 进程管理结构

继续为 str1 程序的加载做准备。对 str1 用户进程管理结构中的 brk、start_stack 等信息进行设置，理由在 2.7.20 节中已经讲解。执行代码如下：

```
// 代码路径：fs/exec.c：
int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
{
    ...
current->brk = ex.a_bss +
        (current->end_data = ex.a_data +
        (current->end_code = ex.a_text));
    current->start_stack = p & 0xfffff000;
    current->euid = e_uid;
    current->egid = e_gid;
    i = ex.a_text+ex.a_data;
    while (i&0xfff)
        put_fs_byte(0,(char *) (i++));
    ...
}
```

设置结果装载到 str1 进程管理结构所占页面的过程如图 5-19 所示：

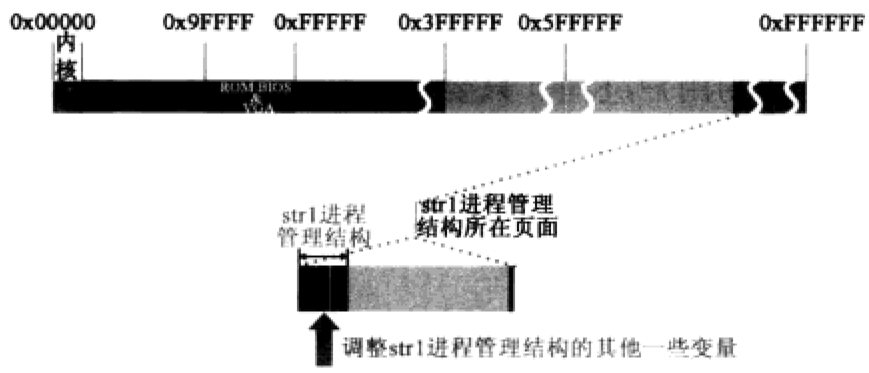


图 5-19　继续调整 str1 进程管理结构

这些信息都是直接或间接从 str1 可执行文件的文件头中采集出来的，这与从 shell 进程对应的可执行文件中采集出来的程序肯定是不一样的，所以这里必须重新设置，以支持 str1 进程在加载子程序后运行。

到此为止，为 str1 程序加载所做的工作中，针对 str1 进程管理结构自身的属性进行调整的工作就全部做完了。

5.2.12　设置 str1 可执行程序的栈指针和 eip 值

这里是 str1 程序加载的最后一步准备工作，即对 esp 进行设置，使其指向 str1 的栈底；对 eip 进行设置，使得程序一旦执行，就会发生缺页中断，具体的设置方式在 2.7.21 节中已经说明。执行代码如下：

```
// 代码路径：fs/exec.c：
    int do_execve(unsigned long * eip,long tmp,char * filename,
    char ** argv, char ** envp)
```

```
{
    ...
    eip[0] = ex.a_entry;
    eip[3] = p;
    return 0;
    ...
}
```

eip 及 esp 的设置过程如图 5-20 所示：

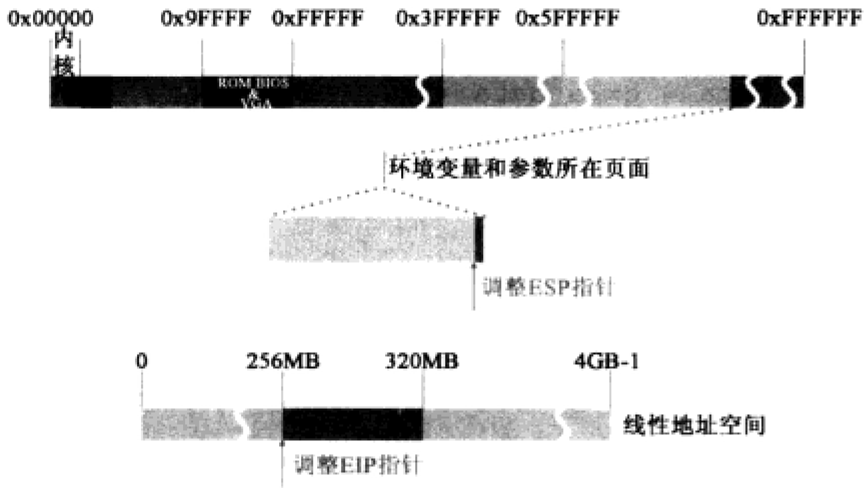


图 5-20　设置 str1 进程管理结构栈顶指针和 eip 值

现在，为 str1 程序的加载所需要做的准备工作就全部完成了，接下来，str1 程序就可以加载了。以上准备工作适用于所有的用户进程，只要它们有对应的程序，则加载方法完全一致。

5.3　对缺页中断的处理

str1 程序的加载和 shell 程序的加载一样，都是通过缺页中断根据代码执行时的实际需要进行加载的，程序没有执行到的代码，暂时不加载。每次都从外设上读取一个页面的代码内容，这一点已经在 2.7.22 节至 2.7.26 节中讲过了。等到程序加载完毕并执行时，如果需要压栈，则又需要缺页中断来支持，这时情况就会有所变化。本节将详细讲解 str1 程序的加载情况和实际运行情况。

5.3.1　产生缺页中断并由操作系统响应

缺页中断信号产生后，page_fault 这个服务程序将对此进行响应，并最终在 _page_fault 中通过“call _do_no_page”调用到缺页中断处理程序 _do_no_page: 中去执行，执行代码如下：

```
// 代码路径：mm/page.s：
 _page_fault:
    ...
    testl $1,%eax
    jne 1f
1:  call _do_no_page
...
```

进入 do_no_page 函数后，在加载 str1 程序之前，先要做如下两方面的检测。第一，str1 进程是否已经把程序加载进来了，或者产生缺页中断的线性地址值是否已经超出了程序代码的末

端。现在显然两种情况都不成立，str1 的代码内容一定是要从硬盘上加载的，而且没有超过程序代码的末端，执行代码如下：

```
// 代码路径：mm/memory.c：
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
    if (!current->executable || tmp >= current->end_data)
    ...
}
```

第二，str1 是否有可能与某个现有的进程共享代码，比如某个其他进程已经把 str1 程序加载了的情况，显然现在也不可能，执行代码是：

```
// 代码路径：mm/memory.c：
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
if (share_page(tmp))
    ...
}
```

因此，现在的情况和加载 shell 程序时的情况一样，都需要从外设上把程序加载进来。下面就要在主内存中申请空闲页面，然后加载 str1 程序了。

5.3.2 为 str1 程序申请一个内存页面

下面将在主内存中申请一个空闲页面，准备将 str1 最起始部分的程序载入，执行代码如下：

```
// 代码路径：mm/memory.c：
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
if (!(page = get_free_page()))
    ...
}
```

为 str1 程序申请的空闲页面及其在内存管理结构 mem_map 中登记的情况如图 5-21 所示：

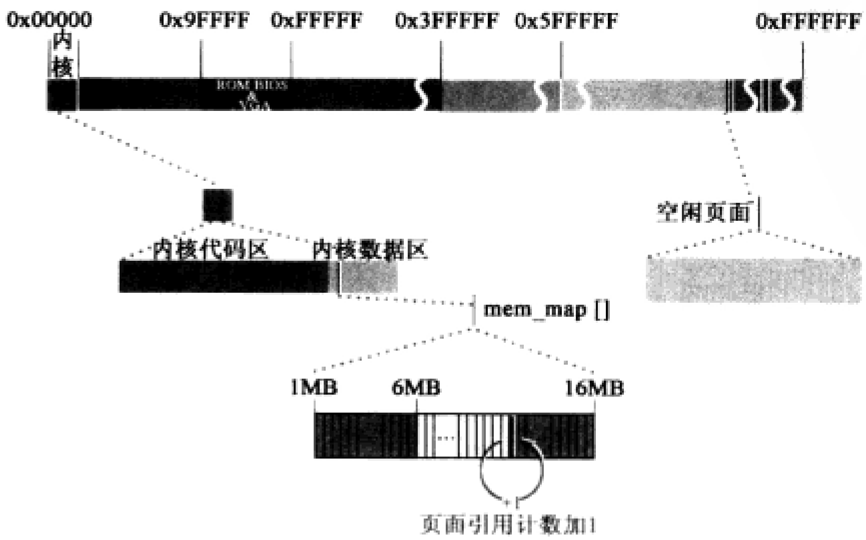


图 5-21 为 str1 程序申请一个内存页面

5.3.3 将 str1 程序加载到新分配的页面中

将 str1 程序从硬盘加载到这个新分配的页面中，一次加载 4KB 的内容，执行代码如下：

```
// 代码路径：mm/memory.c：
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
for (i=0 ; i<4 ; block++,i++)
        nr[i] = bmap(current->executable,block);
    bread_page(page,current->executable->i_dev,nr);
    ...
}
```

其中，bmap 函数在第 4 章 4.4 节已经详细讲解过，bread_page 函数的执行过程本质上与 bread 函数相同。这样，str1 程序中一个页面的内容就已经被加载到内存执行页面了。

过程如图 5-22 所示：

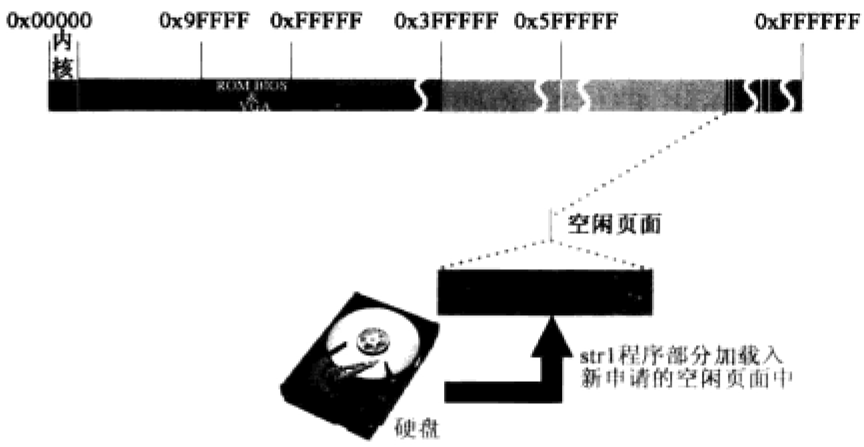


图 5-22 将 str1 程序的初始部分加载到新分配页面

5.3.4 检测是否需要对页面剩余空间清 0

接下来进行检测，如果程序较短，用不完一个页面的空间，则将剩下的空间清 0，执行代码如下：

```
// 代码路径：mm/memory.c：
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
i = tmp + 4096 - current->end_data;
    tmp = page + 4096;
    while (i-- > 0) {
        tmp--;
        *(char *)tmp = 0;
    }
    ...
}
```

我们现在假设 str1 的代码大于一个页面的内容，所以就用不着清 0 了。

5.3.5　将 str1 程序占用的物理内存地址与 str1 进程的线性地址空间对应

str1 程序载入后，也要将它对应的物理页面映射到指定的线性地址空间内，理由在 2.7.27 节中已经说明了，对应的代码如下：

```
//代码路径：mm/memory.c：
 void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
    if (put_page(page,address))
    ...
}
```

映射过程如图 5-23 所示，请读者注意，在此过程中，对页目录项进行了设置：

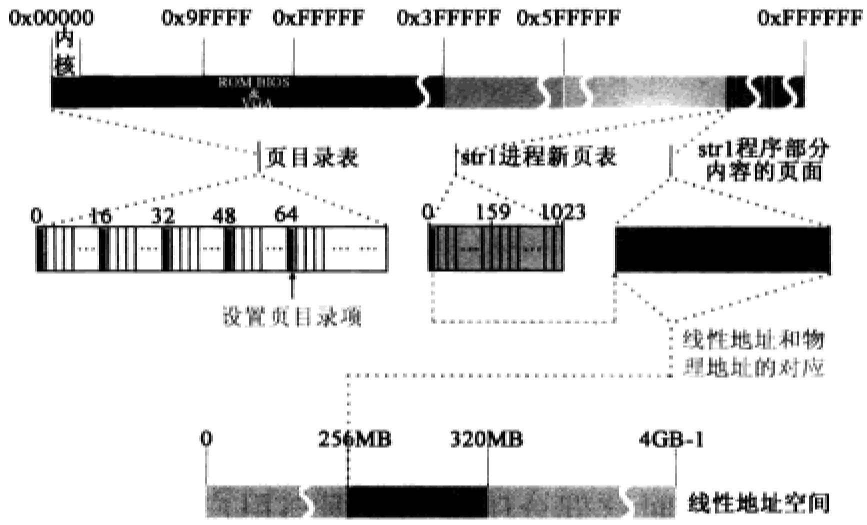


图 5-23　将 str1 程序的物理地址与线性地址对应

5.3.6　不断通过缺页中断加载 str1 程序的全部内容

5.3.4 节中已经介绍过，这个 str1 程序是大于一个页面的，所以，在执行过程中，如果需要新的代码，就会不断地产生缺页中断，以此来不断地加载需要执行的程序。

到此为止，str1 的加载过程我们就介绍完了，下面开始介绍 str1 运行的情况。

首先我们来介绍一下 str1 进程的源代码：

```
#include <stdio.h>
int foo(int n)
{
    char text[2048];
    if (n==0)
        return 0;
    else{
        int i = 0;
        for (i; i<2048; i++)
            text[i] = '\0';
        printf("text_%d = 0x%x, Pid = %d\n", n, text, getpid());
        sleep(5);
        foo(n-1);
    }
```

```
}
int main(int argc, char **argv)
{
    foo(6);
    return 0;
}
```

5.3.7 str1 程序需要压栈

程序开始运行，压栈动作产生。

我们观察一下 str1 进程运行的状况。str1 程序中的 foo 函数被递归调用了，通过在 foo 函数中使用一个局部的大小为 2048 的字符数组 text（为了能够让缺页中断的效果更快地表现出来，所以我们这里设置的数组大小为 2048 字节，这样，两次压栈，效果就出来了），以使 str1 进程的栈空间以较快的速度增长。每调用一次 foo 函数，str1 进程栈（esp）就向下增长 2048 个字节。

5.3.8 str1 程序第一次调用 foo 程序压栈

第一次调用 foo 函数，esp 向下扩充了 2048 字节。在扩充之前，esp 所指向的用户进程参数及环境变量列表已经在页面中占据了一定的空间，此时扩充 2048 字节，再加上原来参数列表所占用的这部分空间，仍然没有超出一个页面的总空间 4KB，所以，目前它尚可以与列表共存于同一物理页面中。情况如图 5-24 所示，其中图 5-24 中右下部分所示页面中的绿色部分代表已压栈的参数和环境变量数据，可见这时这部分压栈的数据与用户进程参数和环境变量列表在同一个页面中。

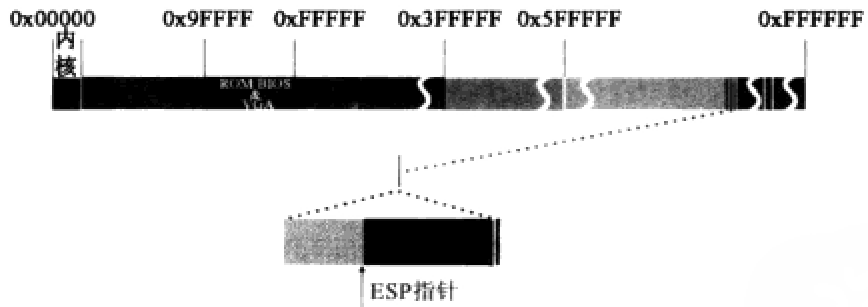


图 5-24 str1 程序第一次压栈

5.3.9 str1 程序第二次压栈，产生缺页中断

第二次调用 foo 函数时，情况就不同了。加上第一次扩充的 2048 字节，此时这个页面已经无法继续承载 2048 字节的内容了，这就意味着需要新的物理页面空间，所以就会产生缺页中断，准备申请新的页面。

5.3.10 处理 str1 程序第二次压栈产生的缺页中断

新的物理页面仍然会映射到线性地址空间内，以此支持寻址，这次缺页中断仍然是进入 do_ no_page 函数去执行，但是执行的代码不同了，接下来会执行如下的代码：

```
// 代码路径：mm/memory.c：
void do_no_page(unsigned long error_code,unsigned long address)
{
    ...
address &= 0xfffff000;
    tmp = address - current->start_code;
    if (!current->executable || tmp >= current->end_data) {
        get_empty_page(address);
        return;
    }
    ...
}
```

这是因为 tmp >= current->end_data 这个条件成立了，触发中断的线性地址已经大于进程的 end_data 地址。因此，接下来调用 get_empty_page 函数去执行，这次并不会将任何外设的数据加载入页面。我们在第 2 章 2.7.23 节中介绍过，压栈也会产生缺页中断，但跟外设上的数据毫无关系，现在这一点已经得到了体现。

进入 get_empty_page 函数后，就会新申请缺少的页面，并将这个物理页面映射到 str1 的线性地址空间内，执行代码如下：

```
// 代码路径：mm/memory.c：
void get_empty_page(unsigned long address)
{
    unsigned long tmp;

    if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
        free_page(tmp);          /* 0 is ok - ignored */
        oom();
    }
}
```

通过对这个函数的调用，我们可以进一步把缺页中断的本质看清楚：缺页中断是否会产生完全取决于“主机中进行的运算是否需要内存中的新页面来支持”，只要需要新的页面做支持，就会产生缺页中断，至于这个新页面中的内容究竟是来自外设，还是产生于程序自身的数据压栈，主机完全不关心。这些数据的来源并不是缺页中断产生的决定条件。

5.3.11　str1 程序继续执行，反复压栈并产生缺页中断

程序继续执行，如此反复进行“压栈→缺页中断→分配物理内存→压栈……”操作，当第 n 次调用 foo 函数时，用户进程栈与物理内存的映射关系如图 5-25 所示，请大家注意压栈数据在内存页面上的变化情况。

5.3.12　str1 程序运行结束后清栈

程序执行完毕了，foo 函数到达递归的终止点返回（if (n==0) return 0），这时由于函数返回导致进程清栈，esp 向上（高地址）收缩，用户进程实际使用的栈空间就变小了。既然栈已经变

小，那么之前映射到栈的线性地址空间的物理页面就应该释放掉，但我们从代码分析和测试中都发现，Linux 0.11 内核并没有释放该物理页面，结果导致内存浪费。

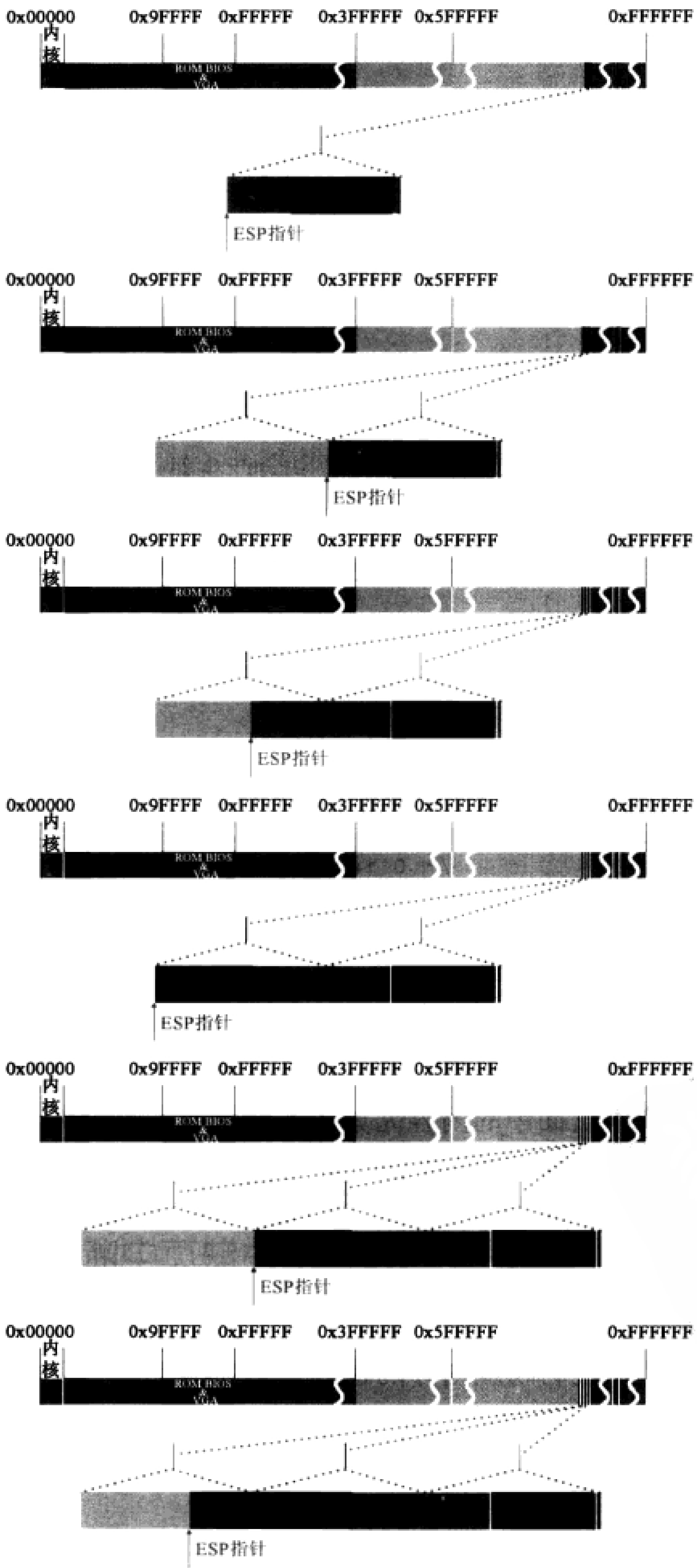


图 5-25 str1 程序不断压栈的效果

清栈效果如图 5-26 所示：

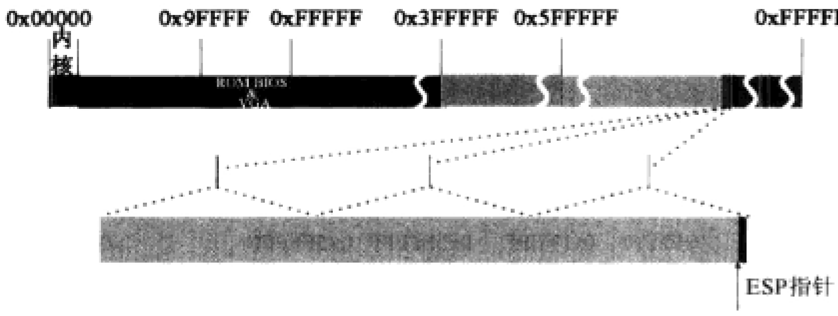


图 5-26 str1 程序运行结束后清栈效果示意图

5.4 str1 用户进程的退出

本节将详细介绍关于 str1 这个用户进程的最后一部分内容，即它是如何退出的。其中包括：它在运行过程中占据的内存空间如何释放，它自身管理结构所占据的空间如何处理，等等。我们在 2.8 节中讲到了 shell 进程的退出，其实 str1 的退出与 shell 进程的退出步骤是一样的，都是通过调用 exit 函数来实现的，善后事务处理完毕后，由它们的父进程负责将它们自身的进程管理结构所在的页面释放掉。下面我们来具体介绍这一过程。

5.4.1 str1 进程准备退出

str1 进程调用 exit() 函数进行退出。最终会映射到 sys_exit 函数去执行，并调用 do_exit 函数来处理 str1 进程退出的相关事务。退出必须注销的内容包括两个方面：第一，释放 str1 进程代码与数据所占用的物理内存并解除其与 str1 这个可执行文件的关系，这一点由 str1 进程自己负责；第二，释放 str1 进程的管理结构 task_struct 所占用的物理内存并解除与进程槽的关系，这一点由 shell 进程负责。

```
//代码路径：include/unistd.h：
volatile void exit(int status);

//代码路径：kernel/exit.c：
int sys_exit(int error_code)
{
    return do_exit((error_code&0xff)<<8);
}
```

5.4.2 释放 str1 程序所占页面

进入 do_exit 函数后，调用 free_page_tables 函数将 str1 程序占用的页面释放掉，并将管理这些页面的页表及页目录项释放掉，执行代码如下：

```
//代码路径：kernel/exit.c：
int do_exit(long code)
{
```

```
    ...
    free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
    free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    ...
}
```

释放过程如图 5-27 所示：

图 5-27　释放 str1 程序占用的页面

值得注意的是，str1 进程退出后，它的程序所占用的页面只能释放掉，具体表现为将引用次数减 1。这是因为，Linux 0.11 是一个支持多进程运行的操作系统，这些页面很有可能正在被其他进程共享着。如果将这些页面全部清空，其他共享的进程也就无法使用这些信息了。也正是因为共享页面这种机制的存在，才有了“页面引用计数”这个概念，如果没有共享情况存在，则页面的使用就只有两种状态：要么“已占用”，要么“未占用”。

但是，其他进程是不可能与 str1 进程共享一个“页表”的，更不可能共享一个“页目录项”，这些都是 str1 进程所独有的东西，所以页目录项一定会被清除，而页表所在内存页面的引用次数一定是“1”，释放后也就一定是“0”，即该页面为空闲。

5.4.3　解除 str1 程序与文件有关的内容并给父进程发信号

解除该进程与 str1 这个可执行文件的关系，具体表现为，先将对应的文件 i 节点释放，然后将它自己设置为僵死状态，并给它的父进程 shell 发送“子进程退出”信号，具体执行步骤在 2.8 节中已经说明，对应代码如下：

```
//代码路径：kernel/exit.c：
int do_exit(long code)
{
    ...
    for (i=0 ; i<NR_OPEN ; i++)
        if (current->filp[i])
            sys_close(i);
    iput(current->pwd);
    current->pwd=NULL;
    iput(current->root);
    current->root=NULL;
    iput(current->executable);
    current->executable=NULL;
    if (current->leader && current->tty >= 0)
        tty_table[current->tty].pgrp = 0;
    if (last_task_used_math == current)
        last_task_used_math = NULL;
    if (current->leader)
        kill_session();
```

```
        current->state = TASK_ZOMBIE;
        current->exit_code = code;
    tell_father(current->father);
        ...
    }
```

关系解除并给父进程发信号的过程如图 5-28 所示：

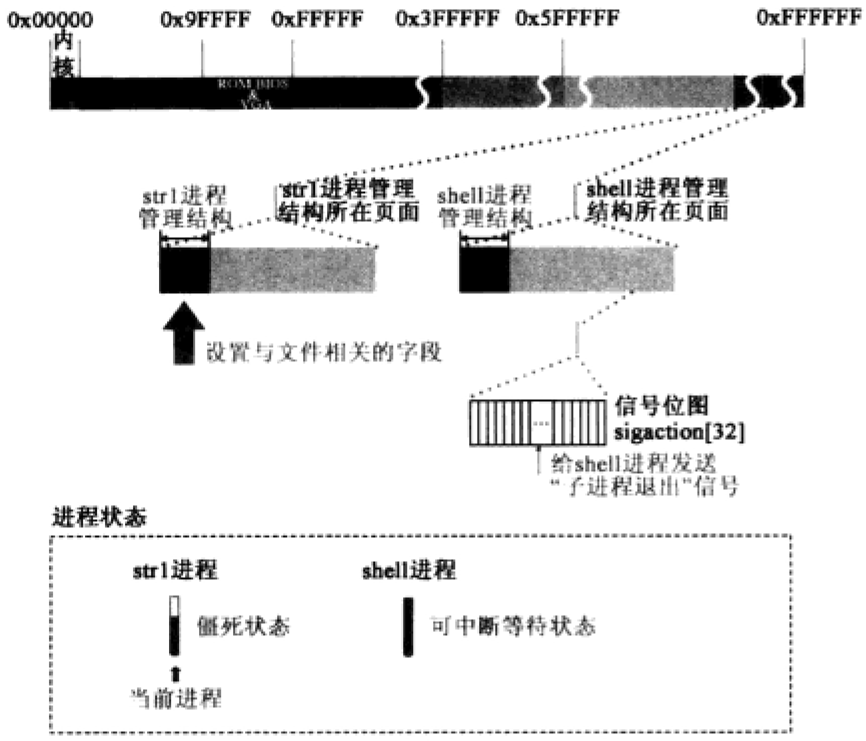


图 5-28　解除 str1 程序与文件有关的内容

值得注意的是，这里解除 str1 进程与文件的关系，与解除它的执行程序与页面的关系所进行的操作非常类似，都是将引用计数减 1，比如 sys_close 这个函数，就是要将文件管理表 file_table 中与 str1 进程相关的文件引用计数减 1，因为其他进程还有可能使用这些文件，等等。但是，从 current->pwd=NULL; 和 current->root=NULL; 这两行代码可以看出，pwd 和 root 这两个成员都是 str1 进程私有的信息，绝对不会有其他进程与之共享，所以就将它们设置为 NULL。

通过 5.4.2 节和 5.4.3 节的分析，我们可以得出一个结论：在系统中，对于需要在多个进程间共享的资源，都只能采取引用计数来记录其使用情况，否则就会影响到其他进程的使用；对于每个进程独有的资源，比如进程管理结构中的成员，或者进程程序的页目录项等，要么使用，要么清除。这一切都是由于 Linux 0.11 是一个支持多进程运行的操作系统导致的，这些使用资源的规则都是为了适应多进程运行而产生的。

5.4.4　str1 程序退出后执行进程调度

到此为止，str1 对退出操作所做的善后工作就完毕了，str1 进程将切换到其他进程去执行。由于现在只创建了一个用户进程，所以现在系统中有进程 0、进程 1、update 进程、shell 进程和 str1 这个用户进程，执行代码如下：

```
// 代码路径：kernel/exit.c：
int do_exit(long code)
{
    ...
    schedule();
    ...
}
```

进程切换效果如图 5-29 所示：

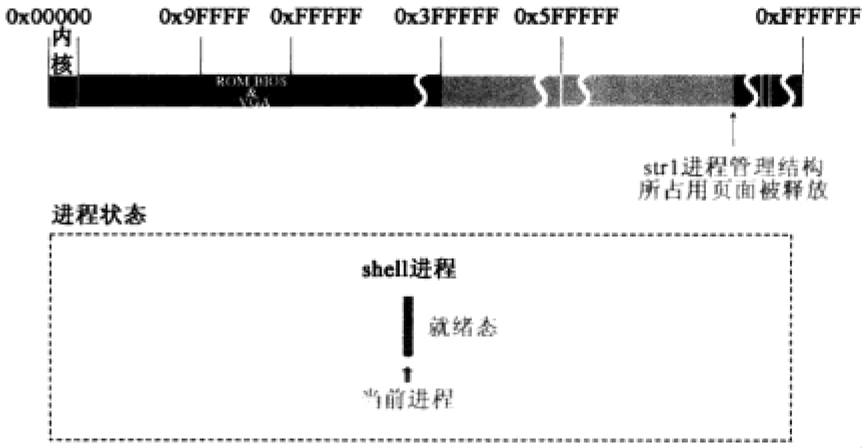


图 5-29　str1 进程退出，切换到 shell 进程

这时候，shell 进程由于接收到 str1 发送的信号而被唤醒，于是就切换到 shell 进程去执行，shell 进程将释放掉 str1 进程的管理结构所占用的页面，并解除 str1 进程与进程槽 task 的关系，这样 str1 就彻底从系统中退出了。

到此为止，本章第一部分的内容就全部介绍完了。通过以上 4 节内容可以看出，在 Linux 0.11 中，用户进程的创建和程序的加载、运行，以及用户进程的退出，与第 2 章所讲述的进程的创建和程序的加载、运行，以及进程退出的执行过程大体是一致的。但这一切仅仅是针对一个用户进程来介绍的，Linux 0.11 是一个多进程的操作系统，这一点主要就表现为能够支持多个用户进程“同时”运行，下面我们就将对多个用户进程“同时”运行的情况进行介绍。

5.5　多个用户进程“同时”运行

本节我们以三个用户进程 str1、str2、str3 为例，来看看多个进程在系统中是如何运行的。它们是否真的像看上去那样，都在“同时”运行；它们在物理内存空间和线性地址空间的连续性是否一致。

5.5.1　依次创建 str1、str2 和 str3 进程

我们假设系统中尚没有任何用户进程存在，外设上有三个可执行文件，分别是 str1、str2、str3，且文件中的程序都与 5.3 节中介绍的 str1 代码一致。在此前提下，我们依次创建三个用户进程：str1、str2、str3。

通过 2.8.9 节中对 shell 进程的介绍得知，现在 last_pid 已经累加为 4 了，所以这三个进程

的进程号依次应该是：5、6、7。现在进程槽 task 中的前 4 项已经被占用了，所以这三个进程只能从第 5 项开始依次与进程槽建立关系，它们在进程槽中的项号依次分别为 4、5、6。由此可进一步得出，它们在线性地址空间的位置依次应该是：4×64MB 至 5×64MB、5×64MB 至 6×64MB、6×64MB 至 7×64MB，理由已经在 2.2.8 节中说明。

这三个进程在线性空间中的分布效果如图 5-30 所示：

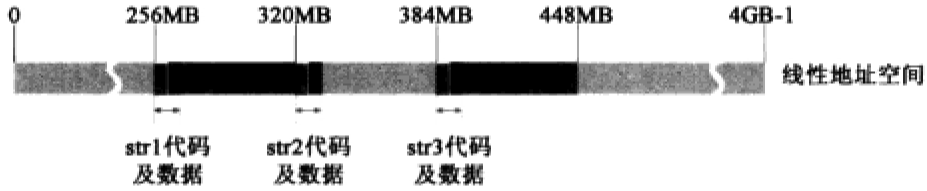


图 5-30　str1、str2 和 str3 进程在线性空间中的分布

图 5-31 展现了三个进程的管理结构和压栈的数据信息在物理内存中的分布情况。接下来，我们将分析这种内存页面在主内存的分布格局是如何产生的。

图 5-31　三个进程的管理结构和压栈数据信息在主内存区的分布

5.5.2　str1 进程压栈的执行效果

假设现在轮到 str1 进程执行，str1 开始执行 foo 函数调用，就需要压栈，于是产生缺页中断。在缺页中断处理中，内核为 str1 进程申请了空闲物理页面，并将其映射到 str1 进程的线性地址空间。之后，进程再对 text 数组进行设置，内容就被写到刚分配的物理页面上了。

执行效果如图 5-32 所示：

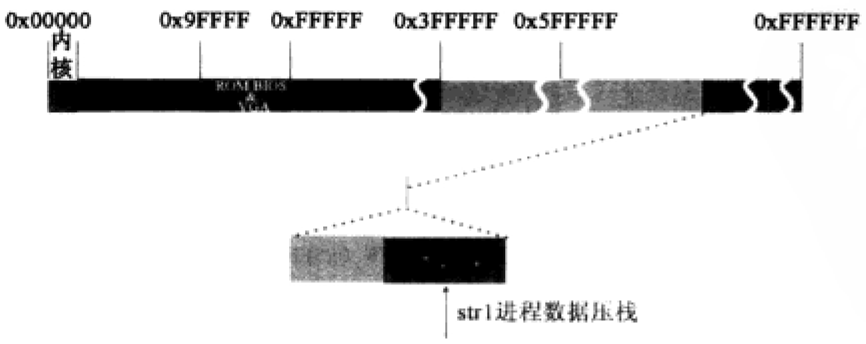


图 5-32　str1 进程压栈的效果

5.5.3　str1 运行过程中产生时钟中断并切换到 str2 执行

在执行过程中，str1 每 10 毫秒就会产生一次时钟中断，这样就会削减它的时间片（理由在 2.2.5 节中已经说明），由于程序中调用了 sleep 函数，这个函数可以起到延时的效果，所以当它

的时间片被削减为 0 时，程序仍然没有执行完，这时候我们来看执行的代码：

```
// 代码路径：kernel/sched.c：
void do_timer(long cpl)
{
    ...
    if ((--current->counter)>0) return;
    current->counter=0;
    if (!cpl) return;
    schedule();
}
```

很显然，要执行 if (!cpl) return; 这一行，cpl 这个参数只有两种可能的值：一个是“0”，表示现在处于内核态；另一个是“3”，表示现在处于用户态。对于此时 str1 的情况来说，它一直在执行用户程序，肯定是用户态，所以就要继续执行，调用 schedule 函数，准备进程切换。

于是切换至进程 str2 执行，进程 str2 也执行同样逻辑的程序。值得注意的是，当设置 text 数组时，屏幕打印的逻辑地址与当时进程 str1 的地址相同。但是，通过 5.5.1 节的介绍我们得知，它们的线性地址是不同的，可见在物理内存中进程 str2 并没有与 str1 重叠。

str2 执行压栈操作的效果如图 5-33 所示：

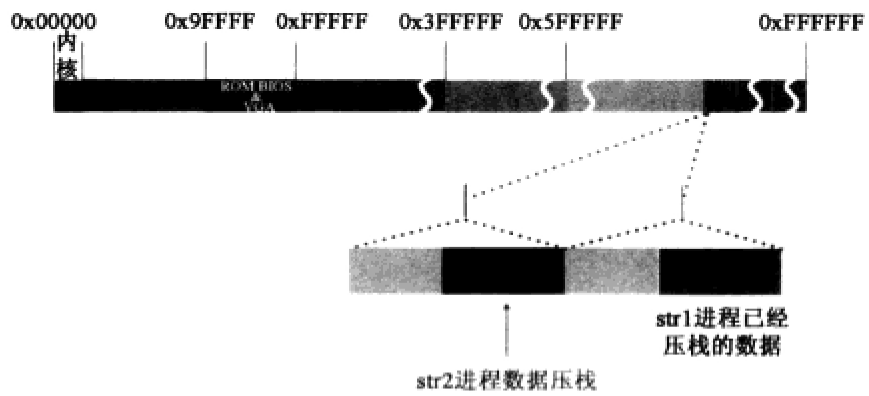


图 5-33 str2 执行压栈操作的效果

5.5.4 str2 执行过程遇到时钟中断切换到 str3 执行

str2 执行一段时间后，时间片被削减为 0 时又切换到 str3 去执行，它也要压栈，str3 开始运行。执行的代码与进程 str2 相同，也是压栈，并设置 text。

str3 程序执行压栈的效果如图 5-34 所示：

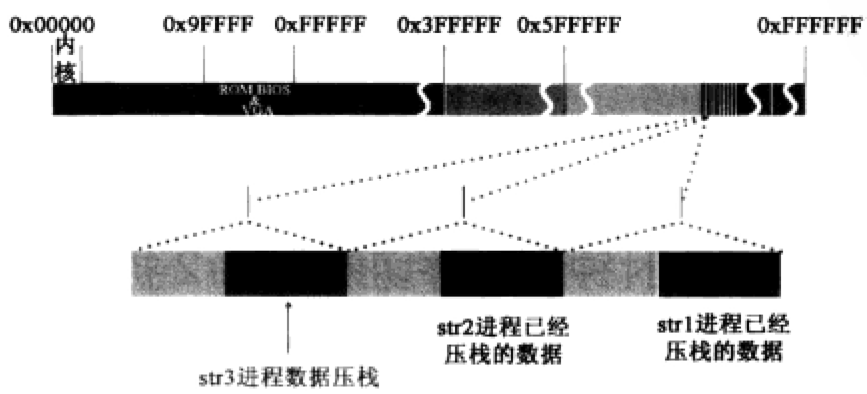


图 5-34 str3 执行压栈操作的效果

5.5.5 三个程序执行一段时间后在主内存的分布格局

str3 执行一段时间后，时间片也用完了，这样三个用户进程虽然还需要继续执行，但时间片都用完了。如果此时再次调用 schedule 函数进行进程切换，系统就会为它们重新分配时间片，开始下一轮的执行与切换，接着它们都会继续不断地压栈，通过图 5-35 我们可以看出这一点。这三个用户进程在线性地址空间内压入到它们各自的栈中的数据都是连续的，但在物理空间内压栈的数据却是完全“交错”分布的。

三个程序执行一段时间后，这三个程序压入到它们各自的栈中的数据在主内存区中的分布如图 5-35 所示：

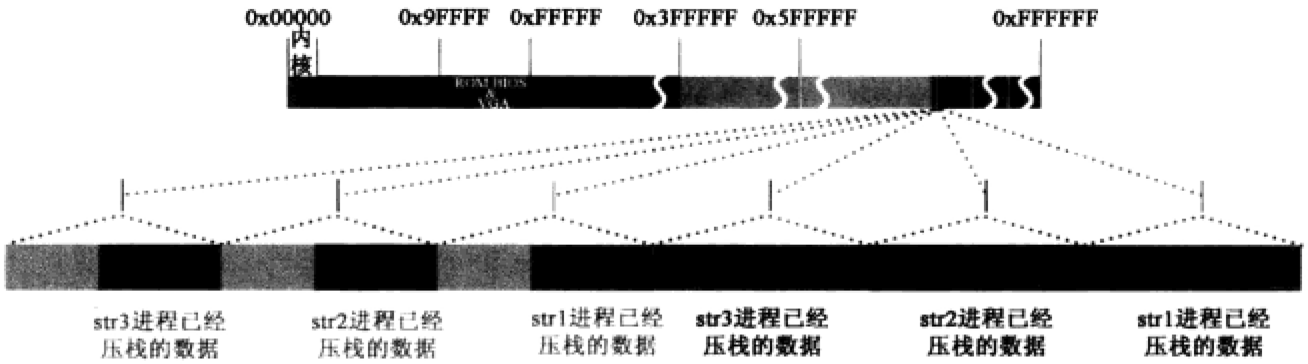


图 5-35 三个程序执行一段时间后压栈数据在主内存区的分布情况

这一节我们介绍了多个用户进程“同时执行”时内存中的数据分布情况。可以看出，任何时刻，都只有一个进程在执行，根本不存在多个进程同时执行的情况。一个进程执行完毕后，将通过调用 shedule 函数来进行切换，用户进程间切换的具体规则将在下一节详细介绍。

5.6 进程的调度与切换

本节我们仍然以用户进程 str1 和 str2 的运行为例详细讲述用户进程是如何被调度和切换的，它们对应的代码还是 5.3 节中所描述的代码。我们已经得知，在任何条件下，同一时刻都只能有一个进程在运行，那么系统又是根据什么条件来决定当前应该让哪个进程运行，又是根据什么来判断是否应该让进程进行切换的呢？

5.6.1 str1 刚被 shell 创建并处于就绪态

假设现在系统中只有一个用户进程 str1，而且当前进程是 shell 进程。此时 shell 进程刚刚创建完 str1，shell 进程的状态为“就绪状态”，str1 也为就绪状态。

此时，处于执行状态的是 shell 进程，str1 虽然处于就绪状态，但是还没有被调度执行。效果如图 5-36 所示。

5.6.2 shell 进程将自己挂起，然后准备切换到 str1 执行

正常情况下，shell 在创建完新进程后就会进入 TASK_INTERRUPTIBLE 状态（原因在第 2

章 2.8 节中已经说明），等待用户再次输入命令。也就是说，此时用户没有敲键盘，shell 进程会调用 schedule 函数进行切换。此时还没有轮到进程 str1 运行，它的时间片为 15，优先级也是 15。效果如图 5-37 所示：

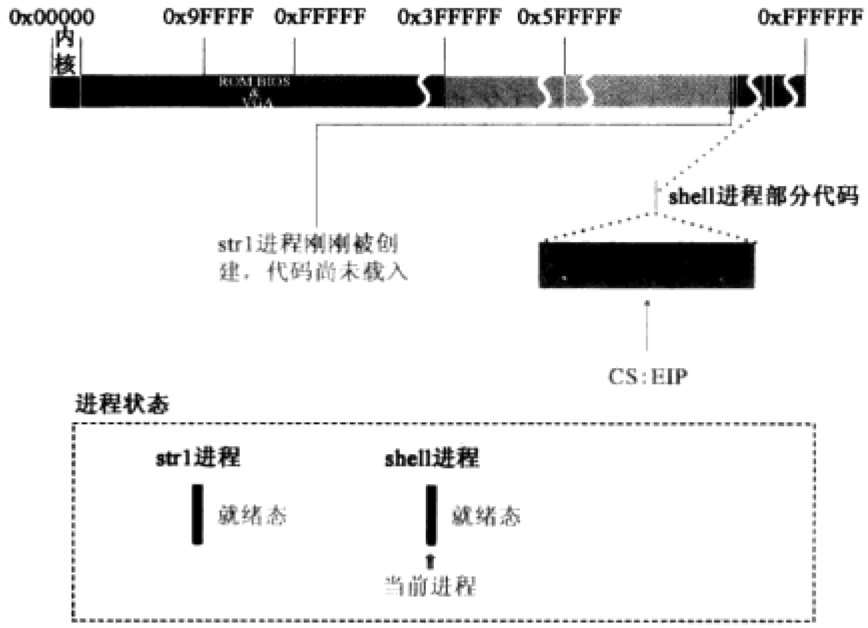


图 5-36 str1 刚被创建完成时的状态

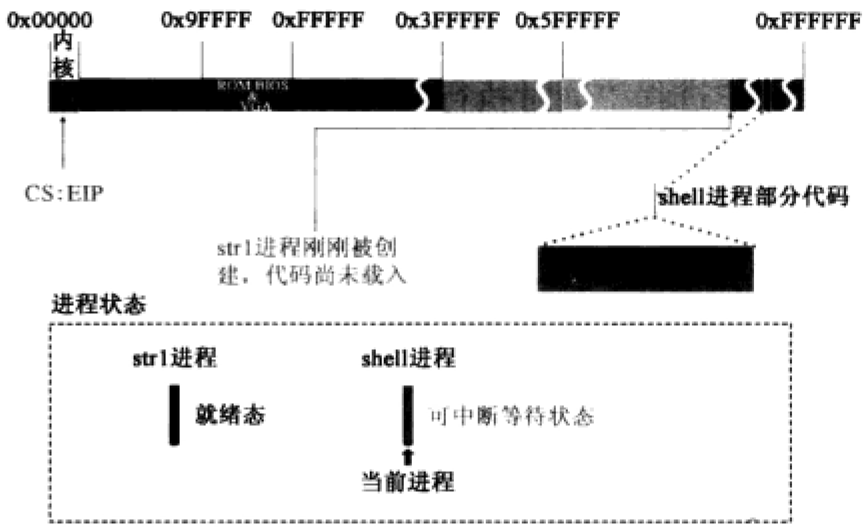


图 5-37 shell 进程设置为可中断等待状态后挂起

5.6.3 准备切换到 str1 进程执行

进入 schedule 函数后，从进程槽 task[64] 数组的末端开始扫描当前系统中已存在的进程（进程 0 除外）是否启用了报警器（alarm），该报警器的意思当到达某个时间点时内核给进程发送一个报警信号（SIGALRM）。内核可以检测报警器的数值，唤醒进程。用户进程调用 sleep 就是通过设置报警器，使自己可以在一定时间后被内核唤醒运行。

执行代码如下：

```
//代码路径：kernel/sched.c：
void schedule(void)
{
      ...
```

```
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p) {
                    if ((*p)->alarm && (*p)->alarm < jiffies) {
                                (*p)->signal |= (1<<(SIGALRM-1));
                                (*p)->alarm = 0;
                                }
    ...
}
```

之后开始第二次遍历，内核通过 while 循环来选择一个合适的进程投入运行。从 task 数组的末端开始遍历每个处于就绪状态且当前剩余时间片（counter）最大的进程。很显然，现在是 str1，内核准备将该进程投入运行。

执行代码如下：

```
// 代码路径：kernel/sched.c：
void schedule(void)
{
    ...
        while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                    (*p)->priority;
    }
    switch_to(next);
    ...
}
```

最后执行 switch_to 实施进程切换，switch_to 使 CPU 切换到新的进程执行。内核会先执行"cmpl %%ecx,_current"这行语句，判断新的进程是否就是当前进程，如果是，直接返回。如果不是，则执行"movw %%dx,%1\n\t"，将新进程 TSS 选择符存入临时结构变量 struct {long a, b} __tmp 中的 b 中。然后将系统当前进程指针 current 修改为新进程，ecx 存放原进程的 task_struct 指针。执行 ljmp 指令将致使 CPU 各个寄存器的状态回填到原进程的 tss 中，再将新进程（此时为 str1 进程）的 tss 中所有寄存器的值载到 CPU 的各个寄存器中（这是由硬件实现的）。对应代码如下：

```
// 代码路径：include/sched.h：
    #define switch_to(n) {\
struct {long a,b;} __tmp; \
__asm__("cmpl %%ecx,_current\n\t" \
    "je 1f\n\t" \
```

```
        "movw %%dx,%1\n\t" \
        "xchgl %%ecx,_current\n\t" \
        "ljmp %0\n\t" \
        "cmpl %%ecx,_last_task_used_math\n\t" \
        "jne 1f\n\t" \
        "clts\n" \
        "1:" \
        ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
        "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

至此任务切换完成。

执行效果如图 5-38 所示，请读者注意，此时运行的进程变为了 str1，而 shell 进程仍处在可中断等待状态。

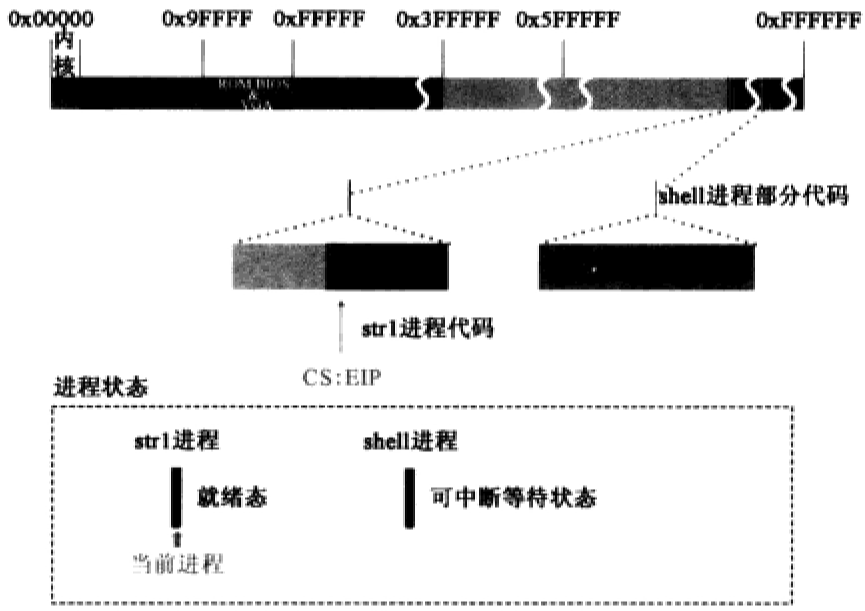


图 5-38　系统切换到 str1 进程执行

5.6.4　str1 执行时发生时钟中断

现在切换到 str1 去执行，我们假设它刚刚开始执行就发生了时钟中断，效果如图 5-39 所示：

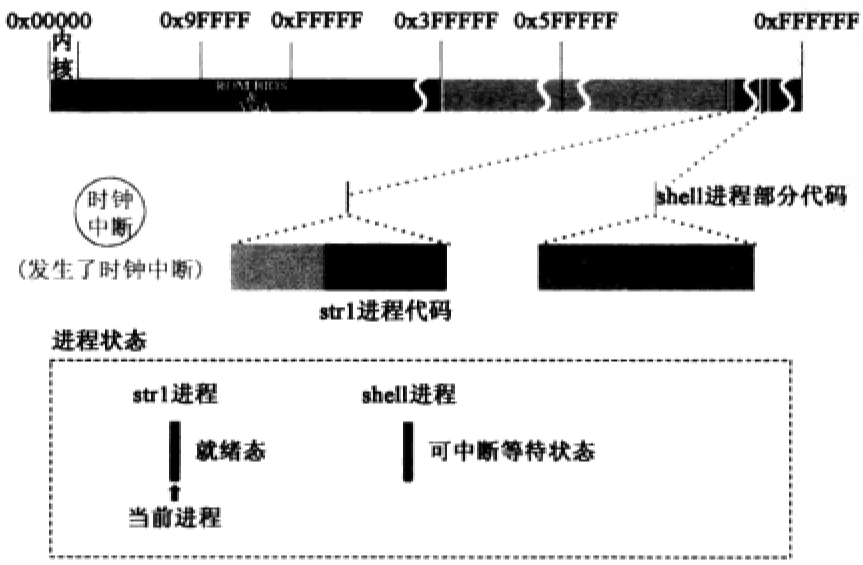


图 5-39　str1 运行时发生时钟中断

5.6.5 时钟中断递减 str1 运行的时间片

在时钟中断处理函数 do_timer 中，内核会递减当前进程的时间片，每次减 1，但只有当时间片递减到 0 时，才有可能从当前进程切换到其他进程。str1 的时间片从 15 递减为 14。当中断返回后，继续运行。

执行代码如下：

```
// 代码路径：kernel/sched.c：
void do_timer(long cpl)
{
    ...
    if ((--current->counter)>0) return;
    ...
}
```

系统响应时钟中断后对 str1 进程产生的影响如图 5-40 所示，请大家注意图中左边代表 str1 时间片的竖条的变化，时间片减少了。

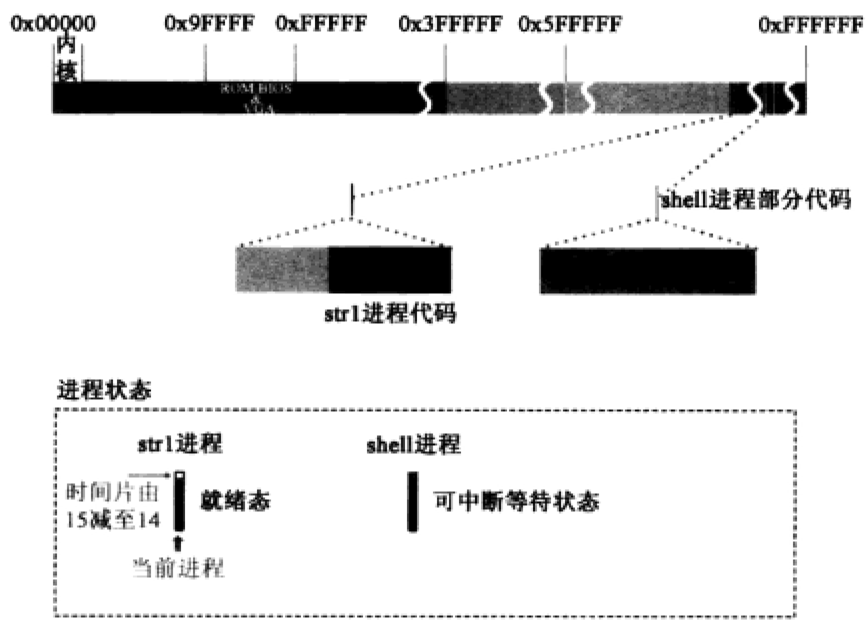


图 5-40 str1 进程的时间片减 1

5.6.6 str1 执行一段时间后挂起，shell 进程新建 str2 进程

在进程 str1 执行一段时间后，会调用 sleep() 进入 TASK_INTERRUPTIBLE 状态，但它的时间片没有用完，假设 counter = 10。如图 5-41 所示：

如果此时用户在 shell 中输入 ./str2 命令，shell 进程就会创建 str2 这个新的用户进程。在 shell 刚新建完 str2 进程时，shell 进程、str1 进程和 str2 进程的状态如图 5-42 所示：

之后，shell 再次进入睡眠状态，并调用 schedule 进行一次任务的切换，此时系统中只有 str2 处于 TASK_RUNNING 状态，因此进程 str2 投入运行。

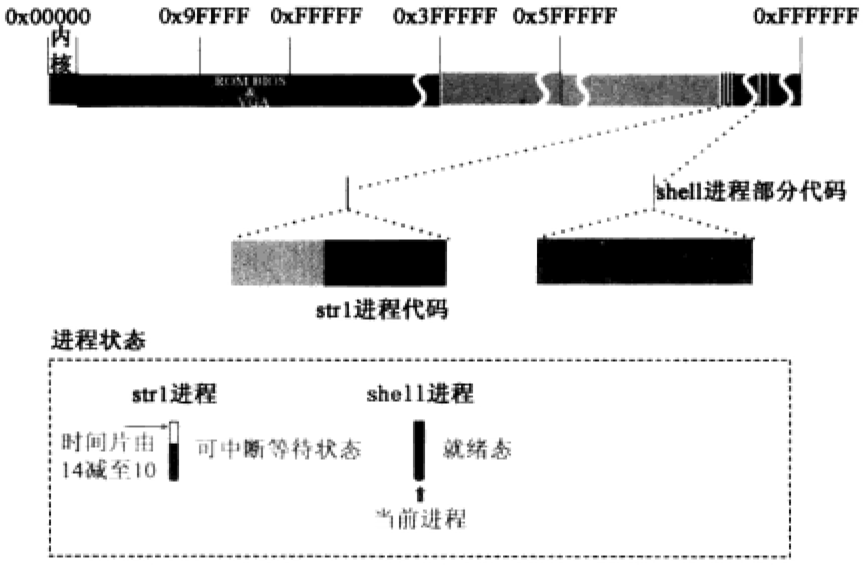


图 5-41 str1 切换时的状态

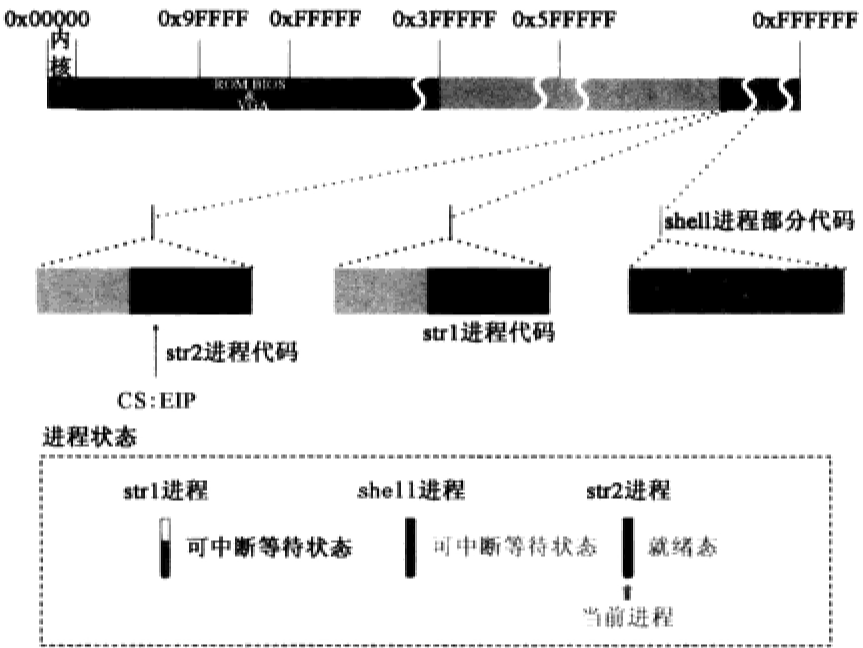


图 5-42 str2 进程刚创建完成时三个进程的状态

5.6.7 str2 运行期间发生时钟中断

进程 str2 在运行期间又发生了时钟中断，中断处理程序检查当前的时间是否超过了 str1 睡眠时间后的时间⊖。如果超过了，则将进程 str1 的信号位设置成 SIGALRM，并置 str1 的进程状态为 TASK_RUNNING（这一点我们在 5.6.3 节中已经说明）。此时，str1 的时间片仍然为 10。同时，当前进程 str2 的时间片会被削减，然后进程 str2 继续执行。str2 的程序代码和 str1 的程序代码是相同的，所以，也会被设置为可中断等待状态，于是中断处理调用 schedule 进行任务的切换。我们假设切换时进程 2 的时间片也被削减为 10 了。

图 5-43 显示了 str2 的时间片递减并被置为可中断等待状态。

⊖ 比如 str1 从 1:00:00 开始睡眠，睡眠时间为 5 分钟，则它应该在 1:05:00 发信号。现在要查看的是当前时间是否超过了 1:05:00，即睡眠时间以后的时间。

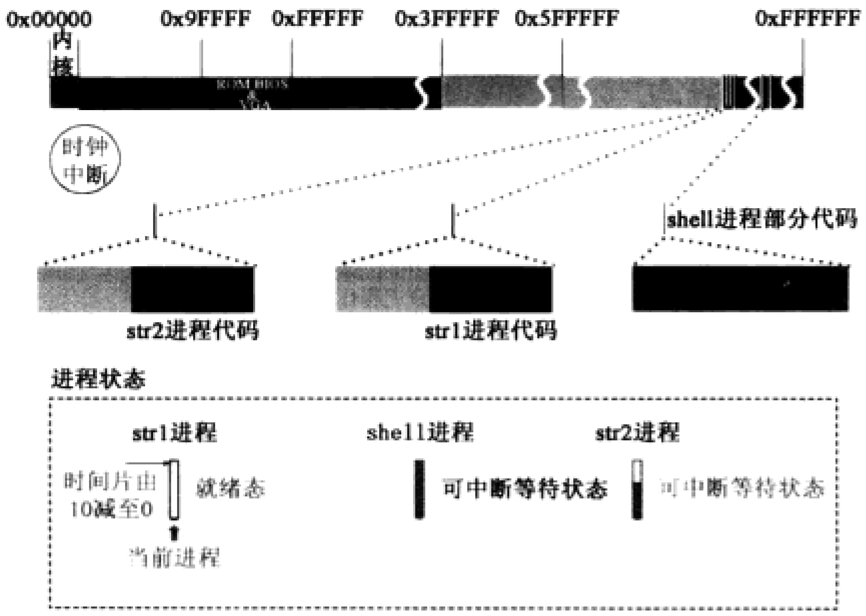


图 5-43　str2 运行器件发生时钟中断

5.6.8　系统切换到 str1 程序执行

由于 str1 此时为就绪态，所以会切换到 str1，等到 str1 时间片削减为 0 后，系统会重新分配时间片，让它们继续轮流操作，如图 5-43 所示。

由于在 5.6.7 节中 str2 的状态已经是可中断等待状态，shell 进程也处在可中断等待状态，只有 str1 进程既有时间片，又处在 TASK_RUNNING 状态，我们可以判断出 schedule 此时会将进程 str1 投入运行。str1 运行，直到时间片耗尽，然后发生时钟中断并执行 schedule 函数。但是，此时 str1 时间片为 0，其他进程又都不是就绪态，该重新分配时间片了。

内核从 task 数组的末端开始重新给当前系统的所有进程（包括处于睡眠的进程，但进程 0 除外）分配时间片。时间片的大小为：counter/2 + priority。 priority 是进程的优先级，所以进程的优先级越高（priority 值越大），分配到的时间片就越多。然后根据此时时间片的情况重新选择进程运行，如此反复进行。

执行代码如下：

```
// 代码路径：kernel/sched.c：
void schedule(void)
{
    ...
                    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                          if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;

    ...
}
```

重新分配时间片的效果如图 5-44 所示：

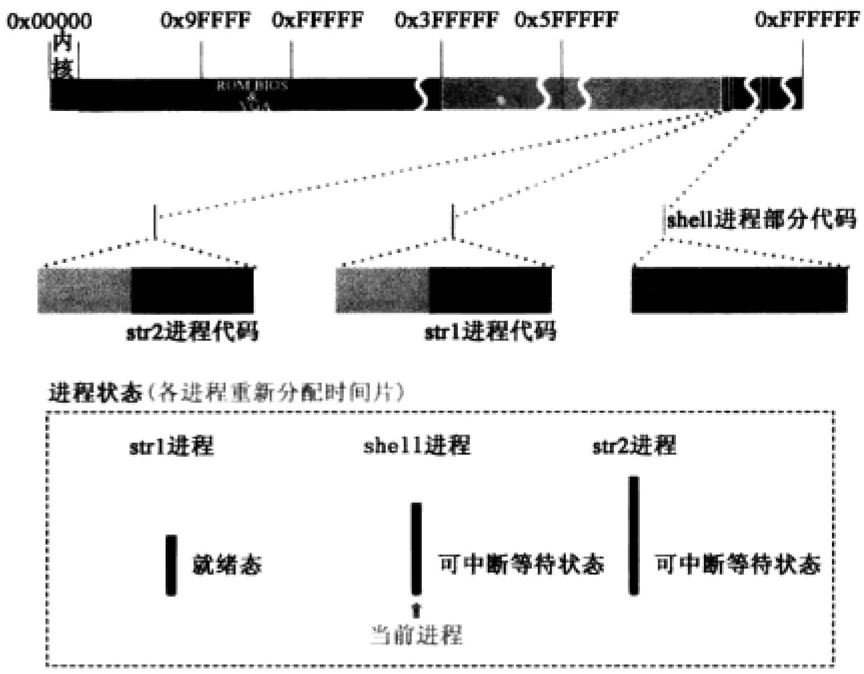


图 5-44　为所有进程重新分配时间片

这里值得注意的是，重新分配时间片时，并不需要给进程 0 分配。这是因为，只要系统中所有进程都暂时不具备执行条件，那么就自动切换到进程 0 去执行，进程 0 将一直执行下去，即便在此过程中它的时间片削减为 0 了，但由于还没有可以运行的进程，所以仍然需要进程 0 继续执行，这样一来，时间片对进程 0 来讲就没有意义了。可见，进程 0 是一个特殊的进程，它的执行是由系统当前的需求来决定的，时间片轮转这套机制并不适用于它。

5.7　内核的分页

本节将通过对 head.s 文件中代码的分析来阐述内核分页的意义，以此来讲述为什么内核能够随心所欲地访问用户进程所在的空间，而用户进程却不能访问到内核所在的空间。

5.7.1　为设置内核的页目录表和页表做准备——所占空间清 0

head.s 中页目录表的前 4 项和后继的 4 个页表都是为内核寻址服务的。每个页表可以访问和管理 4MB 内存，这样内核就可以访问 16MB 内存了。这 4 个页表管理的区域正好是 Linux 0.11 能够支持的最大物理内存空间——16MB。任何一个进程的线性地址最终都要映射到这 16MB 的物理内存中，所以内核能够访全部 16MB 的进程空间，这就为它能够访问到用户进程所在的空间提供了理论上的可行性。

首先，需要将内核的页目录表和页表所占的空间清 0，对应代码如下：

```
// 代码路径：boot/head.s：
   ...
198 setup_paging:
199         movl $1024*5,%ecx
200         xorl %eax,%eax
```

```
        xorl %edi,%edi
        cld;rep;stosl
  ...
```

清 0 的效果如图 5-45 所示：

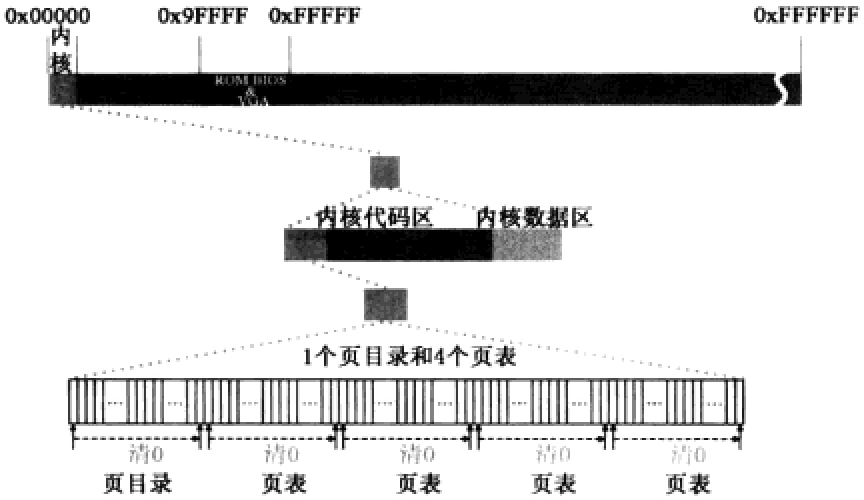


图 5-45 设置内核的页目录表和页表

5.7.2 设置内核对应的页目录项和页表项的内容

我们再来观察一下对这些页目录项和页表的设置。紧接下来的一段代码先设置页目录表中前 4 项的页面地址，内容分别为 0x1000、0x2000、0x3000、0x4000；然后再设置页表，每个页表的各个页表项中都存储着其所对应页面的地址，页表项从第 4 个页表的最后一项（对应着物理内存的 16MB-4KB 处）开始由高到低设置，其中的页面地址内容依次被设置为 0xfff000.0xffe000、0xffd000……0x3000、0x2000、0x1000。通过这些设置可以看出，内核的线性地址的页首地址与物理页框的首地址刚好吻合。

执行代码如下：

```
// 代码路径：boot/head.s：
...
        movl $pg0+7,_pg_dir
        movl $pg1+7,_pg_dir+4
        movl $pg2+7,_pg_dir+8
        movl $pg3+7,_pg_dir+12
        movl $pg3+4092,%edi
        movl $0xfff007,%eax
        std
    1:  stosl
        subl $0x1000,%eax
        jge 1b
...
```

执行完成后的效果如图 5-46 所示，其中页目录表中的每一项对应一个页表，4 个页表项也均被设置，4 个页表正好可以管理 16MB 的内存空间。

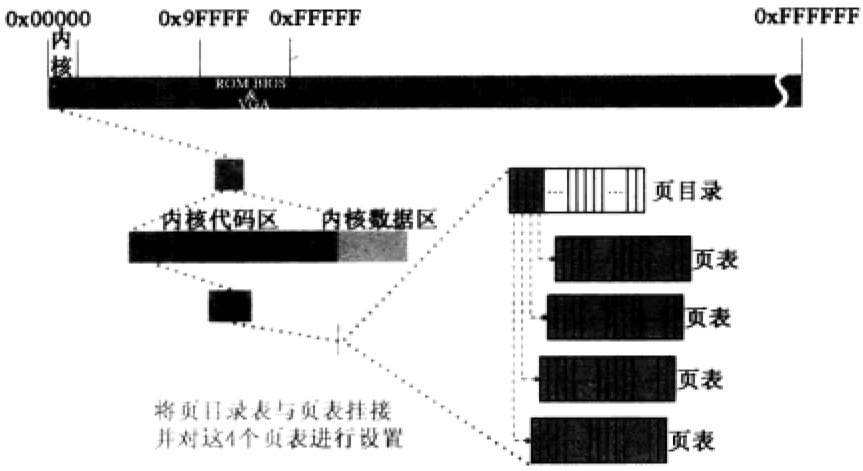


图 5-46　设置内核对应的页目录项和 4 个页表

5.7.3　设置内核对应的全局描述符表 GDT

根据对以上数据的分析我们可以看出，内核的线性地址空间为 0 ～ 16MB，而其物理地址也天然地对应着 0 ～ 16MB 的空间。内核通过恒等映射将 16MB 的线性地址空间涵盖 16MB 的物理地址空间。反观用户进程，它们的线性地址空间只可能在 64MB ～ 4GB 的范围内，并不存在天然对应 0 ～ 16MB 物理内存的映射。这样，它们能映射到什么地方，完全都是由内核来决定的，内核肯定会将它们的线性地址空间映射到物理内存中的“非内核代码区”，即 1MB 以上的区域中，以免干扰自己的运行，而且仍然在 16MB 物理内存以内。所以，内核可以天然地访问到全部用户进程所在的物理空间内也就成了必然，又因为用户进程的线性地址既不可能天然地映射到 0 ～ 16MB 的物理内存中，而且映射到的位置还要由内核来决定，所以它就不可能干扰到内核所在的物理空间。

以下代码决定了内核的段限长是 16MB，这就是一个很好的例证。

```
// 代码路径：boot/head.s：
...
_gdt:    .quad 0x0000000000000000          /* NULL descriptor */
         .quad 0x00c09a0000000fff          /* 16MB */   内核代码段描述符
         .quad 0x00c0920000000fff          /* 16MB */   内核数据段描述符
         .quad 0x0000000000000000          /* TEMPORARY - don't use */
         .fill 252,8,0                     /* space for LDT's and TSS's etc */
```

图 5-47 给出了内核对应的代码段和数据段在 GDT 表中的位置。

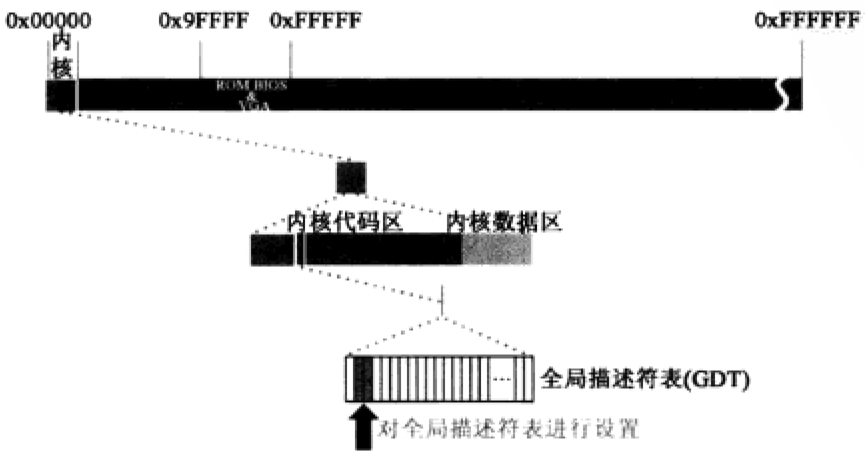


图 5-47　在 GDT 中设置内核的代码段和数据段

5.8 页写保护

在 Linux 0.11 中，主内存中经常发生页面被多个进程所共享的情况，比如父进程创建子进程时，就要将自己的页表项全部拷贝给子进程，使父子进程可以共同操作相同的页面。针对这种情况，内核就会把这些共享页面的属性设置为“只读”，即无论是父进程还是子进程，都只能对这些共享的页面执行读操作，而不能执行写操作，这样就杜绝了由于多个进程往同一个页面中写入数据而导致数据混乱的情况发生。

但是，当父进程创建完子进程后，无论是父进程还是子进程，都免不了对这些共享的页面进行写操作，比如给压栈的数据赋值等。于是，系统为此设计了一套机制——页写保护，一旦父进程或子进程对共享的页面执行写操作，系统就在主内存中申请一个空闲页面，然后把需要执行写操作的数据所在页面的内容在这个新页面中备份，这样新旧页面中就有了两份相同的数据，父子进程分别使用它们，以免相互干扰。这就是“页写保护”机制的工作原理，下面我们就来看一下具体的实现步骤。

5.8.1 进程 A 和进程 B 共享页面

假设现在系统有一个用户进程（进程 A），它自己对应的程序代码已经载入到内存中，此时该进程的程序在内存中所占用的页面的引用计数都为“1”，接下来它开始执行，通过调用 fork() 函数创建一个新进程（进程 B）。在进程创建的过程中，系统将进程 A 的页表项全部拷贝给进程 B，页表项直接对应页面，所以这两个进程就共享页面了，于是系统将共享页面的引用计数加 1，使之成为“2”，并将此共享页面全部设置为“只读”属性，即无论是进程 A 还是进程 B，都只能对这些共享的页面进行读操作，而不能是写操作。执行代码如下：

```
// 代码路径：mm/memory.c：
int copy_page_tables(unsigned long from,unsigned long to,long size)
{
    ...
    for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
        this_page = *from_page_table;
        if (!(1 & this_page))
            continue;
        this_page &= ~2;
        *to_page_table = this_page;
        if (this_page > LOW_MEM) {
            *from_page_table = this_page;
            this_page -= LOW_MEM;
            this_page >>= 12;
            mem_map[this_page]++;
        }
    }
    ...
}
```

页面共享的情况如图 5-48 所示：

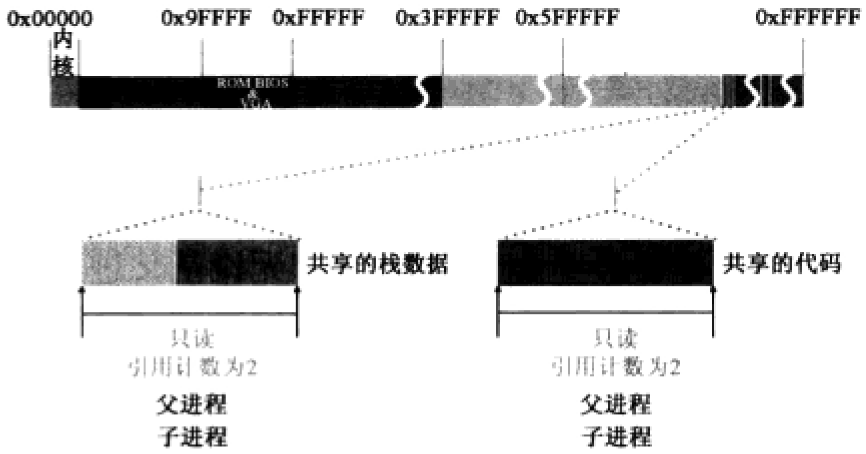


图 5-48 进程 A 与进程 B 共享页面的情况

5.8.2 进程 A 准备进行压栈操作

现在进程 A 成功地创建了一个子进程——进程 B，我们假设接下来轮到进程 A 执行，而且进程 A 接下来的动作是一个压栈动作，我们看看会发生什么。

通过 5.8.1 节中的介绍我们得知，现在进程 A 的程序对应的所有页面都是只读状态的。这就意味着，无论是代码所占用的页面，还是原先压栈的数据所对应的页面，都只能进行读操作，不能进行写操作。然而，压栈动作无疑是一个写操作，现在需要压栈的数据所对应的线性地址经过解析后肯定会映射到原先压栈数据所对应的这些只能够进行读操作的页面中，所以这时候就会产生一个“页写保护”中断来处理这个问题。下面我们来介绍具体的操作步骤，如图 5-49 所示：

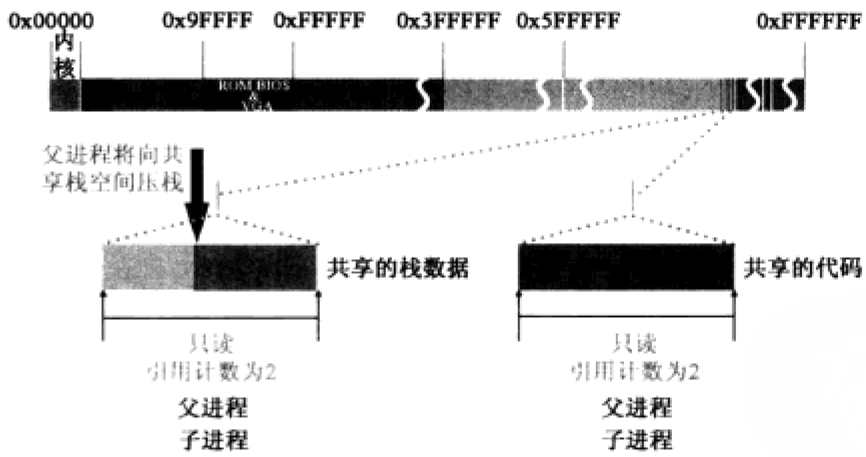


图 5-49 进程 A 准备执行压栈动作

5.8.3 进程 A 的压栈动作引发页写保护

页写保护服务程序最终会执行 un_wp_page 函数来进行页写保护处理。

先要在主内存中申请一个空闲页面（以后我们称之为新页面），以便备份刚才压栈的位置所在页面（以后我们称之为原页面）的全部数据，然后将原页面的引用计数减 1。这是因为，原页面中的数据即将要备份到新页面中了，进程 A 也将要到新页面中操作数据，而不再需要与原页面维持关系了，所以原页面的引用计数减 1。执行代码如下：

```
// 代码路径：mm/memory.c：
void un_wp_page(unsigned long * table_entry)
{
        ...
        if (!(new_page=get_free_page()))
            oom();
        if (old_page >= LOW_MEM)
            mem_map[MAP_NR(old_page)]--;
        ...
}
```

执行页写保护操作为进程 A 新申请一个页面之后的主内存分布如图 5-50 所示：

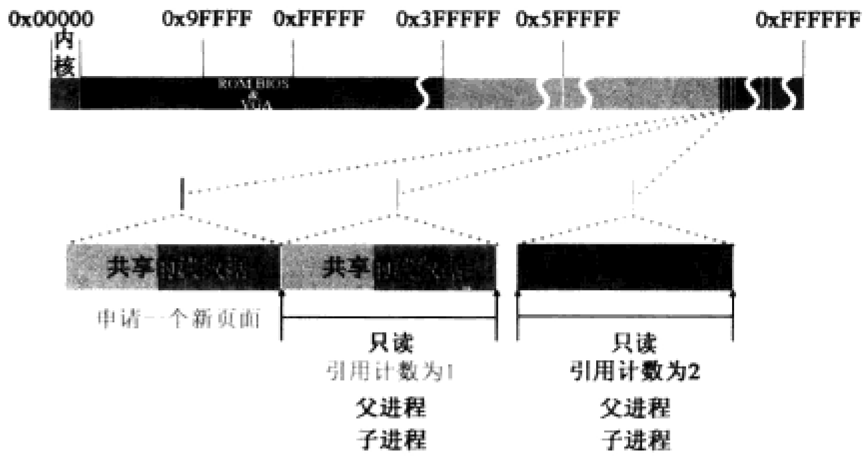


图 5-50　为进程 A 新申请一个页面存储压栈数据

值得注意的是，这里只是将原页面的引用计数减 1，而并没有彻底释放。这是因为在整个操作系统中，所有可能被多个进程所共用的资源，比如文件 i 节点、文件管理表表项、内存页面等，都要通过引用计数来表示它们被使用的状况。当一个进程与它们解除关系后，其他进程未必都与它们解除了关系，简单的“释放”是不行的。

从根本上讲，这些规则都是为了适应多进程的“同时”执行，如果系统是单进程的，那么这些引用计数就完全没有存在的必要了。

5.8.4　将进程 A 的页表指向新申请的页面

新页面虽然申请到了，但此时进程 A 的页表中与原页面对应的页表项还是指向原页面，没有页表项与页面对应，最终还是无法找到物理地址。所以，现在还需要让进程 A 的页表中指向原页面的页表项改指向新页面，并将其使用的属性从“只读”改变为“可读可写”，这样进程 A 才具备了在新页面中处理数据的能力。执行代码如下：

```
// 代码路径：mm/memory.c：
void un_wp_page(unsigned long * table_entry)
{
     ...
     *table_entry = new_page | 7;
     invalidate();
     ...
}
```

此操作执行完毕后，进程 A 新分配的页面所处的状态如图 5-51 所示：

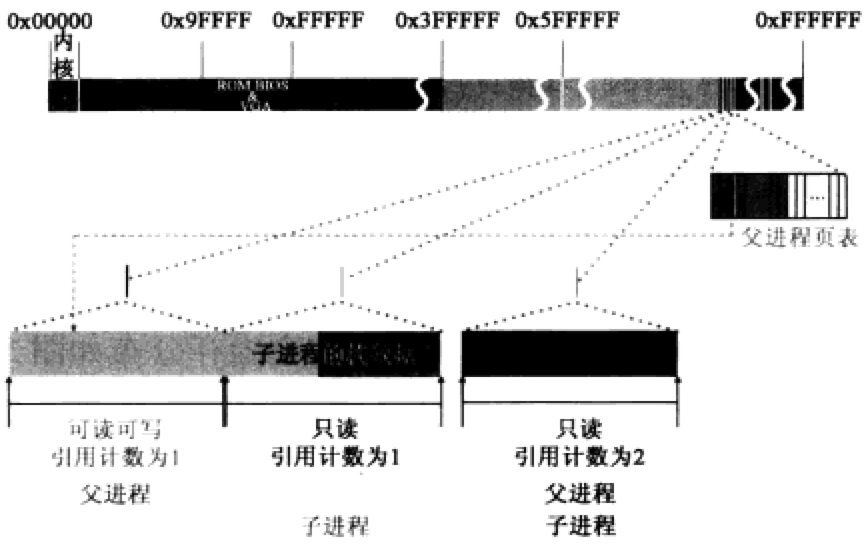


图 5-51　将进程 A 的页表与新申请页面对应

5.8.5　拷贝原页面内容到进程 A 新申请的页面

一切准备就绪后，就可以将原页面中的内容拷贝到新页面中了。拷贝之后，进程 A 就可以在新页面中完成这个压栈动作了。

执行代码如下：

```
//代码路径：mm/memory.c：
void un_wp_page(unsigned long * table_entry)
{
        ...
        copy_page(old_page,new_page);
}
```

拷贝操作完成后，进程 A 新申请的内存页面中的存储情况如图 5-52 所示：

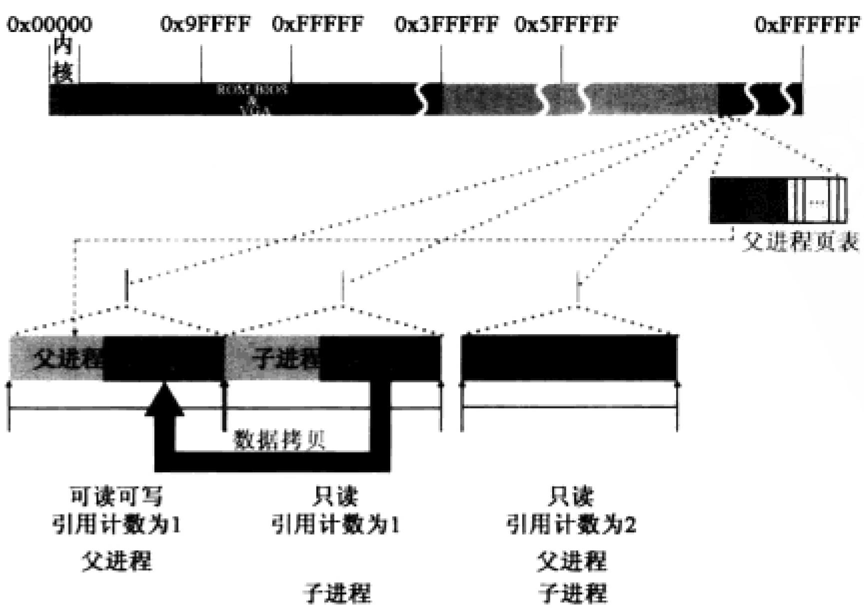


图 5-52　将原页面内容拷贝到进程 A 新申请的页面

5.8.6　进程 B 准备操作共享页面

进程 A 执行一段时间后，就该轮到它的子进程——进程 B 执行了，进程 B 仍然使用着原页面，假设也要在原页面中进行写操作，但是现在原页面的属性仍然是“只读”的，这一点在进程 A 创建进程 B 时就是这样设置的，一直都没有改变过，所以在这种情况下，又需要进行页写保护处理，仍然是映射到 un_wp_page 函数中。执行代码如下：

```
// 代码路径：mm/memory.c：
void un_wp_page(unsigned long * table_entry)
{
    ...
    old_page = 0xfffff000 & *table_entry;
    if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
        *table_entry |= 2;
        invalidate();
        return;
    }
    ...
}
```

由于原页面的引用计数已经在 5.8.3 节中被削减为 1 了，所以现在就要将原页面的属性设置为“可读可写”，然后函数返回。进程 B 修改页面属性的情况如图 5-53 所示。到现在为止，进程 A 和进程 B 在压栈数据的处理方面可以各自操作不同的页面了，这些页面都是可读可写的，而且引用次数都为 1，以后彼此都不会干扰对方。

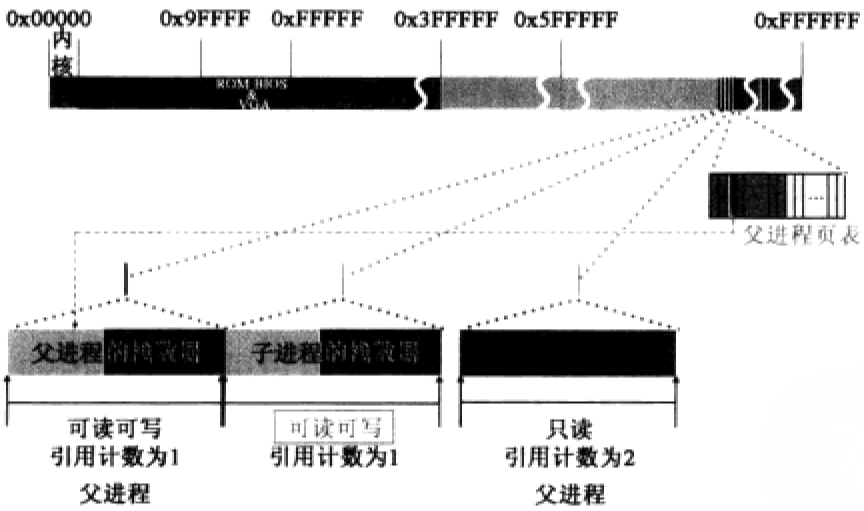


图 5-53　进程 B 将原共享页面性质修改

现在进程 B 并没有自己的程序，如果将来它有了自己的程序，就会和原页面解除关系（理由在 5.2.8 节中已经说明），原页面的引用计数将会继续减 1，于是就变成 0，系统将认定它为“空闲页面”。

5.8.7　假设进程 B 先执行压栈操作的情况

我们现在对 5.8.2 节的内容重新进行假设，假设现在不是父进程——进程 A 执行，而是轮到子进程——进程 B 执行，那么又会出现什么情况呢？

这种情况与 5.8.2 节中所述的情况是“对称”的，即系统先为进程 B 申请一页面空间，然后让进程 B 的页表中与原页表相对应的页表项指向新页面，最后将原页面内容拷贝给新页面，以便进程 B 操作。等到轮到进程 A 执行时，原页面被设置为“可读可写”，使进程 A 仍然使用原页面执行数据操作。

进程 B 先执行压栈操作时的内存分布如图 5-54 所示：

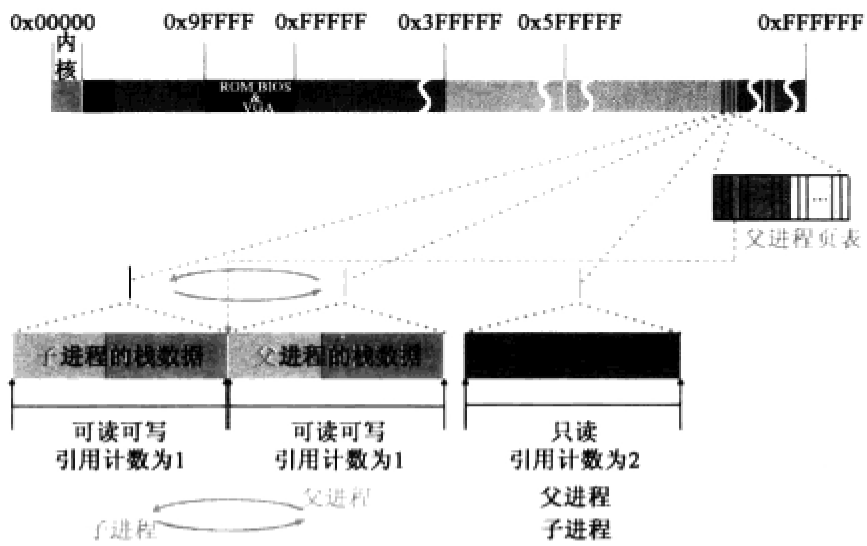


图 5-54　进程 B 先执行压栈操作时的内存分布

值得注意的是，页写保护是一个由系统执行的动作，在整个页写保护动作发生的过程中，用户进程仍然正常执行，它并不知道自己在内存中被复制了，也不知道自己被复制到哪个页面了。系统对用户进程的组织管理和协调，就表现在这里，用户进程能够正常执行是由系统保证的，它自己并不需要知道系统是如何保证的。

5.9 小结

本章分为两个部分对用户进程在内存中的运行情况进行了讲解。

第一部分，以 str1 这个单用户进程为例，对所有用户进程都适用的创建、执行程序的加载、执行程序的执行，以及用户进程的退出进行了讲解。

str1 的创建首先需要在进程槽中注册，之后继承其父进程的全部管理信息并进行调整。str1 的程序以缺页中断的形式，根据需要进行加载。退出时，先由 str1 进程自己处理绝大部分“善后工作”，再由其父进程清除其进程管理结构。

第二部分，以多个进程的运行为例，详细讲解了内核对用户进程的组织管理协调。

多个进程同时运行时，它们在内存中的空间是不连续的。操作系统依托时钟中断对多个进程进行轮询。操作系统通过页写保护机制，对共享页面的多个进程的运行进行协调。

此外，还讲解了操作系统能够控制用户进程，而用户进程却不能控制操作系统的理由。

第 6 章

多个进程“同时”操作一个文件

在第 4 章中，我们以“一个用户进程操作文件为场景”，对文件的具体操作方法进行了介绍；在第 5 章中，我们对多个用户进程在 Linux 0.11 中的轮流执行情况也进行了介绍，本章将在前面两章内容的基础上，通过两个更为复杂的实例来介绍 Linux 0.11 中“多进程操作文件”的具体执行情况。

实例 1：三个进程在操作同一个文件，此时如何形成一个进程等待队列？最后这个队列中的进程又是如何被唤醒的？我们将在第 6.1 节中对这些问题给予详细的解答。

实例 2：三个进程操作三个不同的文件。在这一过程中，会涉及情况更为复杂的“缓冲块申请”、“请求项申请”、“进程等待队列形成”等一系列问题，我们将在第 6.2 节中讲解这些内容。

6.1 三个进程操作同一个文件

假设硬盘上已有一个文件名为 hello.txt 的文件，这个文件的大小为 700B，小于一个数据块大小（1KB），被载入缓冲区后，一个缓冲块就可以承载其全部内容，这三个进程一旦开始操作这个文件，就相当于在依托系统操作同一个缓冲块，这样就会产生进程等待队列。本节将详细介绍该队列的产生过程，以及队列中进程的唤醒过程。

下面我们来介绍实例 1 的场景。三个进程的情况如下，进程 A 是一个读盘进程，其代码如下：

```
void main()
{
char buffer[100];
int i;
//打开文件
int fd = open("/mnt/user/hello.txt", O_RDWR ,0644));
//读文件
read(fd,buffer,sizeof(buffer));
for(i=0;i<1000000;i++)
return;
}
```

该进程执行的目的是将 hello.txt 文件中的 100 个字节读入 buffer 这个数组。

进程 B 也是一个读盘进程，其代码如下：

```
void main()
{
char buffer[200];
int i;
//打开文件
int fd = open("/mnt/user/hello.txt", O_RDWR ,0644));
//读文件
read(fd,buffer,sizeof(buffer));
for(i=0;i<1000000;i++)
return;
}
```

该进程执行的目的是将 hello.txt 文件中的 200 个字节读入 buffer 这个数组。

进程 C 是一个写盘进程，其代码如下：

```
void main()
{

char str1[]="ABCDE";
//打开文件
int i;
int fd = open("/mnt/user/hello.txt", O_RDWR ,0644));
//写文件
write(fd,str1,strlen(str1));
  for(i=0;i<1000000;i++)
return;
}
```

该进程执行的目的是往 hello.txt 文件中写入 str1 这个数组中的字符“ABCDE”。

这三个进程启动顺序为：

进程 A 先启动，并执行；之后进程 B 启动，并执行；最后进程 C 启动，并执行。

下面我们来看具体的执行过程。

6.1.1 进程 A 执行，hello.txt 文件被打开

进程 A 启动并执行后，会执行“int fd = open("/mnt/user/hello.txt", O_RDWR ,0644))”这行代码，以此来确定进程 A 要操作哪个文件。open 最终会映射到 sys_open 函数去执行，sys_open 函数最终的执行结果为：在文件管理表 file_table[64] 中申请一个空闲表项，并以此为中转站，让进程 A 管理结构 task_struct 中的文件管理指针表 *filp[20] 与将要打开的 hello.txt 文件的 i 节点建立关系，从而使进程 A 具备操作这个文件的能力，具体的执行过程我们已经在 4.1 节中进行了详细的介绍。如图 6-1 所示，图中右边还标出了用户进程在主内存空间中的管理结构，正是它承载了与内核中 hello.txt 文件的 i 节点的关联。

6.1.2 进程 A 读取 hello.txt 文件并由于等待硬盘中断而被系统挂起

当前进程为进程 A，它将执行“read(fd,buffer,sizeof(buffer))”这行源代码，read 函数最终会

映射到 sys_read 函数去执行；之后，sys_read 函数会调用 file_read 函数来读取该文件的内容；进入 file_read 函数后，将通过调用 bread 函数从硬盘上读取数据，执行代码如下：

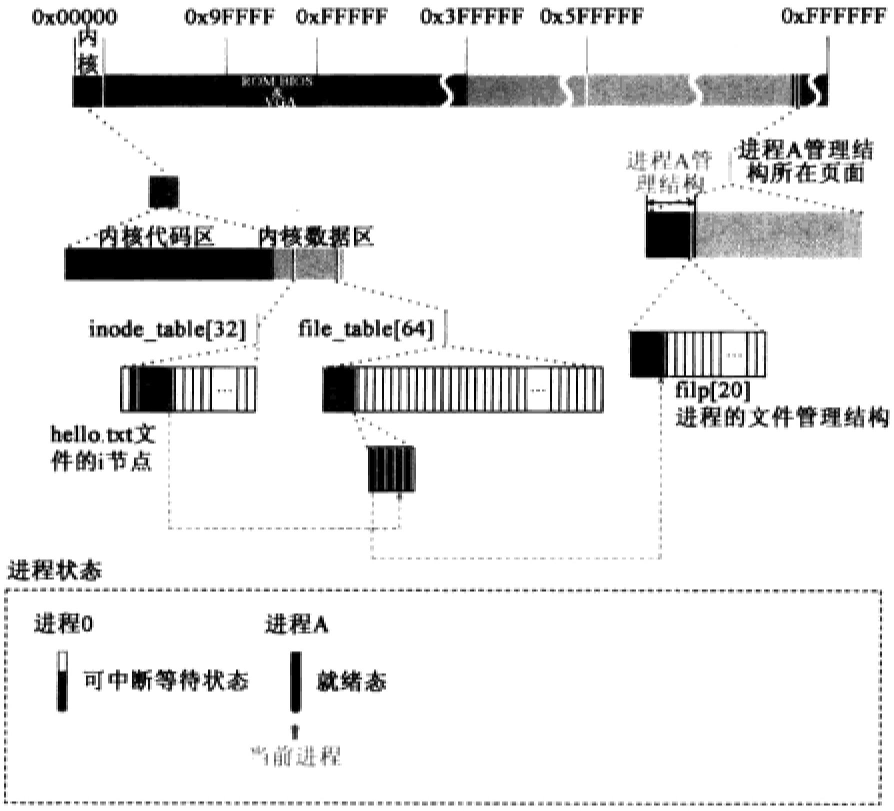


图 6-1　系统为进程 A 打开 hello.txt 文件

```
// 代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
...
if (!(bh=bread(inode->i_dev,nr)))
...
}
```

我们在介绍实例 1 的场景时已经说明，hello.txt 文件的大小为 700B（小于一个数据块 1KB 的大小），于是 bread 此次读取该文件的一个数据块进入缓冲区，就相当于把整个文件的内容全部读入了缓冲区。

进入 bread 函数后，通过调用 getblk 函数在缓冲区中申请一个空闲的缓冲块，再调用 ll_rw_block 函数将该缓冲块加锁，并将其与请求项绑定，最终发送读盘指令，这一过程我们在 2.3 节中已经详细说明，执行代码如下：

```
// 代码路径：fs/buffer.c：
struct buffer_head * bread(int dev,int block)
{
    ...
    if (!(bh=getblk(dev,block)))
    ...
    ll_rw_block(READ,bh);
    ...
}
```

因为硬盘上的数据不可能立即就被读入缓冲块中，所以接下来要调用 wait_on_buffer 函数，

准备将进程 A 挂起，即设置为不可中断等待状态。执行代码如下：

```
// 代码路径：fs/buffer.c：
struct buffer_head * bread(int dev,int block)
{
    ...
    wait_on_buffer(bh);
    ...
}
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    sti();
}
```

进入 wait_on_buffer 函数后，由于缓冲块已经被加锁，所以 bh->b_lock 为 1，于是进入 sleep_on 函数执行，执行代码如下：

```
// 代码路径：kernel/sched.c：
void sleep_on(struct task_struct **p)
{
    ...
        tmp = *p;
        *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    ...
}
```

通过实例 1 的场景分析我们得知，在进程 A 之前，系统并没有操作过该缓冲块，所以就更不可能有哪个进程正在等待该缓冲块解锁，因此由 wait_on_buffer 传递给 sleep_on 的实参 bh->b_wait 一定为空，于是 tmp = *p; 这一行代码执行后，tmp 也为 NULL。

等到执行完 *p = current; 这一行代码后，bh->b_wait 就是进程 A 了，这意味着，进程 A 在等待 hello.txt 文件的缓冲块解锁，于是进程 A 就是进程等待队列的队首，之后，将进程 A 设置为“不可中断等待状态”。

此时系统中还只有进程 A 这一个用户进程，于是进程 A 被挂起后，系统中所有进程都处于非就绪状态，因此切换到进程 0 去执行，理由已经在第 2 章和第 5 章反复讲解过。

与此同时，硬盘也正在向数据寄存器端口中传递数据。

请读者注意，图 6-2 中代表进程 A 的进程条已经变成了灰色，表示进程 A 已经挂起。

6.1.3 进程 B 准备打开 hello.txt 文件

现在启动进程 B，通过实例 1 的场景介绍我们得知，进程 B 也是读盘进程，而且也是读取 hello.txt 文件中的数据。进程 B 的执行也是从 open 函数的调用开始的，即执行 int fd = open("/mnt/user/hello.txt", O_RDWR ,0644)); 这一行代码，以此来确定进程 B 要操作哪个文件。open 最

终会映射到 sys_open 函数去执行，sys_open 函数最终的执行结果为：在文件管理表 file_table[64] 中又申请一个空闲表项，并以此为中转站，让进程 B 管理结构 task_struct 中的文件管理指针表 *filp[20] 与将要打开的 hello.txt 文件的 i 节点建立关系，从而使进程 B 具备操作这个文件的能力，具体的执行过程，我们已经在 4.1 节中进行了详细的介绍。

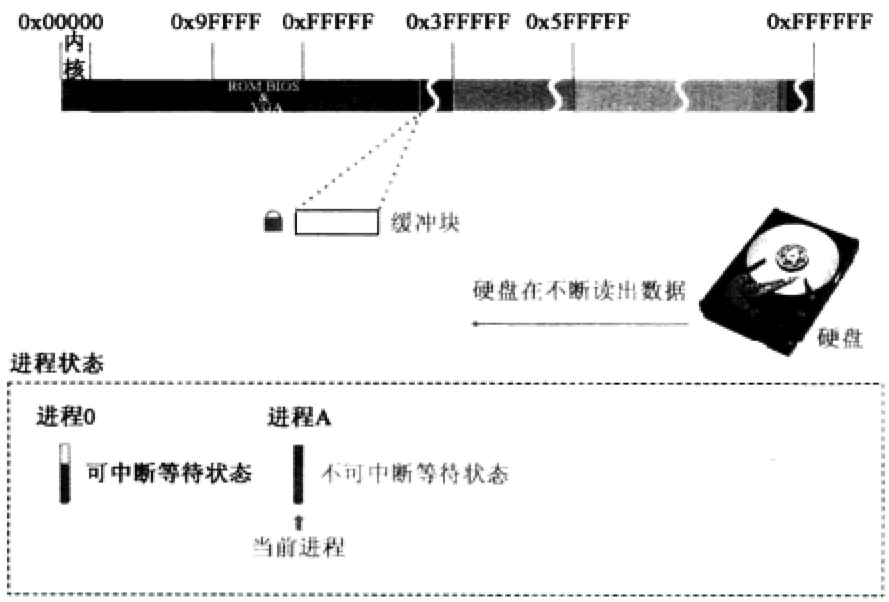


图 6-2　系统为进程 A 读取 hello.txt 的数据并将进程 A 挂起

执行代码如下：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
    ...
    for(fd=0 ; fd<NR_OPEN ; fd++)
        if (!current->filp[fd])
            break;
    ...
    for (i=0 ; i<NR_FILE ; i++,f++)
        if (!f->f_count) break;
    ...
    (current->filp[fd]=f)->f_count++;
    ...
}
```

之后，调用 open_namei 函数，准备获取 hello.txt 文件的 i 节点，执行代码是：

```
// 代码路径：fs/open.c：
int sys_open(const char * filename,int flag,int mode)
{
    ...
    if ((i=open_namei(filename,flag,mode,&inode))<0)
    ...
}
```

图 6-3 表示了进程 B 管理结构中的文件管理指针表 *filp[20] 与内核文件管理表 file_table[64] 的挂接情况。同时，请读者注意硬盘还在不断的读出数据。

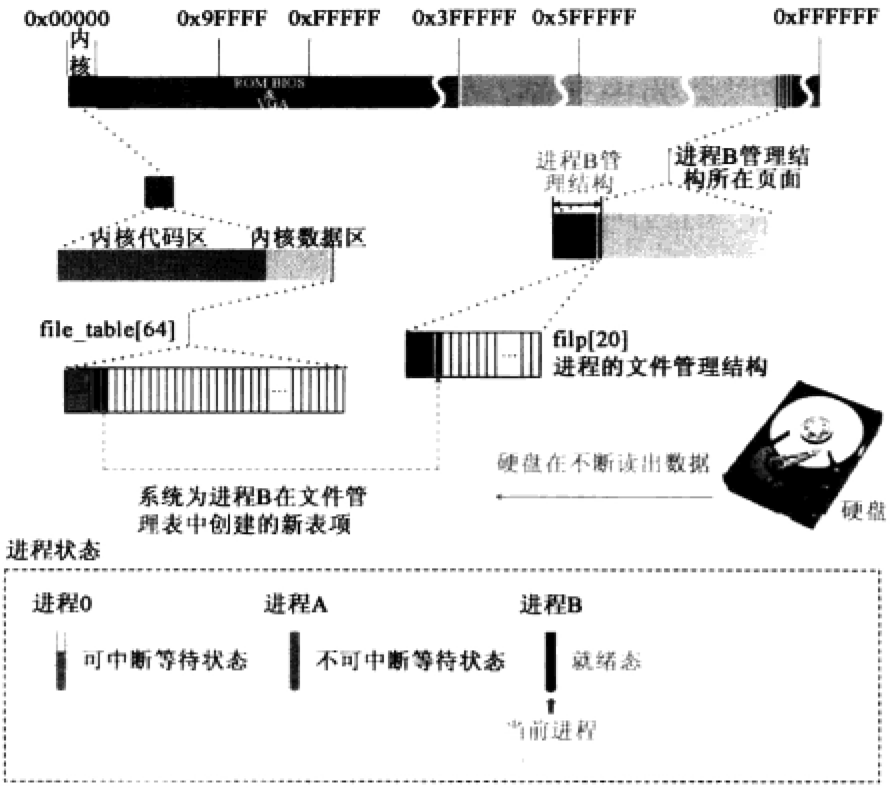


图 6-3　系统将进程 B 与内核文件管理表建立关系

值得注意的是，刚才进程 A 打开 hello.txt 文件的时候，该文件已经在文件管理表中登记了，现在进程 B 打开同样的文件，为什么还要重新登记呢？

这是因为不同的进程对文件会有不同的操作，这就要求系统要“分别记录”它们的操作情况，这些操作情况的关键信息就记录在文件管理表 file_table[64] 这个数据结构中。

比如说，这个数据结构表项中有一个成员是 f_pos;，它用来确定当前文件操作指针，不同的进程操作同一个文件，其当前文件操作指针的位置很有可能是不同的，所以必须让“一本账”对应“一个进程”，否则将引起数据混乱。

6.1.4　系统准备为进程 B 获取 hello.txt 文件的 i 节点

当前进程为进程 B，进入 open_namei 函数后，通过调用 dir_namei 和 find_entry 等函数来确定这个文件的 i 节点；之后，调用 iget 函数，准备获取 hello.txt 文件的 i 节点，执行代码如下：

```
// 代码路径：fs/namei.c：
int open_namei(const char * pathname, int flag, int mode,
    struct m_inode ** res_inode)
{
    ...
    if (!(inode=iget(dev,inr)))
    ...
}
```

进入 iget 函数后，调用 get_empty_inode 函数，准备在 i 节点管理结构 inode_table[32] 中获取一个空闲项，以便装载这个文件 i 节点的信息，执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
```

```
{
    ...
    empty = get_empty_inode();
    ...
}
```

系统在内核 i 节点表中寻找一个空位的过程如图 6-4 所示：

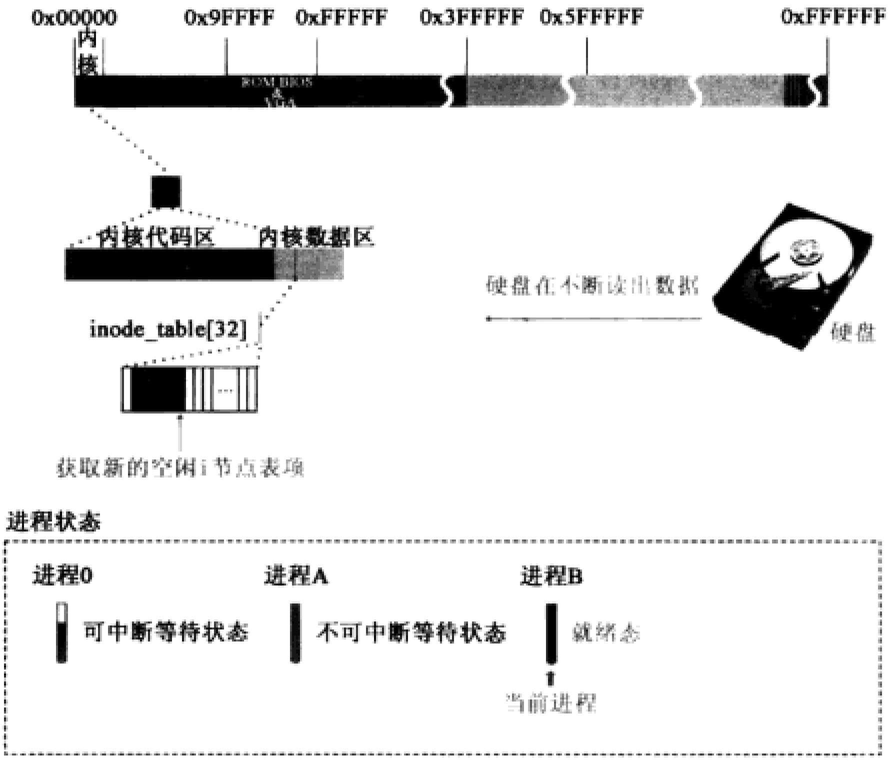


图 6-4 系统为载入 i 节点创造条件

6.1.5 系统找到 hello.txt 文件已经载入的 i 节点

当前进程仍为进程 B。接下来，遍历 i 节点管理结构 inode_table[32]，通过识别设备号和 i 节点号来确定这个 i 节点在 inode_table[32] 这个管理结构中是否已经存在。

由于进程 A 和进程 B 读取的是同一个文件，所以刚才系统就已经把 hello.txt 文件的 i 节点读入 i 节点表了，于是该 i 节点一定会被找到，此次就不用再读盘了，而是通过执行“inode->i_count++”这样的语句，直接将这个 i 节点的引用次数加 1，以表示该 i 节点又被引用了一次，之后释放掉刚刚在 6.1.4 节中申请的空闲 i 节点项 iput（empty），并返回 i 节点指针。执行代码如下：

```
//代码路径：fs/inode.c：
struct m_inode * iget(int dev,int nr)
{
    ...
    while (inode < NR_INODE+inode_table)
    {
        if (inode->i_dev != dev || inode->i_num != nr)
        ...
        inode->i_count++;
        ...
        if (empty)
            iput(empty);
        return inode;
```

```
    }
        ...
    }
```

如图 6-5 所示，请读者注意与图 6-4 进行比较，可以看出之前系统为读取 hello.txt 文件在内核 i 节点表中准备的空位已经被释放。而代表 hello.txt 文件的 i 节点管理结构中的 i_count 字段加 1，表示进程 B 也将对 hello.txt 文件进行操作：

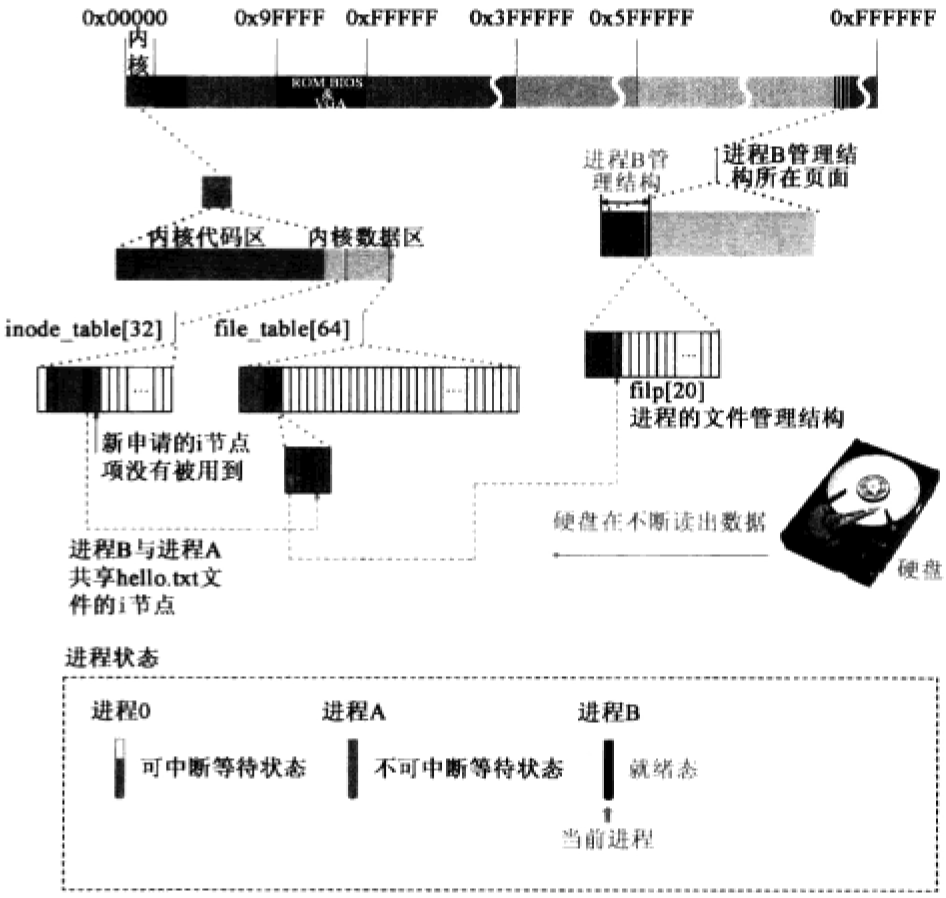


图 6-5 系统为进程 B 找到了 hello.txt 文件已经载入的 i 节点

这里值得注意的是：同样是操作 hello.txt 这个文件，为什么系统要在 6.1.3 节的文件管理表 file_table[64] 这个数据结构中为进程 A 和进程 B 分别申请两个管理表项，而此时却在 i 节点管理表 inode_table[32] 中让进程 A 和进程 B 共用一个 i 节点呢？

理由还要从"账本"说起，6.1.3 节中已经介绍过，进程 A 和进程 B 对同一个文件会进行不同的操作，所以要为这两个进程分别记一本账；而此时共用一个 i 节点，是因为，无论有多少个进程操作 hello.txt 这个文件，该文件永远有且仅有一个 i 节点，这个 i 节点中记载着这个文件的属性信息，比如文件大小等，每个文件也只能拥有一套这样的属性信息。无论是进程 A 还是进程 B，只要某个进程操作文件后，如果对文件的属性信息进行了改动，最终都要落实在这个 i 节点上，所以，它们只能共用一个 i 节点。

6.1.6 系统准备为进程 B 从硬盘上读取 hello.txt 文件

当前进程为进程 B，接下来调用 read 函数，即执行 read(fd,buffer,sizeof(buffer))；这一行代码，以便读取 hello.txt 文件的内容。

read 最终会映射到 sys_read 函数去执行。sys_read 函数会调用 file_read 函数来读取该文件的内容，进入 file_read 函数后，将通过调用 bread 函数来为从硬盘上读取数据做准备，执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
...
if (!(bh=bread(inode->i_dev,nr)))
...
}
```

进入 bread 函数后，会调用 getblk 函数，以便为即将载入的数据申请一个缓冲块，执行代码如下：

```
// 代码路径：fs/buffer.c：
struct buffer_head * bread(int dev,int block)
{
      ...
      if (!(bh=getblk(dev,block)))
      ...
}
```

上述过程如图 6-6 所示，可以看到由于系统已经对同样的数据块执行了读取操作，因此进程 B 在哈希表中直接找到了这个缓冲块，6.1.7 节将继续讲解这个过程。

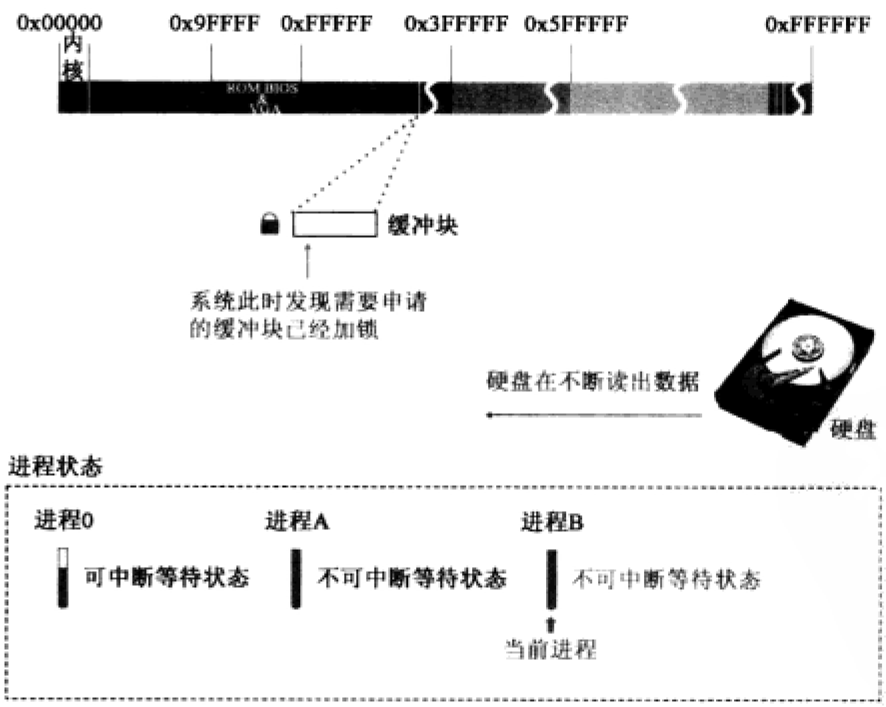


图 6-6 系统为读取 hello.txt 文件寻找缓冲块

6.1.7 系统找到了正在操作的缓冲块，将进程 B 挂起

当前进程为进程 B，进入 getblk 函数后，先调用 get_hash_table，在缓冲区的哈希表管理结构中进行搜索，以便确认将要读取的数据块在缓冲区中是否已经存在。如果已经存在，就不用继

续从硬盘上读取了，直接将该缓冲块指针返回，执行代码如下：

```
// 代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
{
...
if (bh = get_hash_table(dev,block))
          return bh;
...
}
```

由于系统已经对这个文件对应的缓冲块进程操作了（我们在实例 1 的场景介绍时已经说明），所以此时为进程 B 服务，操作的是同一个缓冲块。这就意味着，一定会在搜索哈希表的时候找到它，于是就将这个缓冲块的引用次数累加为 2，以表示再次引用了该缓冲块，之后就返回了，如图 6-6 所示。

返回后继续在 bread 函数中执行，将要调用 wait_on_buffer 函数来确定是否需要将当前进程，即进程 B 挂起，执行代码如下：

```
// 代码路径：fs/buffer.c：
struct buffer_head * bread(int dev,int block)
{
    ...
    wait_on_buffer(bh);
    ...
}
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    sti();
}
```

由于此时硬盘中的数据并没有传递完毕，这就导致这个缓冲块仍然处于加锁状态（加锁的情况我们在 6.1.2 节中已经说明），所以 bh->b_lock 这个条件为真，这样将进入 sleep_on 函数执行，准备将进程 B 挂起，即设置为不可中断等待状态，执行代码如下：

```
// 代码路径：kernel/sched.c：
void sleep_on(struct task_struct **p)
{
    ...
        tmp = *p;
        *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    ...
}
```

通过 6.1.2 节中的分析我们得知，此时实参 bh->b_wait 代表着进程 A，即此时进程 A 在等待着这个缓冲块解锁。于是，在执行完 tmp = *p; 这行代码后，tmp 就是进程 A；之后，执行完

*p = current; 这行代码后，*p（即实参 bh->b_wait）就代表进程 B，这意味着进程 B 也在等待这个缓冲区的解锁，此时进程 B 就是进程等待队列的队首；最后，将进程 B 设置为“不可中断等待状态”，并准备进程切换，如图 6-6 所示。

6.1.8 系统再次切换到进程 0 执行

此时系统中的两个读盘进程：进程 A 和进程 B 也都处于不可中断等待状态了，因此系统中所有的进程又都处于非就绪状态了，于是再次切换到进程 0 去执行，理由和 6.1.2 节中所述一致。

与此同时，硬盘也正在向数据寄存器端口中传递数据。如图 6-7 所示：

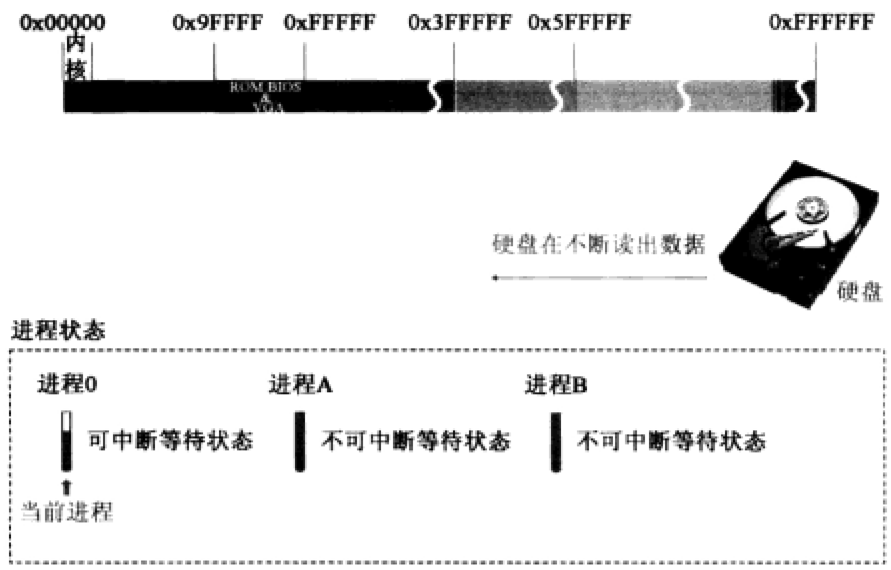


图 6-7 再次切换到进程 0 执行

6.1.9 进程 C 启动并打开 hello.txt 文件

现在启动进程 C，通过实例 1 的场景介绍我们得知，进程 C 是写盘进程，与进程 A 和进程 B 操作同样的 hello.txt 文件，向该文件中写入数据。进程 C 的执行也是从 open 函数的调用开始的，即执行 int fd = open("/mnt/user/hello.txt", O_RDWR ,0644)); 这一行代码，以此来确定进程 C 要操作哪个文件。open 最终会映射到 sys_open 函数去执行，sys_open 函数最终的执行结果为：在文件管理表 file_table[64] 中再次申请一个空闲表项（“再次申请的理由已经在 6.1.3 节中说明”），并以此为中转站，让进程 C 管理结构 task_struct 中的文件管理指针表 *filp[20] 与将要打开的 hello.txt 文件的 i 节点建立关系，从而以使进程 C 具备操作这个文件的能力。

之后 sys_open 调用 open_namei 函数，以便获取 hello.txt 文件的 i 节点，同样会在 i 节点表 inode_table[32] 中找到该文件的 i 节点，于是将该 i 节点的引用次数再次加 1，以表示它又被引用了一次，理由我们在 6.1.5 节中已经说明。

上述过程如图 6-8 所示，所代表的具体含义与图 6-3 基本相同，只是当前进程变为了进程 C。

到这里，我们就可以看出，无论多少进程，只要操作同一个文件，都沿用这个模式，即在 file_table 文件管理表中为该进程新申请一个表项，以便记录该进程自身操作的情况。另外，所有进程都共用一个 i 节点表项，多一个进程操作它，该表项的引用计数就加 1。

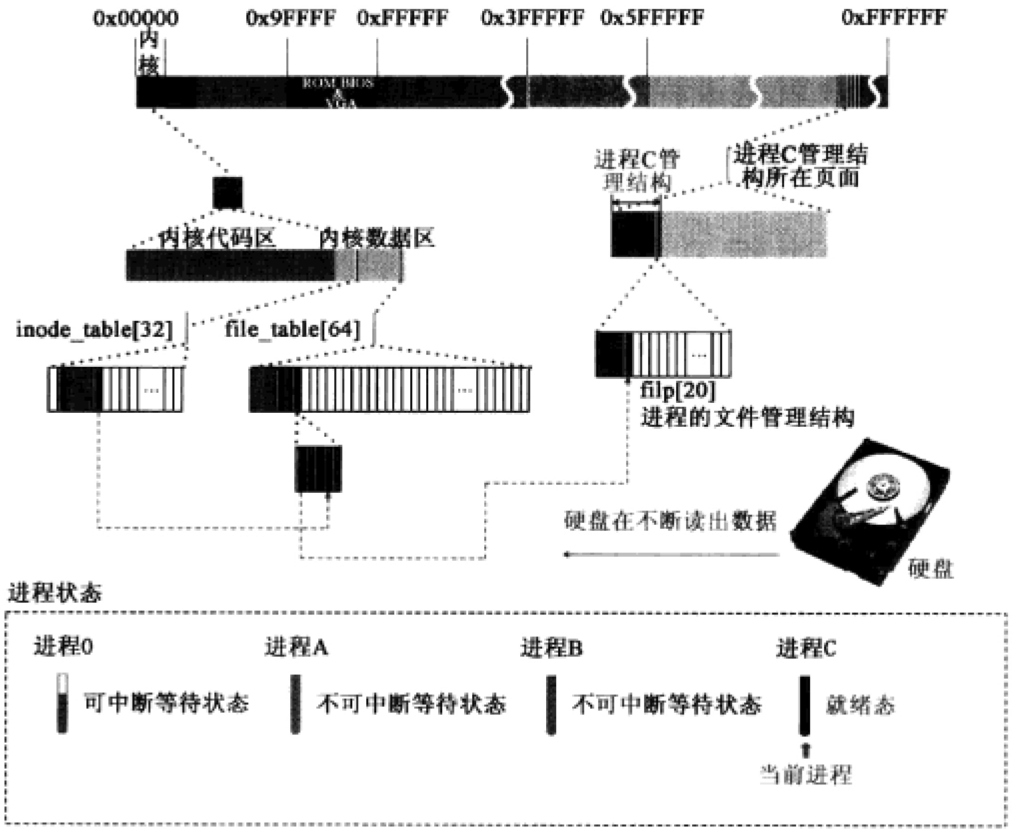


图 6-8　进程 C 准备读取 hello.txt 文件

6.1.10　进程 C 也由于等待缓冲块解锁而被系统挂起

当前进程为进程 C，它最终也要通过调用 bread 函数来把 hello.txt 文件的数据块读进缓冲区中，这样才能写入数据，理由我们在 4.4 节中已经说明。通过本节前面的介绍我们得知，对应的缓冲块中还没有载入该文件数据块的数据，进程 A 和进程 B 也正在等待着该缓冲块解锁，所以进程 C 此时也要等待该缓冲块解锁，于是 bread 函数调用 wait_on_buffer 函数，准备将进程 C 挂起，执行代码如下：

```
// 代码路径：fs/buffer.c：
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    sti();
}
void sleep_on(struct task_struct **p)
{
    ...
        tmp = *p;
        *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    schedule();
    ...
}
```

通过 6.1.7 节中的分析我们得知，此时实参 bh->b_wait 代表着进程 B，即此时进程 B 在等待着这个缓冲块解锁。于是，在执行完 tmp = *p; 这行代码后，tmp 就是进程 B；之后，执行完

*p = current; 这行代码后，*p（即实参 bh->b_wait）就代表进程 C，意味着进程 C 也在等待这个缓冲块的解锁，此时进程 C 就是进程等待队列的队首；最后，将进程 C 设置为"不可中断等待状态"，并准备进程切换。

与此同时，硬盘中的数据还没有传递完毕。

该过程以及进程 C 被挂起的情况如图 6-9 所示：

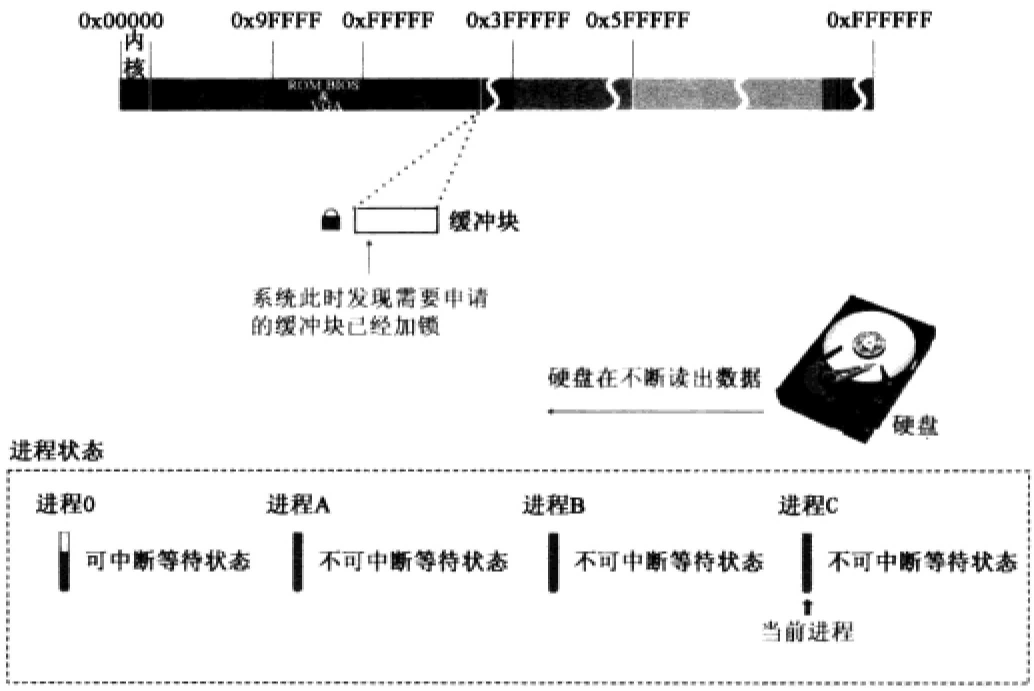


图 6-9　进程 C 挂起并进入对应 hello.txt 文件的缓冲块解锁的等待队列

6.1.11　缓冲块解锁后先唤醒进程 C

此时进程 A、进程 B 和进程 C 都已经被挂起了，系统中所有的进程又都处于非就绪状态了，所以再次默认切换到进程 0 去执行，直到数据读取完毕，硬盘产生中断，如图 6-10 所示：

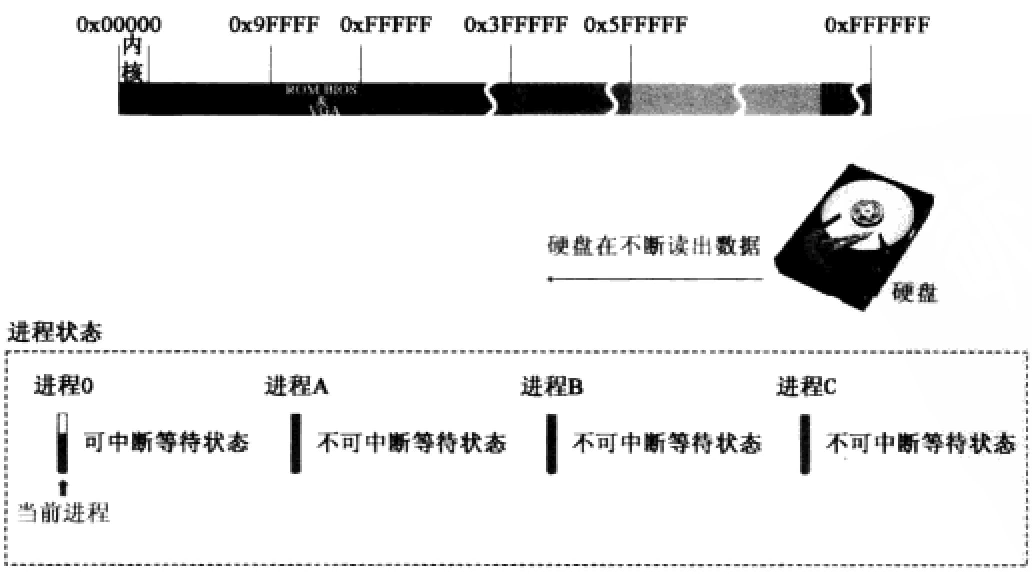


图 6-10　再次切换到进程 0 执行

硬盘中断产生后，中断服务程序将开始工作。我们假设此时硬盘已经将指定的数据全部载入缓冲块，则中断服务程序开始工作后，必将导致"三个进程都在等待的"缓冲块解锁；之后，等

待该缓冲块的进程就没必要等待了，它可以继续执行了，执行代码如下：

```
// 代码路径：kernel/blk_dev/blk.h：
extern inline void end_request(int uptodate)
{
     ...
     unlock_buffer(CURRENT->bh);
     ...
}
extern inline void unlock_buffer(struct buffer_head * bh)
{
    ...
    bh->b_lock=0;
    wake_up(&bh->b_wait);
}
```

通过 6.1.10 节的介绍我们得知：

此时 bh->b_wait 所代表的是进程 C，所以，这里调用 wake_up 函数后会将进程 C 唤醒，即设置为就绪状态，执行代码如下：

```
// 代码路径：kernel/sched.c：
void wake_up(struct task_struct **p)
{
     if (p && *p) {
          (**p).state=0;
          *p=NULL;
     }
}
```

如图 6-11 所示，代表进程 C 的进程条表示进程 C 已处于就绪态：

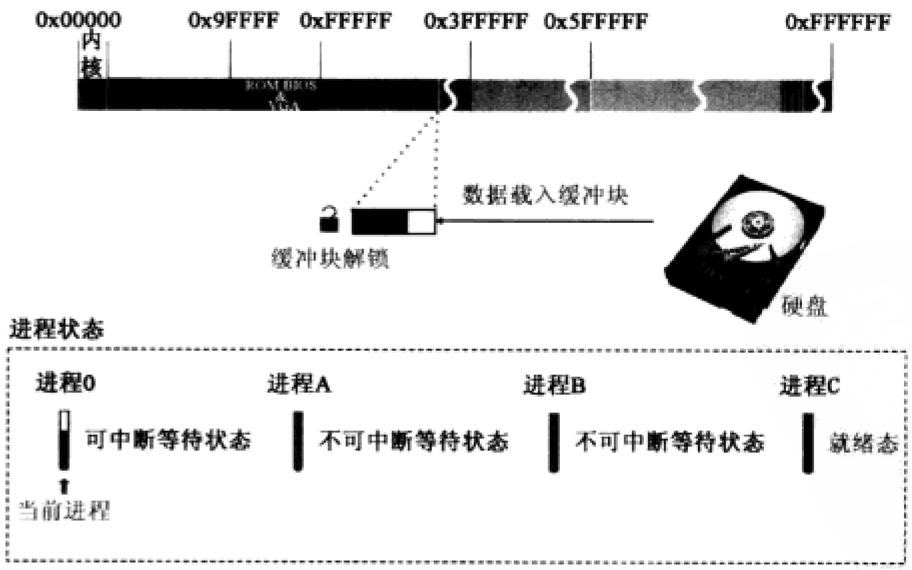


图 6-11　进程 C 被唤醒

6.1.12　系统将进程 B 设为就绪状态

中断服务程序结束后，再次返回到进程 0 中，并切换到就绪的进程 C。进程 C 成为当前进程后，系统从 schedule 函数的 switch_to（switch_to 是一个宏，有关宏展开的内容在第 4 章 4.1.1 中

已详细介绍过）中的 "cmpl %%ecx,_last_task_used_math\n\t" \ 这一行代码开始执行（为什么是这一行，我们已经在第 2 章和第 5 章中关于进程切换的部分反复介绍了），最终返回到 sleep_on 函数中。返回到 sleep_on 函数中以后，马上要执行的代码如下：

```
// 代码路径：kernel/sched.c：
void sleep_on(struct task_struct **p)
{
    ...
    if (tmp)
        tmp->state=0;
}
```

注意：此时是进程 C 是当前进程，tmp 是进程 C 内核栈中的一个数据，它是属于进程 C 的数据，进程 C 切换之前把它设置成什么，它现在仍然还是什么。通过 6.1.10 节的介绍我们得知，当进程 C 为当前进程时，在切换之前，tmp 已经被设置为“进程 B”的指针了，所以，此时执行 tmp->state=0; 这行代码就会将进程 B 的状态设置为就绪状态。如图 6-12 所示：

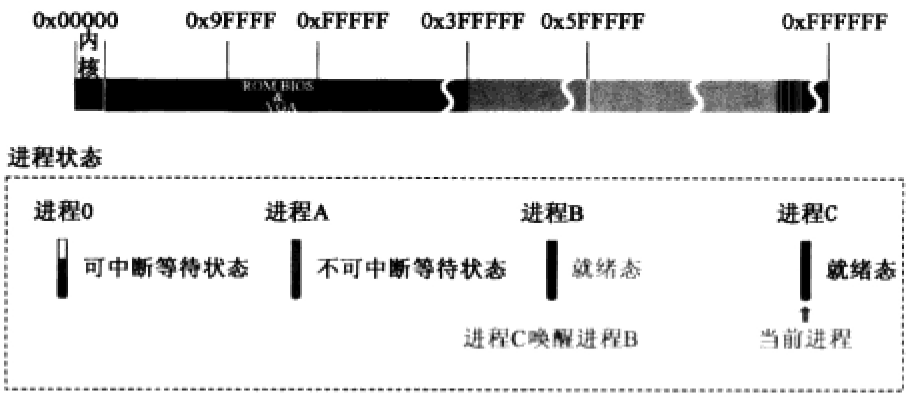


图 6-12　进程 C 将进程 B 唤醒

6.1.13　系统将指定数据写入缓冲块

当前进程为进程 C，系统将把进程 C 源程序中指定的 str1 这个字符数组中的数据写入到 hello.txt 文件对应的缓冲块中，执行代码如下：

```
// 代码路径：fs/file_dev.c：
int file_write(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    ...
    while (c-->0)
            *(p++) = get_fs_byte(buf++);
    ...
}
```

将数据从用户所属的主内存写入到缓冲区的情况如图 6-13 所示：

6.1.14　写入完成后，进程 C 继续执行

进程 C 将 str1 这个字符数组中的数据写入缓冲块后，最终会返回到用户程序中继续执行，接下来将执行进程 C 程序中的 for(i=0;i<1000000;i++) 这一行代码，在此过程中，时钟中断不断

地发生，进程 C 的时间片将被不断地削减。如图 6-14 所示：

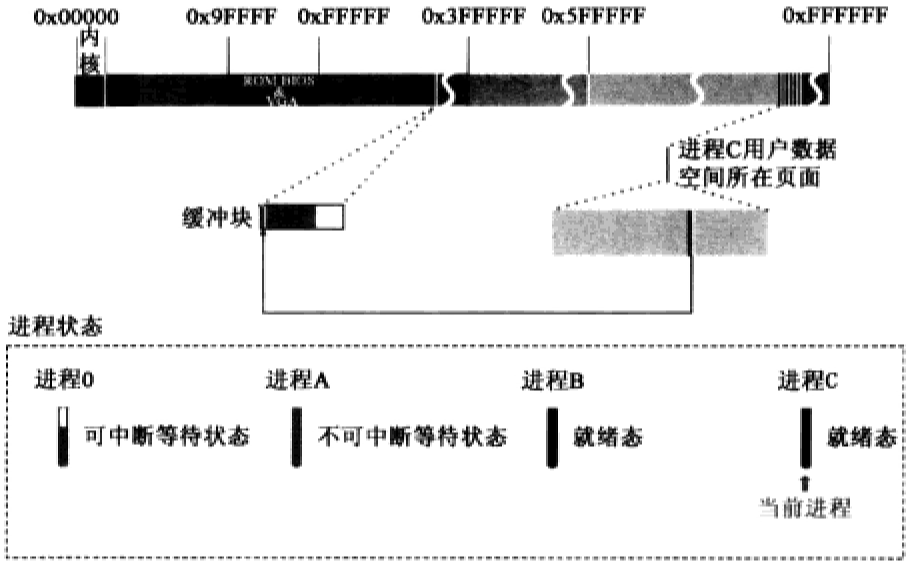


图 6-13　系统为进程 C 将数据写入指定缓冲块

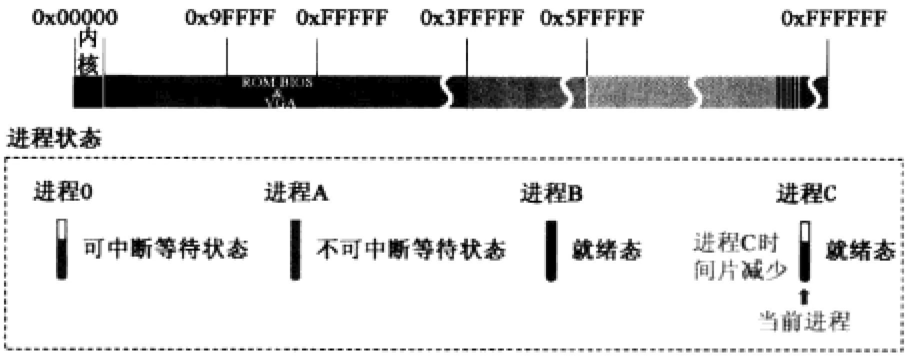


图 6-14　进程 C 执行过程中，时间片不断削减

6.1.15　进程 C 准备切换到进程 B

进程 C 不断执行，时间片在不断的削减。

如图 6-15 所示，请读者注意图中下方代表各个进程执行状态的进程条中，进程 C 的时间片在不断的削减。

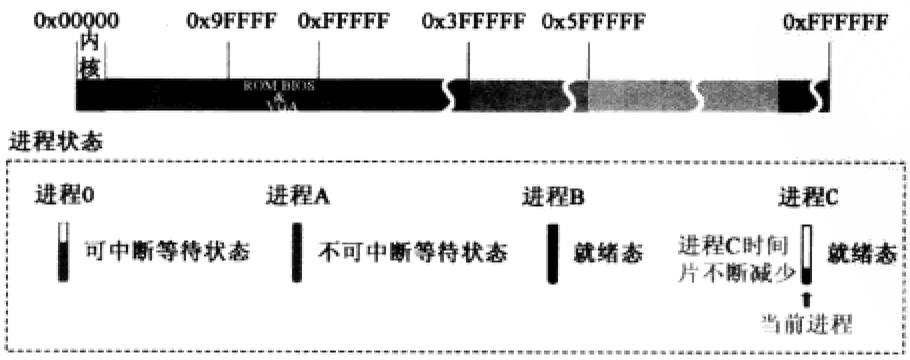


图 6-15　进程 C 的时间片减少到 0

6.1.16　进程 C 切换到进程 B 执行，进程 B 唤醒进程 A

通过 6.1.12 节中的介绍我们得知，进程 C 在成为“当前进程”的时候，系统第一件事就是

将进程 B 设置为了就绪状态，所以此时系统中只有进程 B 和进程 C 两个进程是就绪状态，进程 C 的时间片用完了，就会切换到进程 B 去执行。如图 6-16 所示：

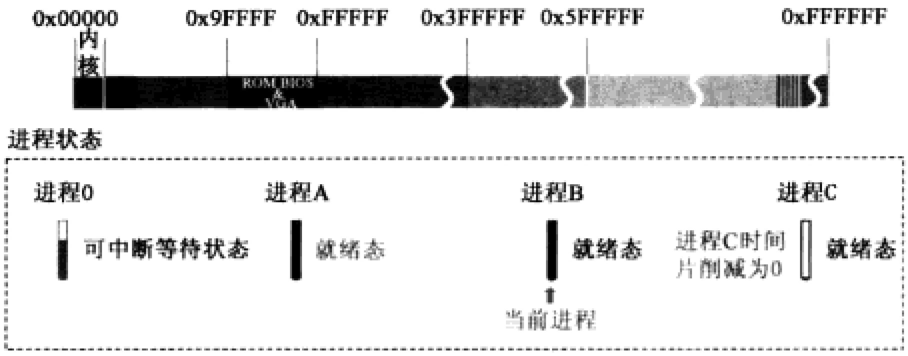


图 6-16　切换到进程 B 执行

进程 B 从 schedule 函数 switch_to 中的 "cmpl %%ecx,_last_task_used_math\n\t" \ 这一行开始不断执行（为什么是这一行，我们已经在第 2 章和第 5 章进程切换的部分反复介绍了），最终返回到 sleep_on 函数中，返回后，马上要执行的代码如下：

```
// 代码路径：kernel/sched.c：
void sleep_on(struct task_struct **p)
{
    ...
    if (tmp)
        tmp->state=0;
}
```

注意：此时是进程 B 为当前进程，tmp 是进程 B 内核栈中的一个数据，它是属于进程 B 的数据，进程 B 切换之前，系统把它设置成什么，它现在仍然还是什么。通过 6.1.7 节的介绍我们得知，在切换之前，tmp 已经被设置为“进程 A”的指针了，所以，此时执行 tmp->state=0; 这行代码，就会将进程 A 的状态设置为就绪状态。

如图 6-17 所示：

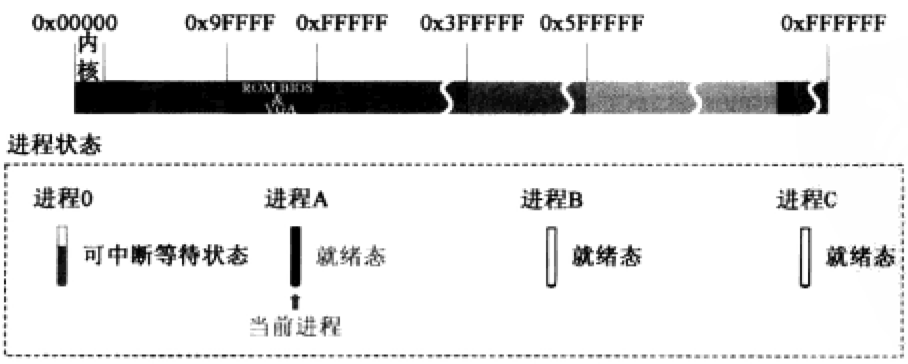


图 6-17　进程 B 唤醒进程 A

6.1.17　进程 B 不断执行，直到时间片减为 0 后切换到进程 A 执行

当前进程为进程 B，将缓冲块中指定的数据读出，之后回到进程 B 的程序中执行 for(i=0;i<1000000;i++) 这一行代码。随着时钟中断的不断发生，进程 B 的时间片削减为 0 后，由

于此时系统中只有进程 A 处于就绪状态，而且它的时间片没有被削减为 0，所以就会切换到进程 A 去执行。

进程 A 从 schedule 函数 switch_to 中的 "cmpl %%ecx,_last_task_used_math\n\t" \ 这一行代码开始不断执行（为什么是这一行，我们已经在第 2 章和第 5 章进程切换的部分反复介绍了），最终返回到 sleep_on 函数中，返回后，马上要执行的代码如下：

```
// 代码路径：kernel/sched.c：
void sleep_on(struct task_struct **p)
{
    ...
    if (tmp)
        tmp->state=0;
}
```

注意：此时当前进程是进程 A，tmp 是进程 A 内核栈中的一个数据，它是属于进程 A 的数据，进程 A 切换之前，系统把它设置成什么，它现在仍然还是什么。通过 6.1.2 节的介绍我们得知，在切换之前，tmp 已经被设置为“NULL”了，所以，此时就不会执行 tmp->state=0; 这行代码。

通过 6.1.2 节、6.1.7 节、6.1.10 节中的介绍我们得知，这三个进程挂起的顺序依次为进程 A、进程 B、进程 C，这三个进程都在等待同一缓冲块的解锁，这样就构成了一个进程等待队列；再通过 6.1.11 节、6.1.12 节、6.1.16 节中介绍的内容得知，此三个进程被唤醒的顺序为进程 C、进程 B、进程 A。这就是进程等待队列的形成，以及唤醒的全部过程。

6.1.18 进程 A、B、C 退出，写入数据由 update 进程同步

进程 A、进程 B 和进程 C 各自执行完毕后就退出了，进程 C 写入缓冲区中的这些关于 hello.txt 文件的数据将通过 update 进程同步到硬盘上，我们在 4.4.5 节中已经介绍过，update 进程是一个常驻于内核的进程，它的职责就是将缓冲区中的数据定期地同步到硬盘上，具体过程我们已经在 4.4.5 节中说明。该过程如图 6-18 所示。

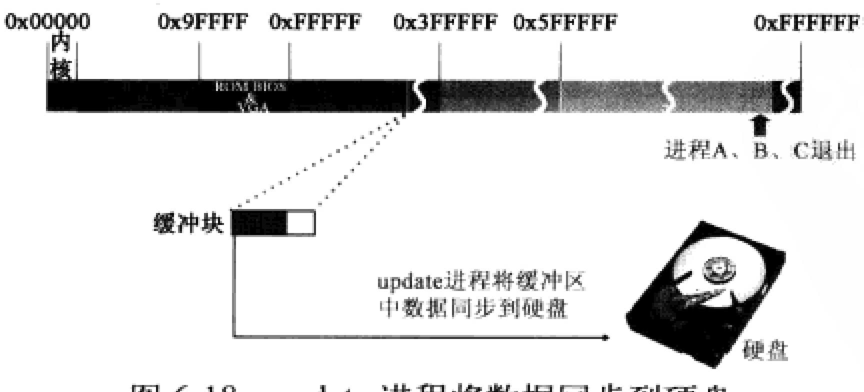


图 6-18 update 进程将数据同步到硬盘

6.2 缓冲区与外设的数据同步

本节将以三个进程操作三个不同的文件为场景（即实例 2）来详细介绍 Linux 0.11 中更为复

杂的多进程操作文件的情况，这一过程中会涉及情况更为复杂的“缓冲块申请”、“请求项申请”、“进程等待队列形成”等一系列问题。我们来看系统对这些情况都是如何处理的。

下面我们先来介绍一下实例 2 的场景。三个进程的情况如下，进程 A 是一个写盘进程，其代码是：

```
void main()
{
char str1[]="ABCEDABCDE";
//打开文件
int i;
int fd = open("/mnt/user/hello1.txt", O_RDWR ,0644));
//写文件
for(i=0;i<1000000;i++)
write(fd,str1,strlen(str1));
return;
}
```

该进程执行的目的是将 str1 这个字符数组的内容重复 1000000 次写入 hello1.txt 文件中，于是写入的数据量为 10MB。假设此时 hello1.txt 文件已经存在于硬盘中，且没有任何内容。

进程 B 也是一个写盘进程，其代码如下：

```
void main()
{
char str2[]="ABCEDABCDE";
//打开文件
int i;
int fd = open("/mnt/user/hello2.txt", O_RDWR ,0644));
//写文件
for(i=0;i<1000000;i++)
write(fd,str2,strlen(str2));
return;
}
```

该进程执行的目的是将 str2 这个字符数组的内容重复 1000000 次写入 hello2.txt 文件中，于是写入的数据量也为 10MB。假设此时 hello2.txt 文件已经存在于硬盘中，且没有任何内容。

进程 C 是一个读盘进程，其代码如下：

```
void main()
{
char buffer[200];
//打开文件
int fd = open("/mnt/user/hello3.txt", O_RDWR ,0644));
//读文件
read(fd,buffer,sizeof(buffer));
return;
}
```

该进程执行的目的是从 hello3.txt 文件中读取 200 字节的内容到 buffer 这个数组中。假设此时 hello3.txt 文件已经存在于硬盘中，且文件中有内容。

这三个进程启动顺序为：进程 A 先启动，并执行；之后进程 B 启动，并执行；最后进程 C

启动，并执行。下面我们来看具体的执行过程。

6.2.1 系统不断为进程 A 向缓冲区写入数据

进程 A 先开始执行后，执行 write 函数，通过对实例 2 的介绍我们得知，hello1.txt 文件没有任何内容，所以进程 A 只需在缓冲区中不断申请缓冲块并将指定的数据写入缓冲块就可以了，具体的步骤我们已经在 4.3 节和第 4.4 节中说明。任何一个进程，能直接将数据写入缓冲块的原则就是：该缓冲块空闲且不脏。我们假设现在系统已经在缓冲区中的所有空闲且不脏的缓冲块内写满了数据。图 6-19 显示了系统已经将所有空闲且不脏的缓冲块写满的状态。接下来，我们就要看一下，在缓冲区处于图 6-19 所示的状态时，再进行写入操作，会导致什么情况。

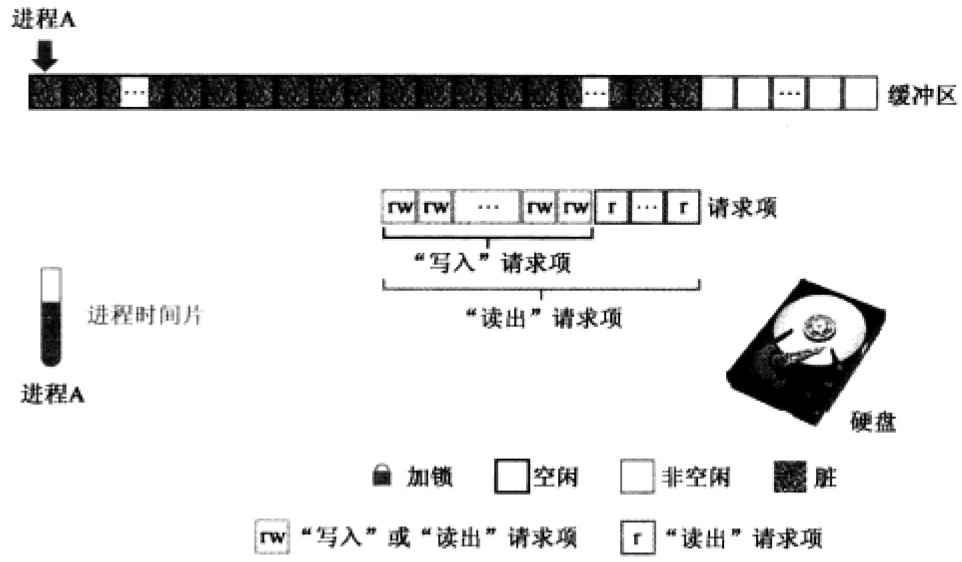


图 6-19 系统不断为进程 A 写入数据

6.2.2 继续执行引发缓冲块数据需要同步

当前进程仍然是进程 A，它提出的请求，系统还远没有完成，还要继续向缓冲区写数据，这就需要通过 getblk 函数在缓冲区中找到一个空闲的缓冲块，即 b_count 为 0 的缓冲块。

执行代码如下：

```
//代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
{
    ...
    tmp = free_list;
    do {
        if (tmp->b_count)
            continue;
        if (!bh || BADNESS(tmp)<BADNESS(bh)) {
            bh = tmp;
            if (!BADNESS(tmp))
                break;
        }
/* and repeat until we find something good */
    } while ((tmp = tmp->b_next_free) != free_list);
        ...
}
```

但现在的情况是，缓冲区中已经没有空闲且不脏的缓冲块，只有空闲且脏的缓冲块。这就意味着，接下来要通过 4.4.6 节中介绍的方式来强行将缓冲区中的数据同步到硬盘，以便在缓冲区中空出更多的空间，为后续的写盘工作提供支持，执行代码如下：

```
//代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
{
    ...
    while (bh->b_dirt) {
        sync_dev(bh->b_dev);
        ...
    }
    ...
}
```

6.2.3　将缓冲区中的数据同步到硬盘上

此时，sync_dev 这个函数用来将缓冲区中的数据同步到硬盘上，进入 sync_dev 函数后，代码如下：

```
//代码路径：fs/buffer.c：
int sync_dev(int dev)
{
    ...
    bh = start_buffer;
    for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
        if (bh->b_dev != dev)
            continue;
        wait_on_buffer(bh);
        if (bh->b_dev == dev && bh->b_dirt)
            ll_rw_block(WRITE,bh);
    }
    ...
}
```

sync_dev 函数将会遍历整个缓冲区，将所有“脏”的缓冲块中的内容全部同步到硬盘上。每个“脏”缓冲块的同步步骤是这样的：

第一，先将缓冲块与申请到的空闲请求项进行绑定，请求项中记录的内容将作为数据同步的唯一依据。

第二，如果硬盘此时没有工作，则下达写盘命令，将数据进行同步；如果硬盘正在工作，则将该请求项挂接在请求项队列中，硬盘同步完数据并产生中断后，中断服务程序会不断地给硬盘发送指令，以使请求项队列中各个请求项对应的数据陆续同步硬盘中。

sync_dev 函数将会不停地执行上述工作，直到无法再申请到空闲请求项为止。6.2.3 节中讲述的就是这一过程。

每个缓冲块同步的过程都是在 ll_rw_block 函数中完成的。在此过程中，缓冲块会被加锁，加锁只是阻止进程与缓冲块的数据交互，阻止系统自身与缓冲块的数据交互，但并不阻止缓冲块与硬盘之间的数据交互，其目的是确保缓冲块与硬盘之间的数据交互的正确性。在发送同步

指令之前，需要同步的缓冲块的脏标志 b_dirt 将会被设置为 0，以表示它不再是“脏”的缓冲块了。

具体的执行路线是：进入 ll_rw_block 函数后会调用 make_request 函数将缓冲块与请求项挂接；在 make_request 函数中会先将缓冲块加锁，并通过 add_request 函数加载请求项；在请求项加载完毕后，系统将通过调用 do_hd_request 函数向硬盘发送写盘命令。 do_hd_request 函数是系统与硬盘交互的底层函数，该函数已经没有任何策略可言，它将根据请求项的数据，最终实现将指定的缓冲块数据写到指定的硬盘块上，执行代码如下：

```
// 代码路径：kernel/blk_dev/ll_rw_block.c：
static void make_request(int major,int rw, struct buffer_head * bh)
{
    ...
    lock_buffer(bh);
    ...
    add_request(major+blk_dev,req);
}
static void add_request(struct blk_dev_struct * dev, struct request * req)
{
    ...
    if (req->bh)
    req->bh->b_dirt = 0;
    ...
    (dev->request_fn)();// 这行代码对应的就是 do_hd_request 函数
    ...
}
```

同步一个缓冲块所表现的情况如图 6-20 所示，由于在 make_request 中把这个缓冲块加锁了，而在 add_request 函数中已经将该缓冲块的脏标志置 0，因此这个缓冲块已经成为了空闲且不脏的缓冲块，请读者将此图与图 6-19 对比，注意该缓冲块的状态变化：

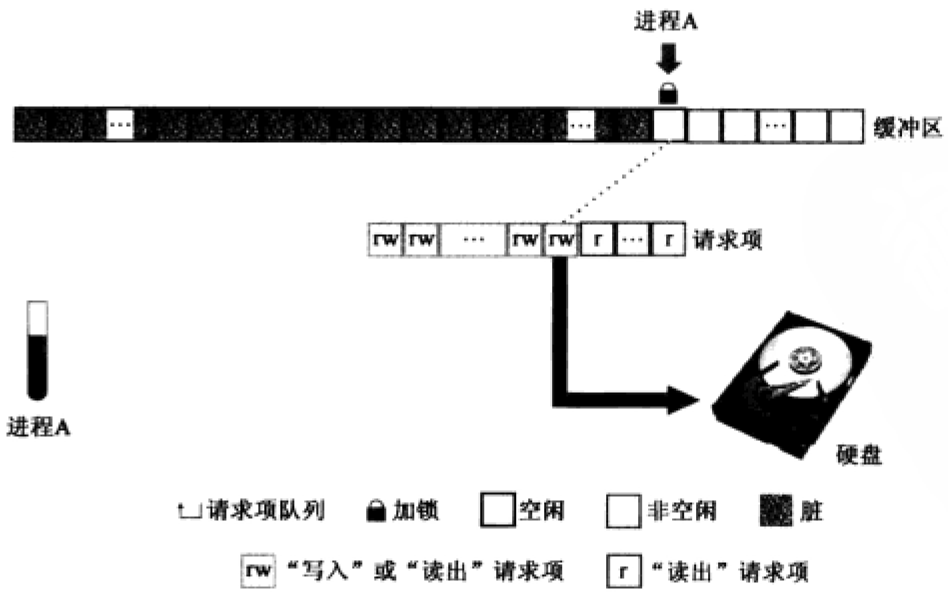


图 6-20　将写请求插入到请求项队列

sync_dev 函数不断同步缓冲块的最终结果如图 6-21 所示，注意所有留给写入操作的请求项均已被占用，同时与写入请求项对应的缓冲块的状态也已被置为空闲且不脏的状态，而硬盘正在

不断对请求项进行处理。

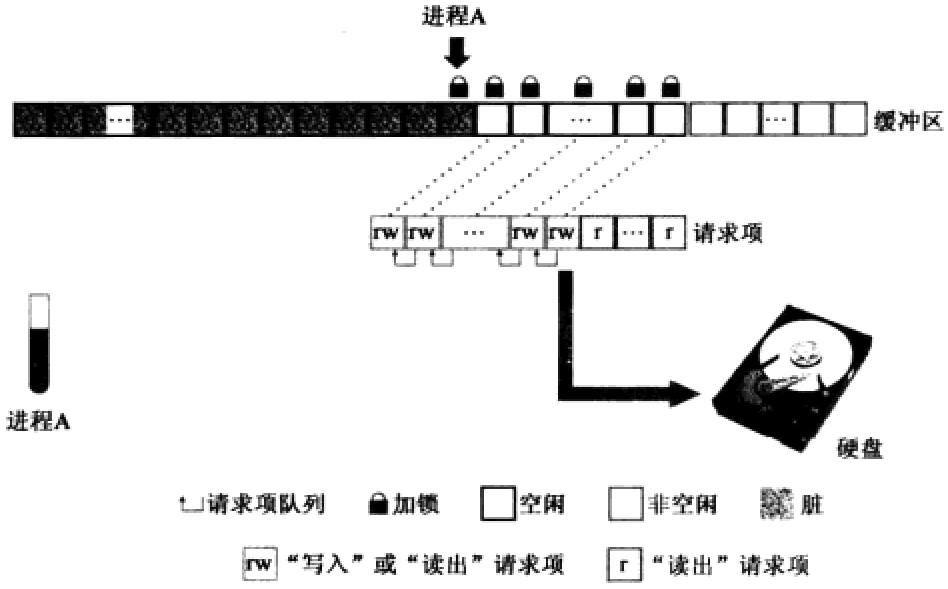


图 6-21　请求项结构中供写请求使用的空间已用完

在请求项结构中虽然还有空闲的请求项，但留给“写入”操作的请求项只占请求项总数的 2/3，对应的代码如下：

```
// 代码路径：kernel/blk_dev/ll_rw_block.c：
static void make_request(int major,int rw, struct buffer_head * bh)
{
    ...
    if (rw == READ)
    req = request+NR_REQUEST;
    else
    req = request+((NR_REQUEST*2)/3);
    ...
}
```

由于现在这 2/3 的请求项已经全部被占用了，所以现在已经没有空闲的请求项为“同步”服务了，如图 6-21 中的写入请求项所示。

6.2.4　进程 A 由于等待空闲请求项而被系统挂起

空闲的“写盘”请求项没有了，但 sync_dev 函数仍然会被继续调用，再次进入 make_request 函数后，将会执行如下代码：

```
// 代码路径：kernel/blk_dev/ll_rw_block.c：
static void make_request(int major,int rw, struct buffer_head * bh)
{
    ...
    while (--req >= request)
    ...
    if (req < request) {
        ...
        sleep_on(&wait_for_request);
    }
    ...
}
```

这些代码的作用是，如果最后也没有找到合适的空闲请求项，就将当前进程挂起。调用 sleep_on 函数后，进程 A 就成为了等待空闲请求项的进程，它被挂起了。该过程如图 6-22 所示，请大家注意，此时硬盘仍然在不断地处理请求项，而进程 A 已经处于挂起状态，尽管它还有时间片。

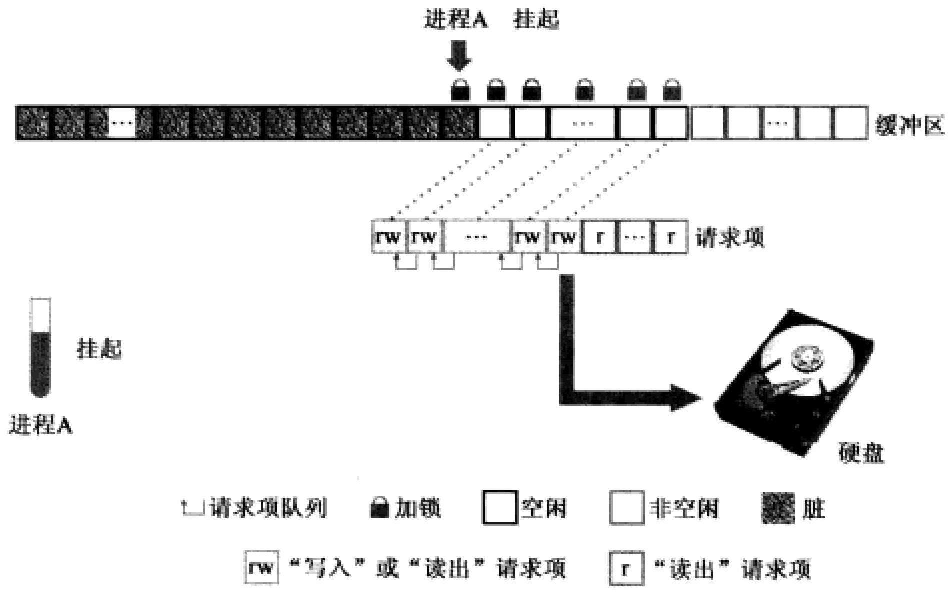


图 6-22 进程 A 被挂起

6.2.5 进程 B 开始执行

这里我们要介绍的内容是，进程 B 开始执行后系统如何为它申请缓冲块以及如何处理它的执行状态。

进程 B 启动并执行，通过实例 2 的介绍我们得知，进程 B 也是一个写盘进程，于是系统也要给进程 B 申请缓冲块，以便其写入数据。缓冲块并不是属于某一个进程的资源，它是属于操作系统的资源，因此只有操作系统自身才有资格为每一个进程申请缓冲块，不同的情况下，申请什么样的缓冲块更合适，都是由系统决定的，进程无权干预。下面我们就来介绍系统将如何为进程 B 申请缓冲块，执行代码如下：

```
// 代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
{
    ...
    tmp = free_list;
    do {
        if (tmp->b_count)
            continue;
        if (!bh || BADNESS(tmp)<BADNESS(bh)) {
            bh = tmp;
            if (!BADNESS(tmp))
                break;
        }
/* and repeat until we find something good */
    } while ((tmp = tmp->b_next_free) != free_list);
    ...
}
```

通过图 6-22 可以看出，现在缓冲区中的各个空闲缓冲块的状态有所不同，系统就是要在当前情况下综合分析所有空闲缓冲块的状态，从而确定为进程 B 申请哪个缓冲块。系统是通过 BADNESS(tmp) 来进行综合分析的，BADNESS(tmp) 的定义如下：

```
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock),
```

它的作用是将缓冲区中所有的缓冲块按照使用的便利性分为 4 个等级，从优到劣依次为：

1 等：没有“脏”，且没有“加锁”的空闲缓冲块，此等缓冲块的 BADNESS 值为 0；

2 等：没有“脏”，且有“加锁”的空闲缓冲块，此等缓冲块的 BADNESS 值为 1；

3 等：有“脏”，且没有“加锁”的空闲缓冲块，此等缓冲块的 BADNESS 值为 2；

4 等：即有“脏”且有“加锁”的空闲缓冲块，此等缓冲块的 BADNESS 值为 3。

BADNESS 值越小，则该缓冲块就越方便使用，越大就越不方便使用。

通过 6.2.3 节的介绍我们得知，系统已经将一些缓冲块加锁了，并将它们的“脏”标志设置为 0，于是就让它们的 BADNESS 值变为了 1。在目前的情况下，这已经是最方便使用的缓冲块了，因此系统就为进程 B 申请到了一个 BADNESS 值为 1 的缓冲块，这个缓冲块是加锁的，这意味着该缓冲块并不能马上被操作，但并不影响进程 B 得到这个缓冲块，只要缓冲块是空闲的就可以得到。图 6-23 表示了系统给进程 B 申请到的缓冲块：

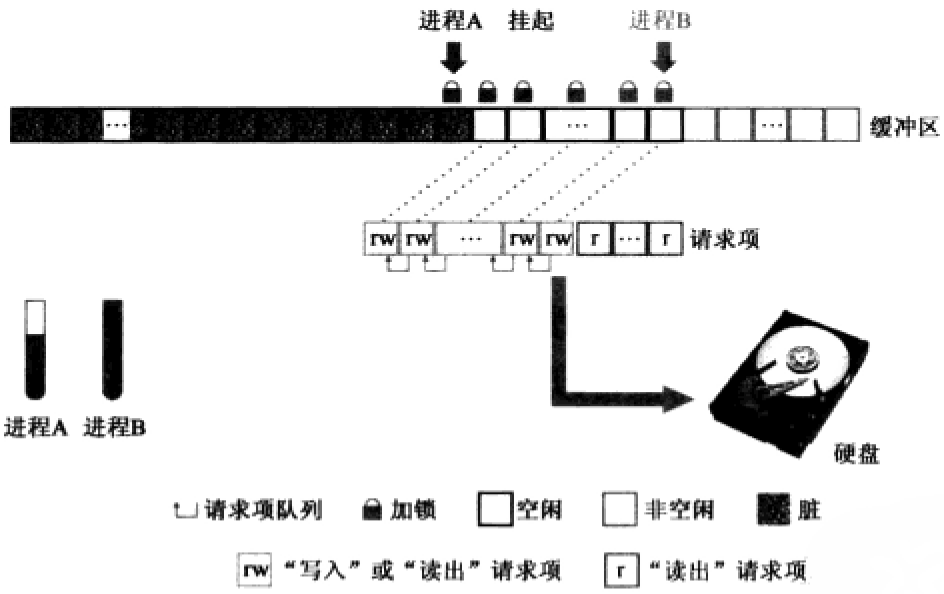


图 6-23 系统为进程 B 申请到缓冲块

6.2.6 进程 B 也被挂起

由于系统申请到的缓冲块是“加锁”的缓冲块，这就导致进程或系统不能立即与该缓冲块进行数据交互，否则将产生数据混乱。于是，系统会直接调用 wait_on_buffer(bh); 函数，进程 B 将会被挂起，如图 6-24 所示。注意，此时系统和硬盘还在不断处理请求项。

执行代码如下：

```
// 代码路径：fs/buffer.c：
struct buffer_head * getblk(int dev,int block)
```

```
{
    ...
    wait_on_buffer(bh);
    ...
}
```

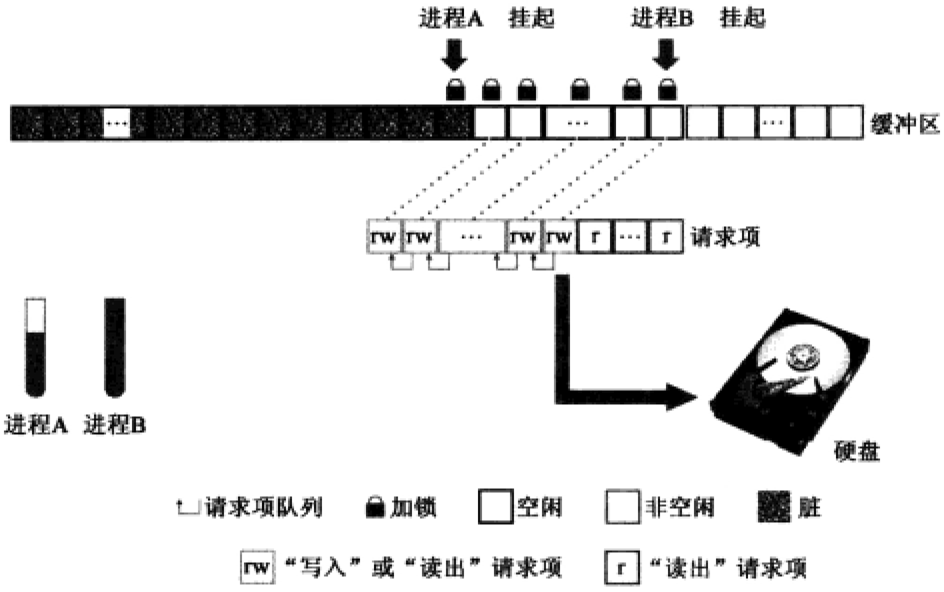


图 6-24 进程 B 也被挂起

6.2.7 进程 C 开始执行并随后被挂起

这里我们要介绍的内容是，进程 C 开始执行后系统如何为它申请缓冲块以及如何处理它的执行状态。

进程 C 启动并执行，通过对实例 2 的介绍我们得知，进程 C 是一个读盘进程，系统也要为它申请一个缓冲块，申请的方式我们在 6.2.5 节中已经讲解，而且从现在缓冲区的情况来看，系统给进程 C 和进程 B 申请的应该是同一个缓冲块，通过 6.2.6 节的介绍我们得知，这个缓冲块是加锁的，该缓冲块同样不能被操作，但可以被申请到，于是进程 C 也会被挂起，理由在 6.2.6 节中已经说明。

进程 C 和进程 B 现在都因为在等待同一个缓冲块的解锁而被挂起，所以这两个进程现在就形成了一个进程等待队列。

到现在为止，实例 2 中的三个用户进程都被挂起了，于是默认切换到进程 0 去执行。它们被挂起的情况如图 6-25 所示，其中进程 A 处在等待空闲请求项的等待队列，而进程 B 和进程 C 处在等待同一个缓冲块解锁的队列。

6.2.8 进程 A 和进程 C 均被唤醒

下面我们将介绍这三个用户进程被唤醒的过程，它们被唤醒后，系统将继续根据缓冲块和请求项的各方面状况来决定用户进程将如何执行。

一段时间之后，硬盘完成了请求项交付的同步工作，产生硬盘中断，中断服务程序开始执行，执行代码如下。

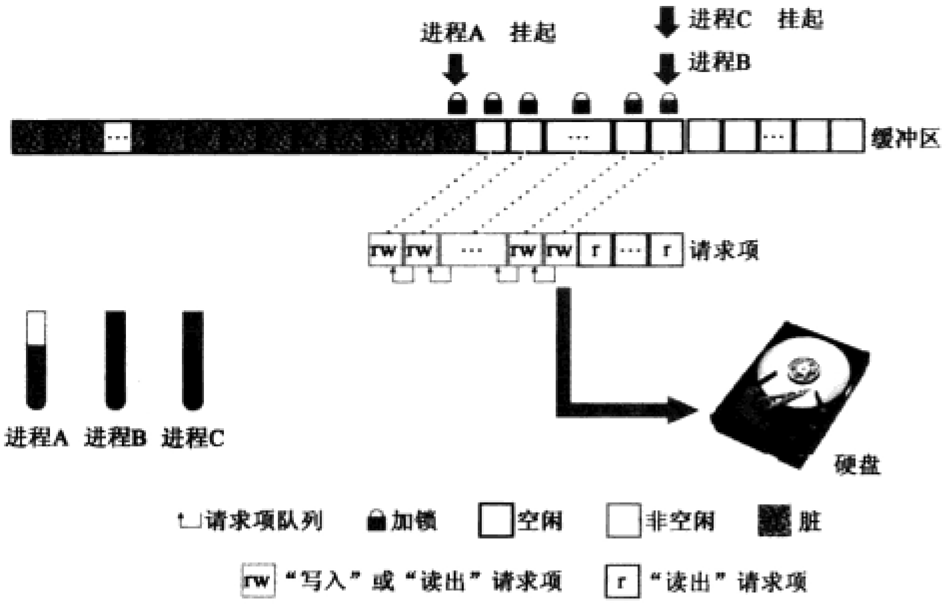


图 6-25 进程 A、B 和 C 此时的运行状态和所处等待队列

```
// 代码路径：kernel/blk_dev/blk.h：
extern inline void end_request(int uptodate)
{
      ...
      unlock_buffer(CURRENT->bh);
      ...
      wake_up(&wait_for_request);
      ...
}
extern inline void unlock_buffer(struct buffer_head * bh)
{
      ...
      bh->b_lock=0;
     wake_up(&bh->b_wait);
}
```

从 6.2.4 节中的内容我们得知，进程 A 是由于等待空闲请求项才被挂起的，于是 wake_up(&wait_for_request); 这行代码将唤醒进程 A，如图 6-26 左下方的进程 A 所示。

从 6.2.7 节中的内容我们得知，进程 B 和进程 C 构成了一个进程等待队列，而且进程 C 是队列的队首（理由我们在 6.1 节已经介绍过了），它们在等待共同的缓冲块，于是 wake_up(&bh->b_wait); 这行代码唤醒了队首，即进程 C，如图 6-26 左下方的进程 C 所示。

另外，由于指定的缓冲块中的数据已经操作完毕了，所以中断服务程序还会将该缓冲块解锁。中断服务程序会在以上工作做完后继续调用 do_hd_request 函数，如果还有请求项需要处理，将继续像 6.2.3 节中介绍的那样，再次给硬盘发送写盘指令，很显然，硬盘又要开始进行后续的同步工作了。从图 6-26 中的右半部分可以看出，硬盘已经在处理下一个请求项，而此刻已经有一个可用的“写”请求项空闲。

通过图 6-26 我们可以看到，此时进程 C 的时间片多于进程 A 的时间片，所以系统将当前进程由进程 0 切换到进程 C，开始执行后的第一件事就是要唤醒进程 B，如图 6-27 的左下方所示。理由我们在 6.1 节中介绍进程等待队列的唤醒时已经说明，执行代码如下。

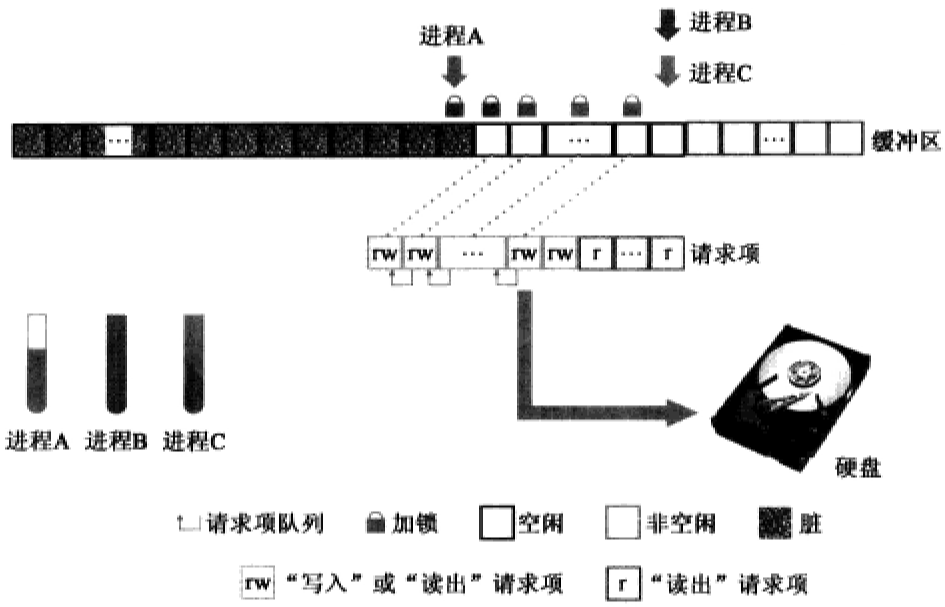


图 6-26 一个缓冲块数据同步完成后各进程的状态

```
//代码路径：fs/buffer.c:
struct buffer_head * getblk(int dev,int block)
{
    ...
    wait_on_buffer(bh);
    ...
}
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    sti();
}
```

系统在 6.2.7 节中为进程 C 申请到的缓冲块现在已经被解锁了，可以使用了。系统将先对这个缓冲块进行设置，包括将它的引用次数设置为 1，于是该缓冲块就不再是空闲缓冲块了。然后，在请求项结构中申请一个空闲请求项与此缓冲块绑定，并将该缓冲块再次加锁。但是，请求项设置完毕，并不代表马上就能处理这个请求项。此时硬盘正忙着同步处理其他请求项呢，现在只能先将"读盘"请求项插入到请求项队列中，然后系统会将进程 C 挂起。此时系统中的进程状态如图 6-27 所示，其中读请求项占据的是请求项数组的最后一项，并且由请求项数组中第 1 项的指针指向它。

6.2.9 进程 B 切换到进程 A 执行

进程 C 挂起后，进程 B 的时间片显然比进程 A 要多，所以切换到进程 B。在 6.2.5 节中，系统早已经为进程 B 申请了缓冲块，当时由于这个缓冲块是加锁的，所以进程 B 要挂起。现在，这个缓冲块仍然是加锁的，所以进程 B 将再次被挂起，并切换到进程 A。如图 6-28 所示。

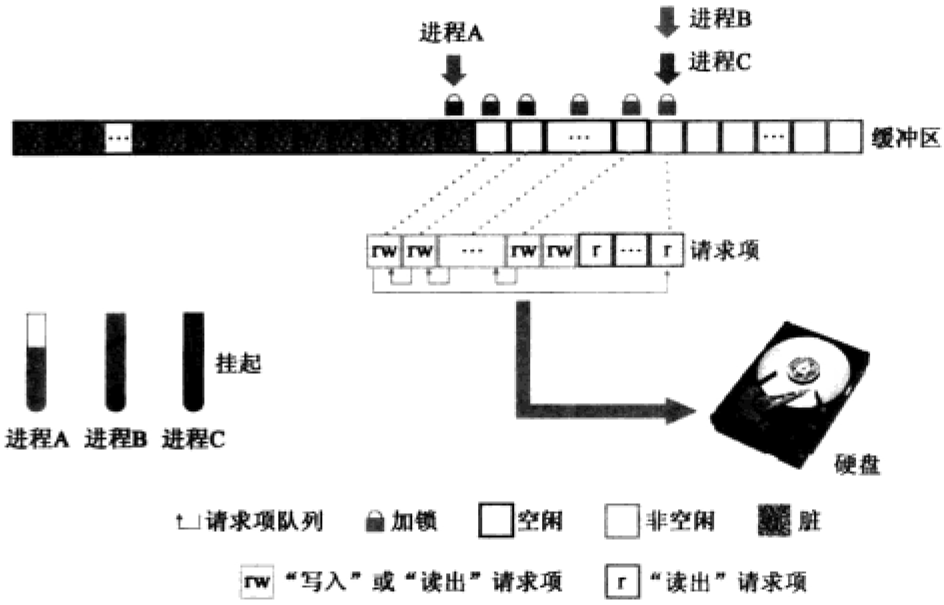


图 6-27　进程 C 将读请求插入到请求项队列后各进程的状态

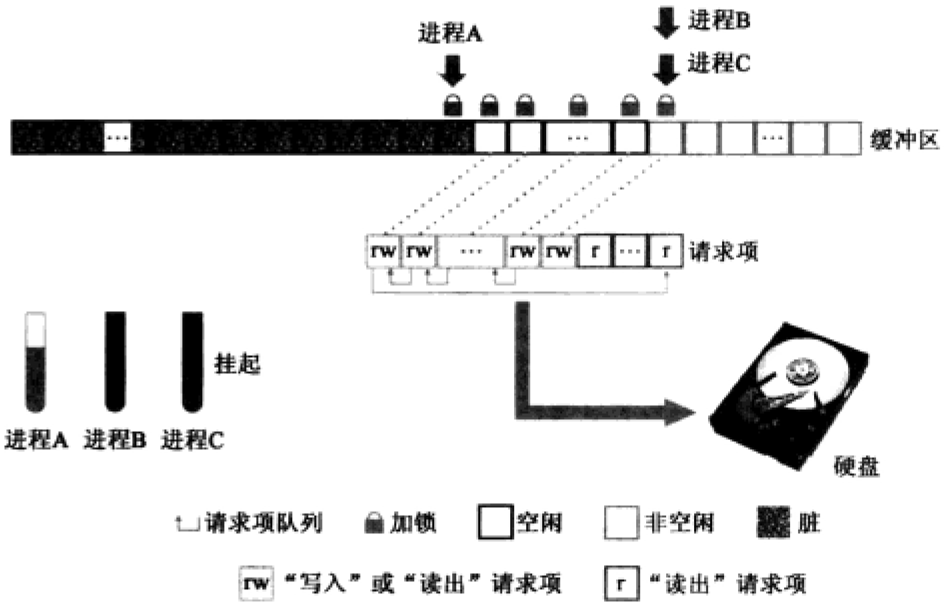


图 6-28　进程 B 挂起，切换到进程 A 执行

现在的当前进程是进程 A，进程 A 在 6.2.4 节中是由于同步缓冲块时缺少空闲请求项才被挂起的，所以它被唤醒后，系统将继续同步缓冲块。通过图 6-28 可以看出，现在系统已经有空闲的请求项用于写盘了，所以系统将此请求项与即将要同步的缓冲块相绑定，并插入到请求项队列中，之后又没有空闲的请求项了，进程 A 将再次被挂起，如图 6-29 所示。

接下来，以上步骤将会重复执行，即一方面只要硬盘执行完一次同步操作，就会释放一个请求项，并将其对应的缓冲块解锁，这些都将导致等待空闲请求项或等待缓冲块解锁的进程被唤醒；另一方面，被唤醒的进程又不断地引发缓冲区与硬盘之间进行数据交互，从而使这些进程不断地被挂起。如图 6-30 所示。

直到最后三个进程全部完成各自的写盘任务和读盘任务。如图 6-31 所示。

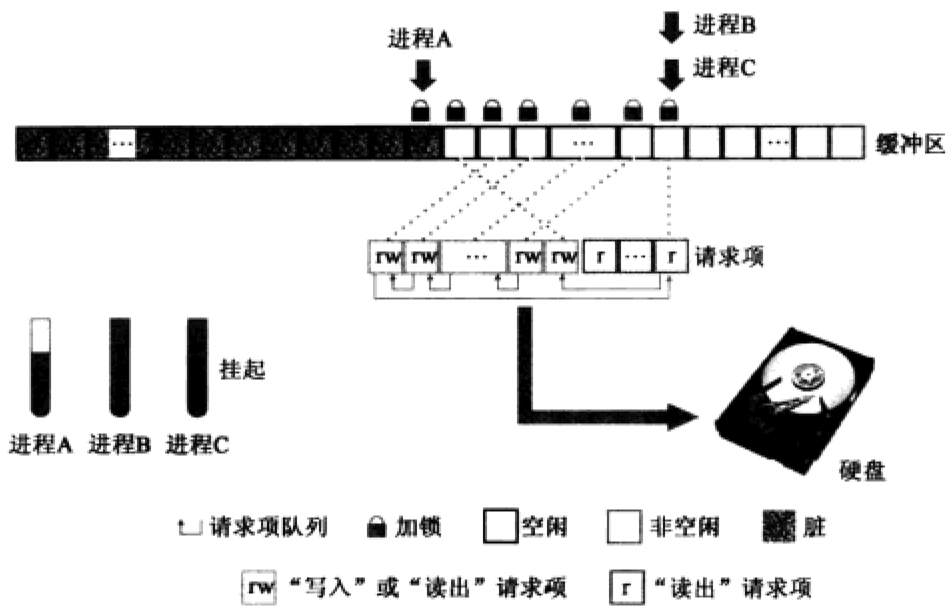


图 6-29　进程 A 也再次挂起

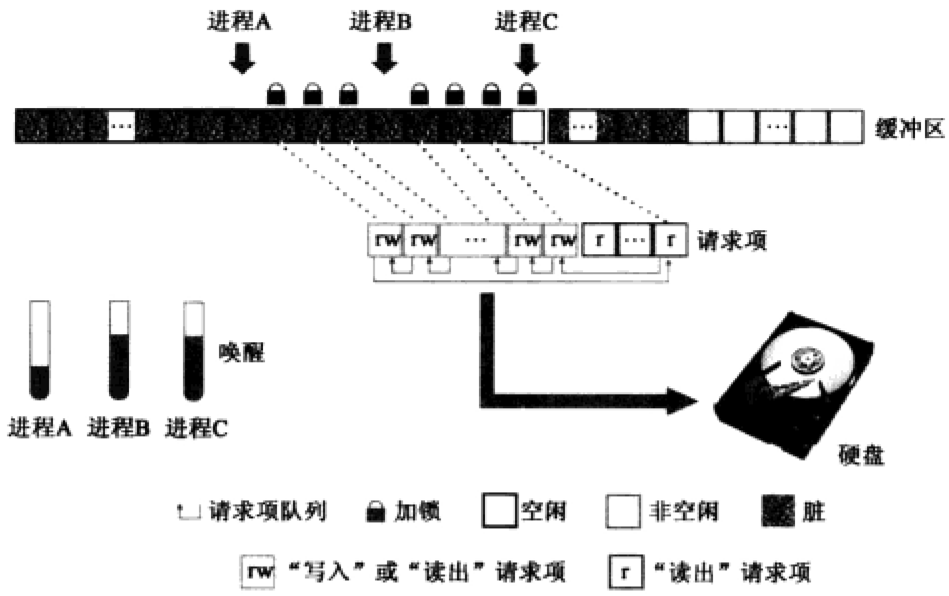


图 6-30　缓冲块再次同步完成时唤醒进程 B

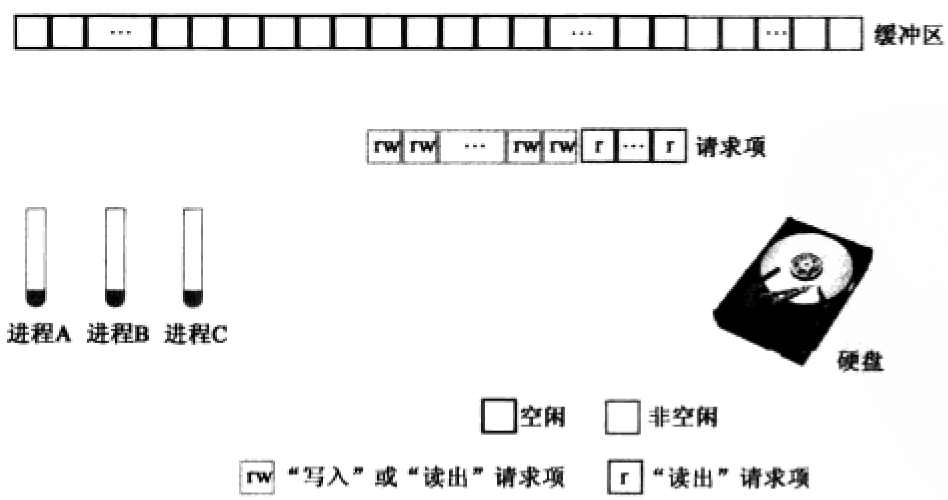


图 6-31　进程所有请求均已处理完毕

6.3　小结

本章以“多进程操作一个文件”和“多进程操作多个文件”为例，对多进程操作文件进行了

讲解。

多个进程在操作同一个文件过程中，由于需要等待着同一个缓冲块的解锁而被操作系统一一挂起，形成了进程等待队列。缓冲块解锁后，等待队列中的各个进程又以相反的顺序被操作系统一一唤醒。

多个进程在操作多个文件的过程中，缓冲块很快被占用完毕，操作系统将缓冲区中的数据同步到硬盘上，需要等待缓冲块解锁的进程也将被挂起。由于请求项的数量有限，所以请求项将很快被占用完毕，需要等待空闲请求项的进程也将被挂起。每同步完一个缓冲块上的数据，都会出现可用的缓冲块和空闲的请求项，导致某些等待的进程被唤醒，操作系统继续为解锁的进程申请合适的缓冲块和空闲的请求项，直到所有文件操作完毕。

第 7 章

IPC 问题

通过前面几章的内容我们得知，在 Linux 0.11 中是不允许进程跨越自己的“边界”去影响其他进程执行的，这也是操作系统保护模式的核心内容。但是，从进程实际的应用角度讲，它们往往需要协同工作，具体表现为，共同操作同一内存中的数据来实现自己的执行目的。比如说，一个进程往内存某处写入数据，另一个进程从该处读取数据，以此来影响进程的执行和状态。

针对进程的这种执行需求，Linux 0.11 设计了两套机制来为此需求提供服务：一套是“管道机制”，一套是“信号机制”，本章将通过两个实际的应用案例来对这两套机制进行详细的介绍。

7.1 管道机制

在 Linux 0.11 中，管道就是一个允许两个进程共同操作同一个内存页面的方式。这两个进程，一个往该页面上写入数据，另一个从该页面上读出数据，以此来完成各自的执行目的。操作系统将在此过程中对两个进程的执行进行组织、管理与协调，从而保证它们在操作同一个页面时不会产生数据混乱。下面我们先来看实例 1 中对管道的操作，代码如下所示：

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int         n, fd[2];
    pid_t       pid;
int  i,j;
char str1[]="ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
```

```
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDEABCDE
ABCDEABCDEABCDEABCDEABCD";

char str2[512];

    if (pipe(fd) < 0){ //创建管道
        printf("pipe error\n");
        return -1;
    }

    if ( (pid = fork()) < 0){
        printf("fork error\n");
        return -1;
    }

    else if (pid > 0) {// 父进程向管道中写入数据
        close(fd[0]);
for(i=0;i<10000;i++)
      write(fd[1],str1,strlen(str1));
      }
      else {    // 子进程从管道中读取数据
        close(fd[1]);
for(j=0;j<20000;j++)
        read(fd[0],str2,strlen(str2));
    }
    return 0;
}
```

我们在实例 1 中展现了“父子”两进程共同操作一个管道的场景：父进程先创建一个管道，之后，父进程不断往管道中写入数据，子进程不断从管道中读出数据。下面我们将参照实例 1 的场景，先来介绍管道的创建过程，之后，再来介绍这两个进程是如何共同操作管道的，系统又是如何对此进行组织管理协调的。

本节的程序架构图如图 7-1 所示：

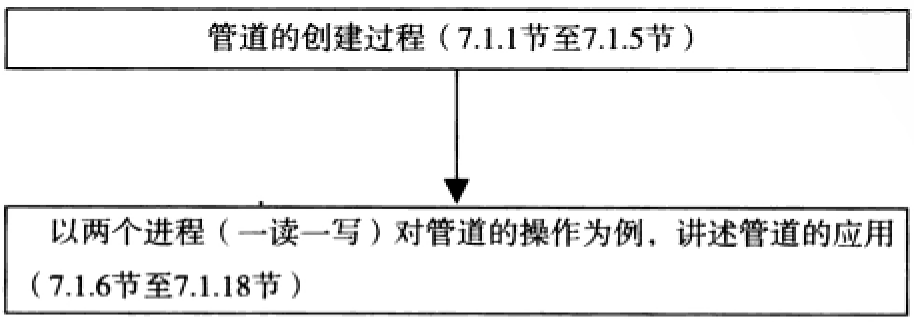


图 7-1 管道操作程序架构图

7.1.1 为管道文件在文件管理表中申请空闲项

本章开始部分已经对管道的本质进行了介绍，管道的本质实际上就是一块内存页面，系统将申请一块内存页面作为管道，然后把这个内存页面当做一个“已有的文件”进行设置。这样的设置将致使在进程看来，操作管道就是在操作一个“管道文件”。它这样设置的理由，我们将在第 8 章中介绍。

现在我们来看创建管道的过程：内核是按照创建“管道文件”的模式创建“管道”的，其他类型的文件都是为“一个进程”的使用而创建的，而管道文件则“天然地”就是为了“两个进程（即读管道进程和写管道进程）”的使用而创建的。所以，在创建时，先在内核文件管理表 file_table[64] 中申请“两个”空闲项，并将这两个空闲项的引用次数设置为 1，表示它们被引用了，这两个空闲项将作为实例 1 中父子进程所要操作的“管道文件”的文件句柄，以便这两个进程能够操作管道文件，执行代码如下：

```
// 代码路径：fs/pipe.c：
    int sys_pipe(unsigned long * fildes)
{
    ...
    for(i=0;j<2 && i<NR_FILE;i++)
        if (!file_table[i].f_count)
            (f[j++]=i+file_table)->f_count++;
    ...
}
```

为创建管道文件而在文件管理表中申请的两个空闲项如图 7-2 所示：

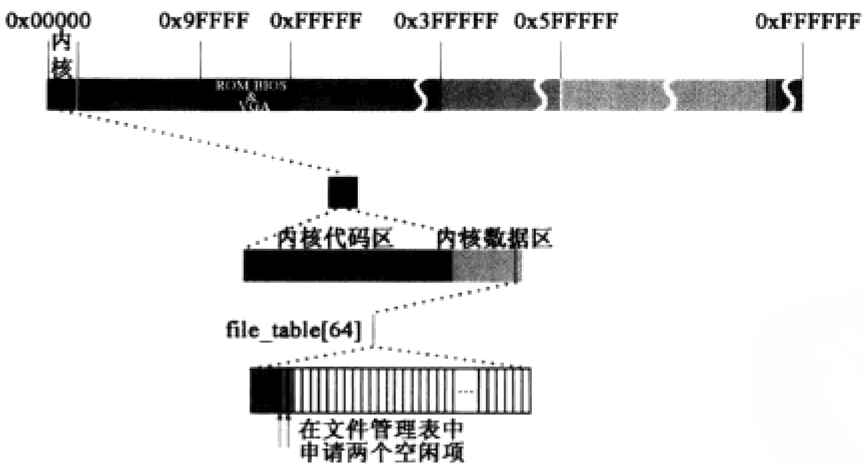


图 7-2 为创建管道文件在文件管理表中申请两个空闲项

7.1.2 为管道文件与进程建立联系创造条件

以 file_table[64] 中这两个表项为基础，在当前进程管理结构的文件管理结构 *filp[20] 中申请两个空闲项，分别与系统中的文件管理表 file_table[64] 的这两个表项相挂接，这样，当前进程文件结构 *filp[20] 中就有两个表项与 file_table[64] 建立关系了，等到它作为父进程创建子进程时，*filp[20] 中的这两个表项就会自然而然地拷贝给它的子进程，使之也“天然地”和 file_table[64] 结构中同样的管道文件表项建立了关系，我们所说的这对父子进程就是将来要操作管道的“写管

道进程”和“读管道进程”，执行代码如下：

```
// 代码路径：fs/pipe.c：
    int sys_pipe(unsigned long * fildes)
{
    ...
    for(i=0;j<2 && i<NR_OPEN;i++)
        if (!current->filp[i]) {
            current->filp[ fd[j]=i ] = f[j];
            j++;
        }
    ...
}
```

图 7-3 表示了将当前进程与管道文件建立关联的效果：

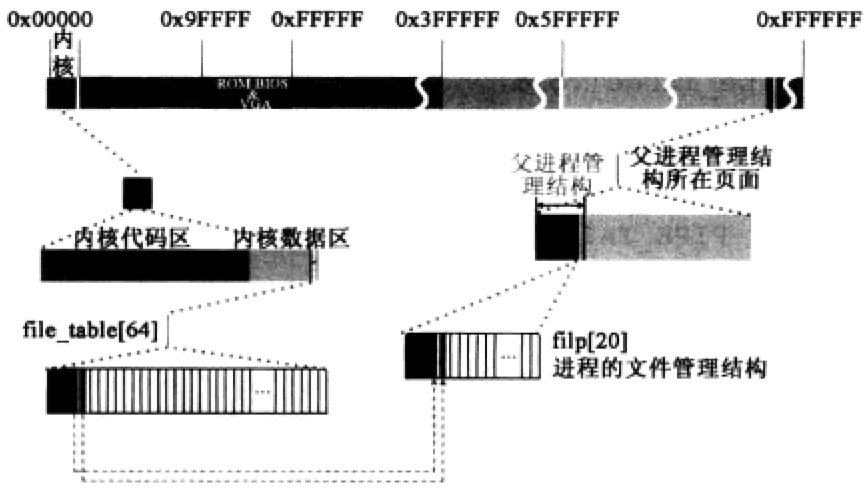


图 7-3 将当前进程与管道文件建立联系

7.1.3 创建管道文件 i 节点

通过 4.1 节的介绍得知，进程要想具备操作管道文件的能力，我们在 7.1.2 节中只是完成了一半的任务，还有一半就是建立管道文件的 i 节点与文件管理表的关系。于是将调用 get_pipe_inode 函数，先为该管道文件在 i 节点管理表 inode_table[32] 中申请一个 i 节点，执行代码如下：

```
// 代码路径：fs/pipe.c：
int sys_pipe(unsigned long * fildes)
{
    ...
    if (!(inode=get_pipe_inode()))
    ...
}
```

每个 i 节点都要有自己的属性信息，管道 i 节点也不例外，由于管道的本质就是一个内存页面，所以，系统要申请一个空闲的内存页面，并将该页面的地址载入 i 节点。请大家注意，此刻 inode->i_size 字段承载的不再是通常对于块设备文件来说以字节数表示的文件大小，而是内存页面的起始地址。执行代码如下：

```
// 代码路径：fs/inode.c：
struct m_inode * get_pipe_inode(void)
```

```
{
    ...
    if (!(inode->i_size=get_free_page()))
    ...
}
```

管道 i 节点也要有引用计数。在 Linux 0.11 中，默认操作这个管道文件的进程“能且仅能”有两个，一个是读进程，一个是写进程，所以这里直接设置为 2。

之后，让读管道指针和写管道指针都指向管道的（其实就是这个空闲页面的）起始位置，以便将来读写管道的进程操作，并将该 i 节点的属性设置为“管道型 i 节点”，以此来标识该 i 节点的特殊性，即它并不是实际存储在硬盘上的文件的 i 节点，它只不过是一个内存页面。执行代码如下：

```
//代码路径：fs/inode.c：
struct m_inode * get_pipe_inode(void)
{
    ...
    inode->i_count = 2;
     PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;
     inode->i_pipe = 1;
    ...
}
```

为管道文件申请 i 节点并对其进行设置的过程如图 7-4 所示。

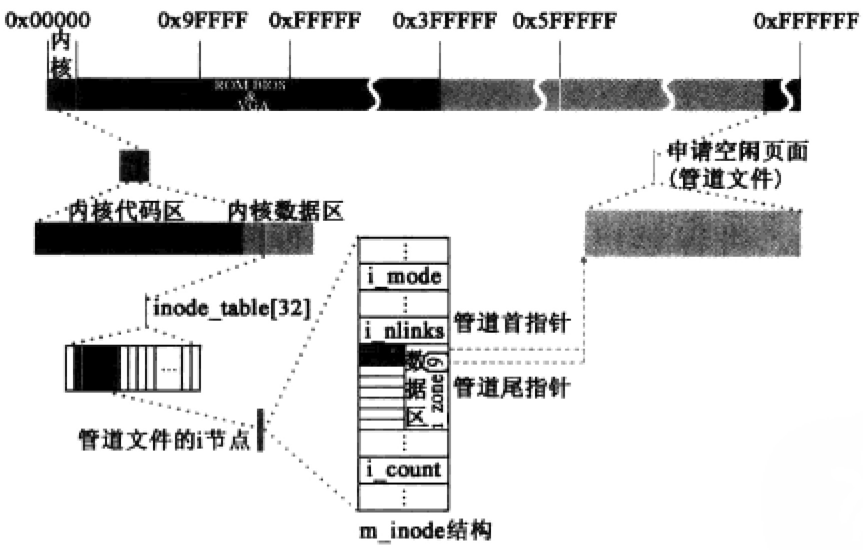


图 7-4 为管道文件创建 i 节点

7.1.4 将管道文件 i 节点与文件管理表建立联系

管道 i 节点已经设置完毕，现在就可以将它与文件管理表建立关系了，具体的操作是：初始化两个文件管理表中的空闲项，让它们都指向这个管道 i 节点，文件读写指针都指向管道的起始位置。第 1 个空闲项的文件模式置为读，第 2 个空闲项的文件模式置为写，这样，经过 7.1.1 节至 7.1.4 节的执行，进程已经具备了操作管道文件的能力，执行代码如下：

```
// 代码路径：fs/pipe.c：
int sys_pipe(unsigned long * fildes)
{
    ...
    f[0]->f_inode = f[1]->f_inode = inode;
    f[0]->f_pos = f[1]->f_pos = 0;
    f[0]->f_mode = 1;            /* read */
    f[1]->f_mode = 2;            /* write */
    ...
}
```

如图 7-5 所示：

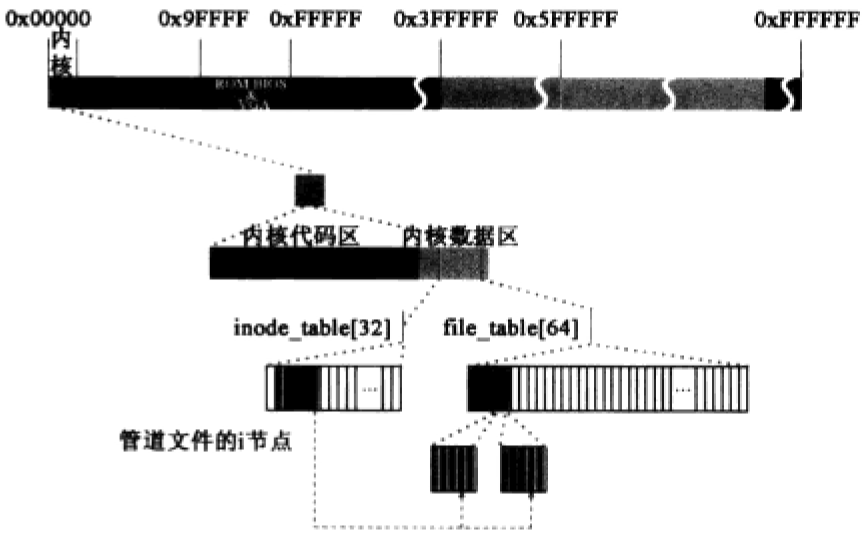


图 7-5　管道 i 节点与文件管理表建立联系

7.1.5　将管道文件句柄返回给用户进程

现在将管道文件的两个句柄返回给用户进程，即返回给实例 1 代码中的 fd[2]，这个数组有两项，每一项分别存放一个句柄，这样父子两个进程就可以通过不同的文件句柄操作这个管道文件了。执行代码如下：

```
// 代码路径：fs/pipe.c：
int sys_pipe(unsigned long * fildes)
{
    ...
    put_fs_long(fd[0],0+fildes);
    put_fs_long(fd[1],1+fildes);
    ...
}
```

将文件句柄返回给用户进程的结果如图 7-6 所示。

7.1.6　读管道进程开始操作管道文件

从这里开始，我们来介绍父子进程对管道文件的操作，以及系统对两个进程的组织管理协调。

我们不妨假设实例 1 中的读管道进程开始运行，即执行“read(fd[0], ,str2,strlen(str2))”这行源代码。read 函数会映射到系统调用函数 sys_read 中去执行，并最终执行到 read_pipe 函数中。

由于此时管道内没有任何数据，所以此时系统会将读管道进程挂起，然后切换到写管道进程中去执行，执行代码如下。

```
//代码路径：fs/pipe.c：
int read_pipe(struct m_inode * inode, char * buf, int count)
{
    ...
    while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
        ...
        sleep_on(&inode->i_wait);
    }
    ...
}
```

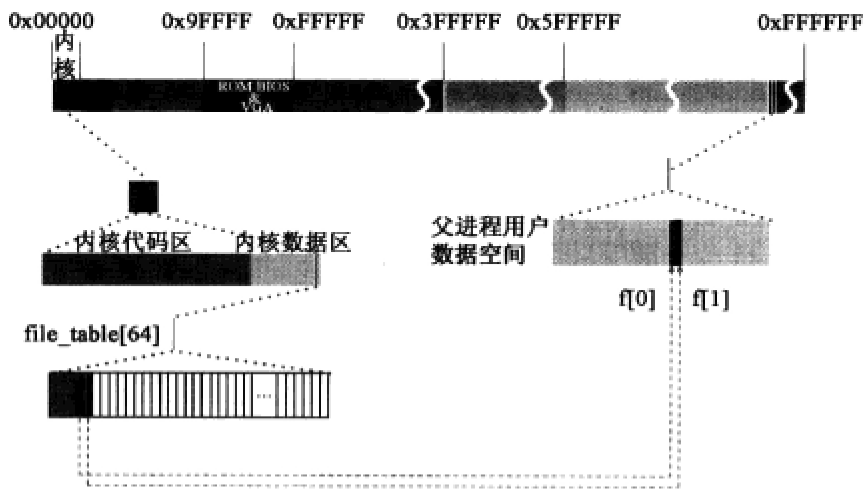


图 7-6　将管道文件句柄返回给用户进程

此时管道内没有任何数据，读管道进程进行读操作，进程的状态如图 7-7 所示：

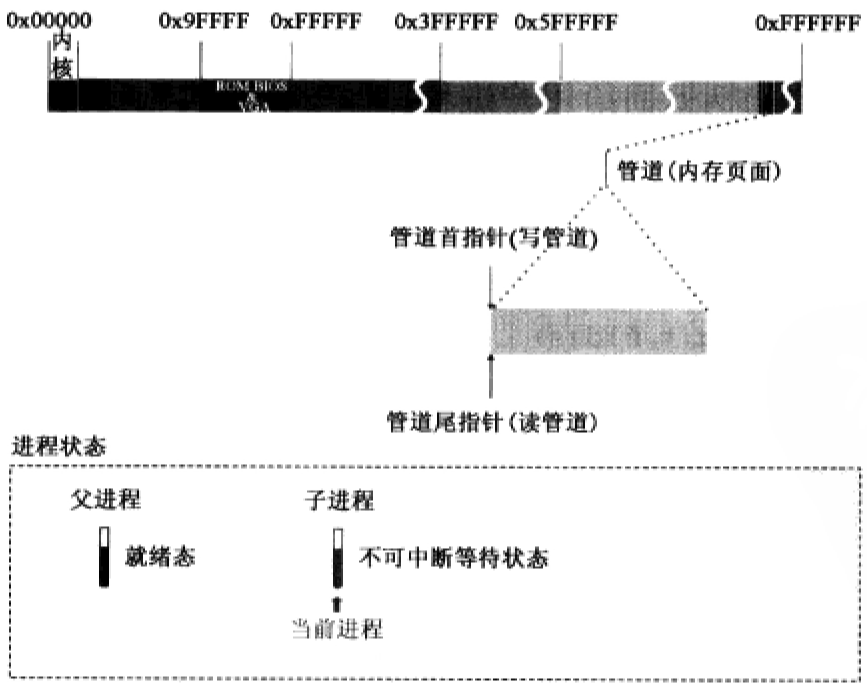


图 7-7　进程开始操作管道——挂起读进程

7.1.7　写管道进程向管道中写入数据

写管道进程开始执行，它会将实例 1 中的 str1 数组指定的 1024 字节的数据循环地写入管道

内，即执行“write(fd[1],str1,strlen(str1))”这行源代码，write 函数会映射到系统调用函数 sys_write 中去执行，并最终执行到 write_pipe 函数中。Linux 0.11 默认了一条规则：每次写管道执行完毕后，管道中就已经拥有了可以读出的数据，所以就会唤醒读管道进程；每次读管道操作执行完毕后，管道中就已经拥有了可以写入数据的空间，所以就会唤醒写管道进程。

因此，在 write_pipe 函数中，字符将被循环写入管道中，写完后管道中就已经有数据可以读出了，所以系统就会唤醒读管道进程（当然，唤醒了读管道进程并不等于读管道进程就立即执行），此次写管道操作就执行完了，执行代码如下：

```
// 代码路径：fs/pipe.c：
int write_pipe(struct m_inode * inode, char * buf, int count)
{
    ...
    while (count>0) {
    ...
    chars = PAGE_SIZE-PIPE_HEAD(*inode);
        if (chars > count)
            chars = count;
        if (chars > size)
            chars = size;
        count -= chars;
        written += chars;
        size = PIPE_HEAD(*inode);
        PIPE_HEAD(*inode) += chars;
        PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
        while (chars-->0)
            ((char *)inode->i_size)[size++]=get_fs_byte(buf++);
    }
    wake_up(&inode->i_wait);
}
```

写管道进程向管道中写入数据的过程如图 7-8 所示：

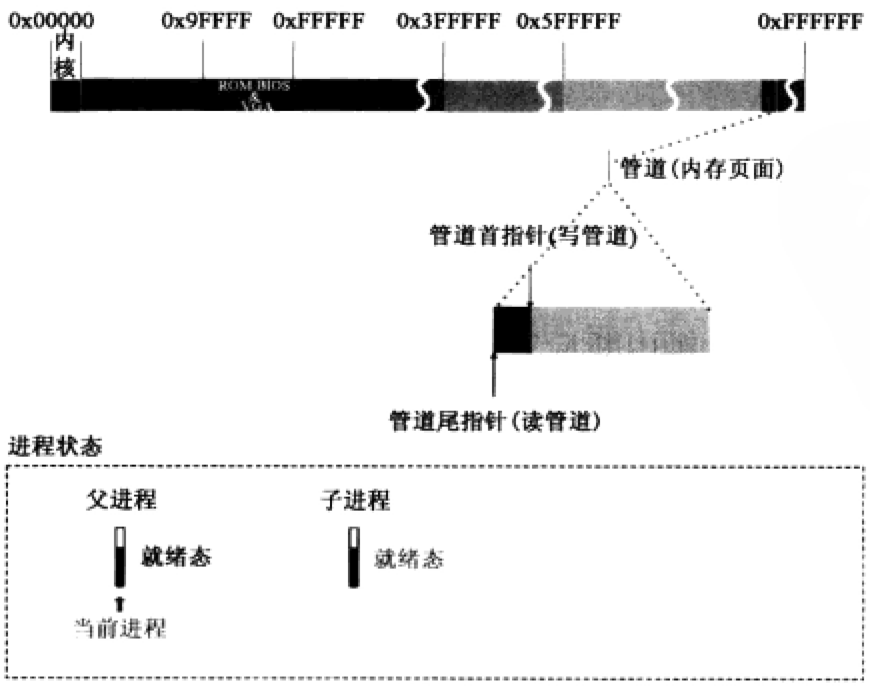


图 7-8　写管道进程向管道中写入数据

7.1.8 写管道进程继续向管道写入数据

当前进程是写管道进程，写完一次管道之后将返回用户空间，通过实例 1 中的"for (i=0; i<10000;i++)"这行代码我们得知，写管道要操作 10000 次，而写管道进程的时间片还没有用完，它还要继续执行写管道操作。

向管道中不断写入数据的过程如图 7-9 所示：

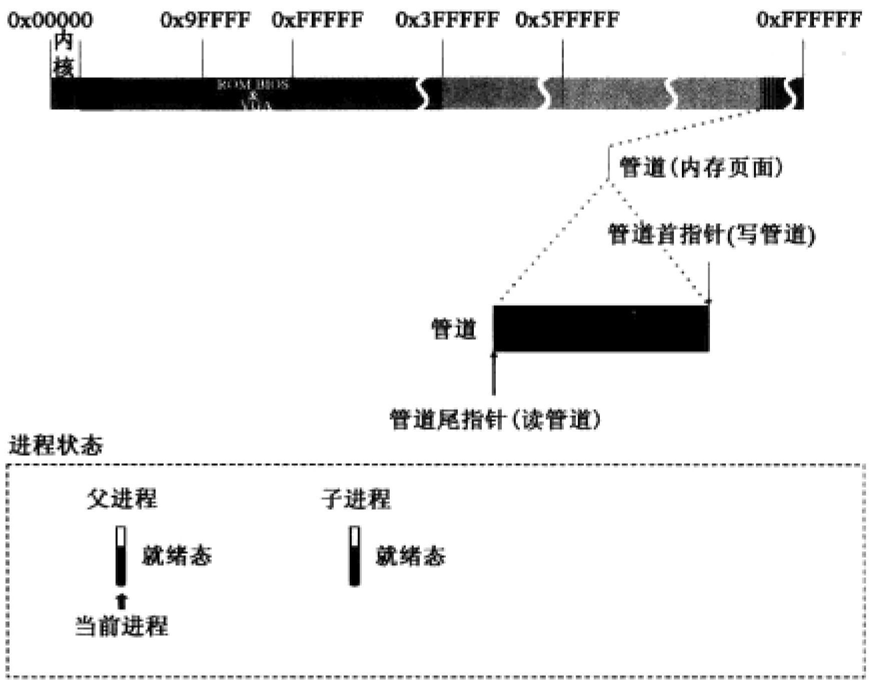


图 7-9 写管道进程不断向管道中写入数据

7.1.9 写管道进程已将管道空间写满

在写管道进程工作的过程中，一定会发生时钟中断，而时钟中断仅仅是削减了它的时间片，只要时间片不被削减为 0，它就会继续执行。对应代码如下：

```
// 代码路径：kernel/sched.c：
void do_timer(long cpl)
{
    ...
    if ((--current->counter)>0) return;
    ...
}
```

图 7-10 的下方表示了发生时钟中断后对写管道进程的影响。

直到写管道进程把管道写满为止，在写入的过程中，写管道指针一直指向数据的写入位置，一直向管道尾端移动，写满后，指针已经从管道的首端移动至尾端了，如图 7-10 的右下角所示，标识写管道的管道首指针已经移动到了尾端。

7.1.10 写管道进程挂起

再次执行写管道操作时，系统会将写管道指针从管道尾端移动至管道首端，继续将数据写入管道，执行代码如下：

```
// 代码路径：fs/pipe.c：
int write_pipe(struct m_inode * inode, char * buf, int count)
{
    ...
    size = PIPE_HEAD(*inode);
    PIPE_HEAD(*inode) += chars;
    PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
    ...
}
```

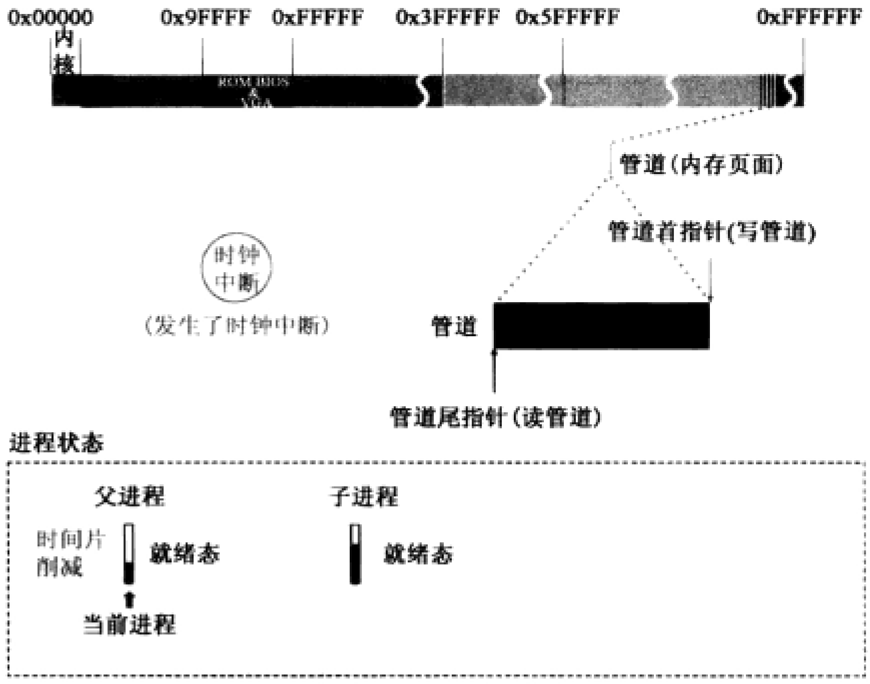


图 7-10　写管道执行过程中产生时钟中断

但是，此时管道中已经没有空间可以写入数据了，所以系统就要将写管道进程挂起。然后切换到读管道进程去执行，执行代码如下：

```
// 代码路径：fs/pipe.c：
int write_pipe(struct m_inode * inode, char * buf, int count)
{
    ...
    while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
            wake_up(&inode->i_wait);
            ...
            sleep_on(&inode->i_wait);
    }
    ...
}
```

写管道进程挂起并切换到读管道进程的过程如图 7-11 所示。

7.1.11　读管道进程从管道中读出数据

通过 7.1.6 节中的介绍我们得知，读管道进程将继续在 read_pipe 函数中执行，此次执行将会把管道中 512 字节的数据读入到读管道进程的用户空间内，执行代码如下：

```
// 代码路径：fs/pipe.c：
int read_pipe(struct m_inode * inode, char * buf, int count)
```

```
{
...
while (count>0) {
chars = PAGE_SIZE-PIPE_TAIL(*inode);
        if (chars > count)
            chars = count;
        if (chars > size)
            chars = size;
        count -= chars;
        read += chars;
        size = PIPE_TAIL(*inode);
        PIPE_TAIL(*inode) += chars;
        PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
        while (chars-->0)
            put_fs_byte(((char *)inode->i_size)[size++],buf++);
}
...
}
```

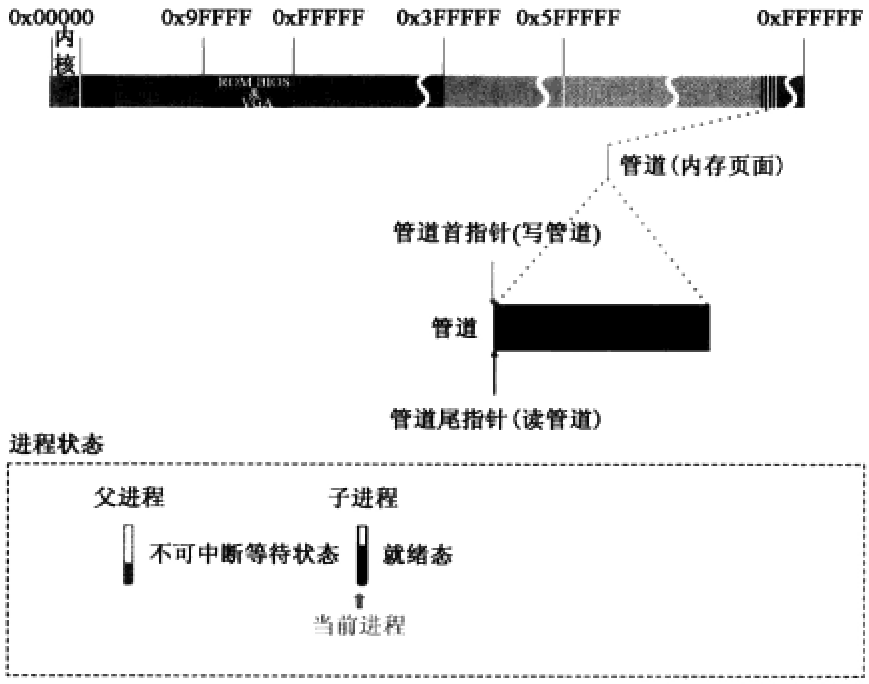


图 7-11 写管道进程挂起并唤醒读管道进程

在 7.1.7 节中，我们已经介绍过 Linux 0.11 默认的一条规则：每次写管道执行完毕后，管道中就已经拥有了可以读出的数据了，所以就会唤醒读管道进程；每次读管道执行完毕后，管道中就已经拥有了可以写入数据的空间了，所以就会唤醒写管道进程。

遵照这条规则，系统此时会唤醒写管道进程，执行代码如下：

```
// 代码路径：fs/pipe.c：
int read_pipe(struct m_inode * inode, char * buf, int count)
{
     ...
     while (count>0){
     ...
}
     wake_up(&inode->i_wait);
}
```

读管道进程从管道中读出数据的过程如图 7-12 所示：

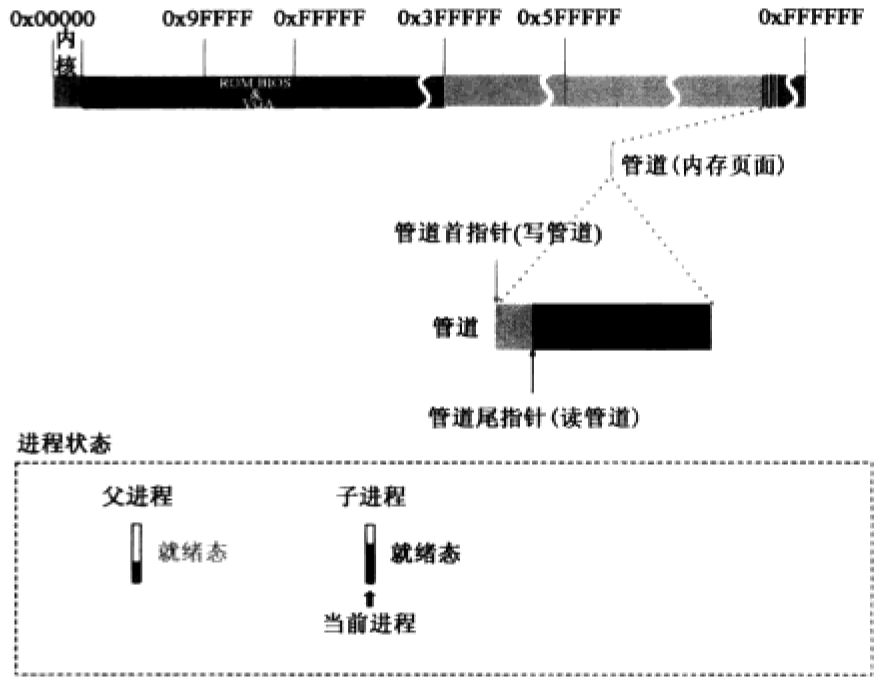


图 7-12 读管道进程从管道中读出数据

7.1.12 读管道进程继续执行，不断从管道中读出数据

当前进程还是读管道进程，读一次管道之后将返回用户空间，通过实例1中的"for(j=0;j<20000;j++)"这行代码我们得知，读管道要操作 20000 次，而且读管道进程的时间片还没有用完，所以还要继续执行读管道操作。读管道进程不断从管道中读出数据的过程如图 7-13 所示：

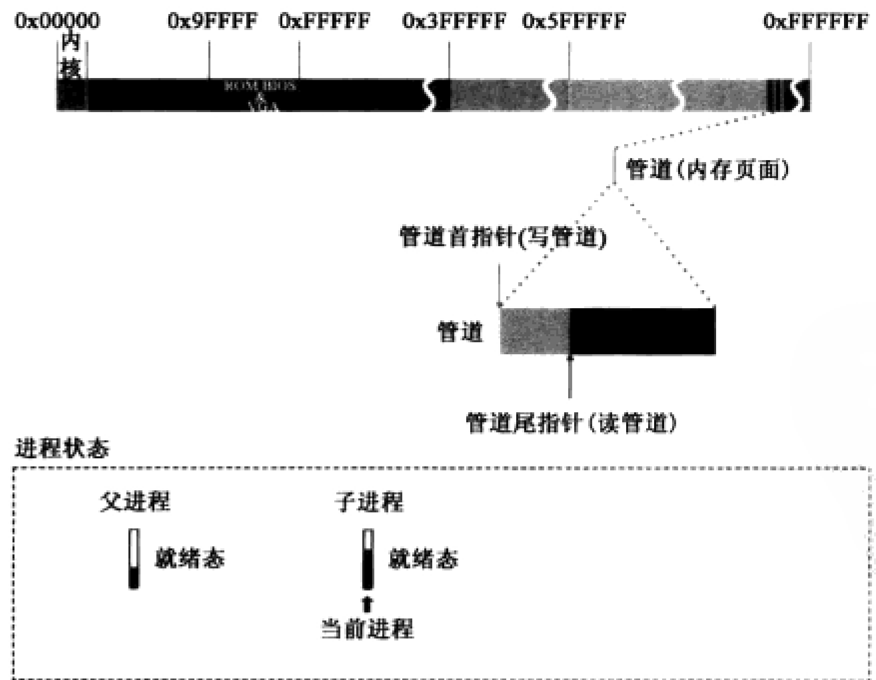


图 7-13 读管道进程不断从管道读出数据

7.1.13 读管道进程执行中发生时钟中断

在读管道进程工作的过程中，也一定会发生时钟中断，而时钟中断也仅仅是削减了它的时间

片，只要不削减为 0，它就继续执行，执行代码如下：

```
// 代码路径：kernel/schde.c：
void do_timer(long cpl)
{
    ...
    if ((--current->counter)>0) return;
    ...
}
```

读管道进程执行过程中发生时钟中断的处理方法如图 7-14 所示，注意图中读管道进程的进程条，它的时间片已经削减。

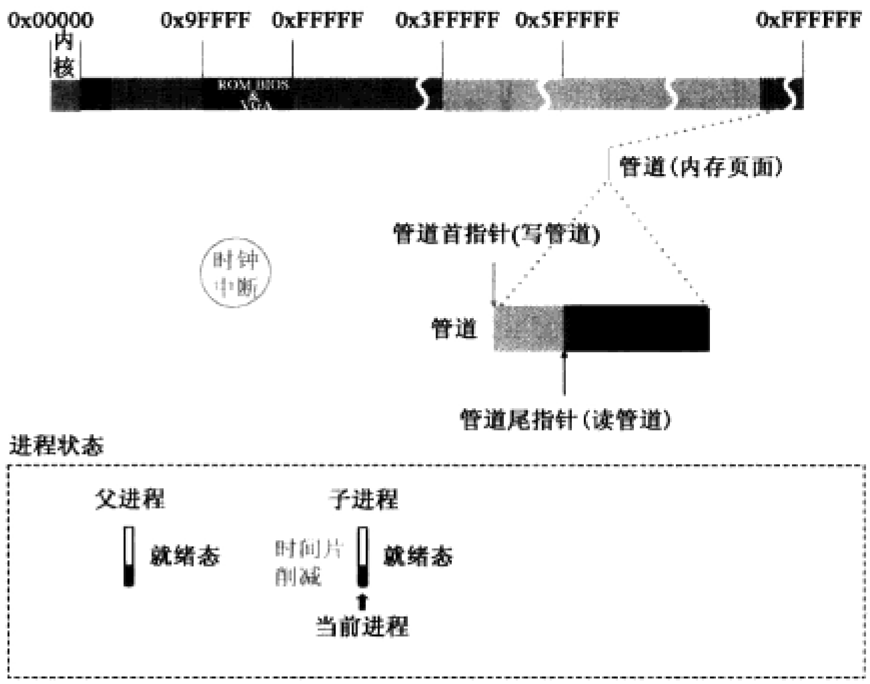


图 7-14 读管道进程执行过程中发生时钟中断

7.1.14 读管道进程执行过程中再次发生时钟中断

读管道进程执行过程中又一次发生时钟中断后，读管道进程时间片为 0 了，它被挂起并切换到写管道进程去执行，执行代码如下：

```
// 代码路径：kernel/schde.c：
void do_timer(long cpl)
{
    ...
    if ((--current->counter)>0) return;
    ...
    schedule();
}
```

对于时钟中断的处理如图 7-15 所示，图中代表读管道进程的进程条时间片已经削减为 0。

通过以上介绍我们得知，两次时钟中断的产生并不会影响到数据的写入或读出，根本原因就是数据的写入和读出都是在内核代码中进行的。

时钟中断如果在内核态产生，只会削减时间片，与数据的操作毫无关系；如果在用户态产

生，虽然会导致进程切换，但数据都是在内核态操作的，用户态进程切换并不会对数据的操作产生影响。

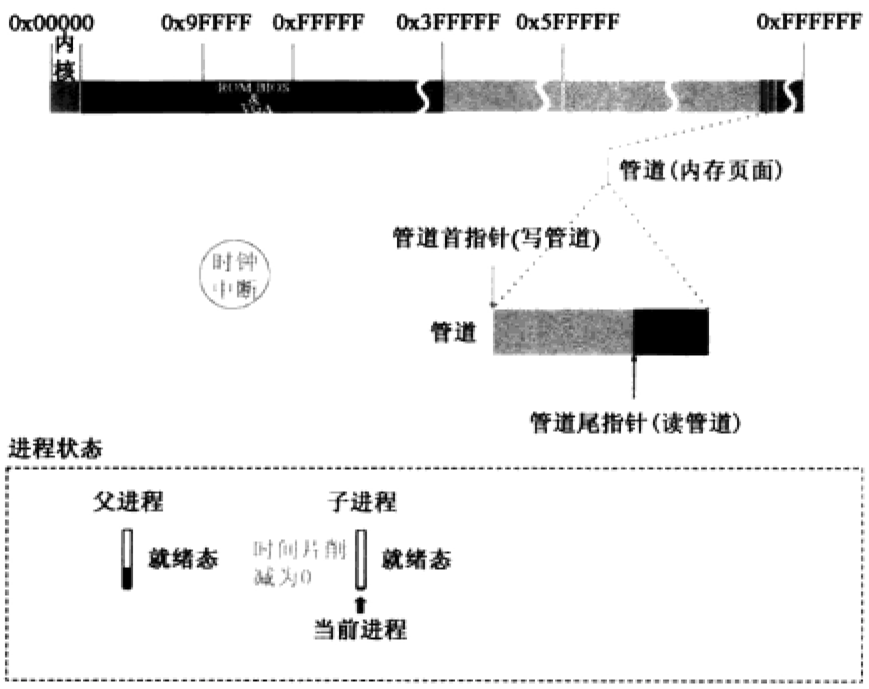


图 7-15　读进程执行过程中发生时钟中断

7.1.15　读管道进程切换到写管道进程执行

通过 7.1.10 节中的介绍我们得知，写管道进程的管道操作指针已经被移动至管道首端了，接下来，写管道进程将从管道首端开始，继续往管道中写入数据，直到再次没有剩余空间了为止。该过程如图 7-16 所示：

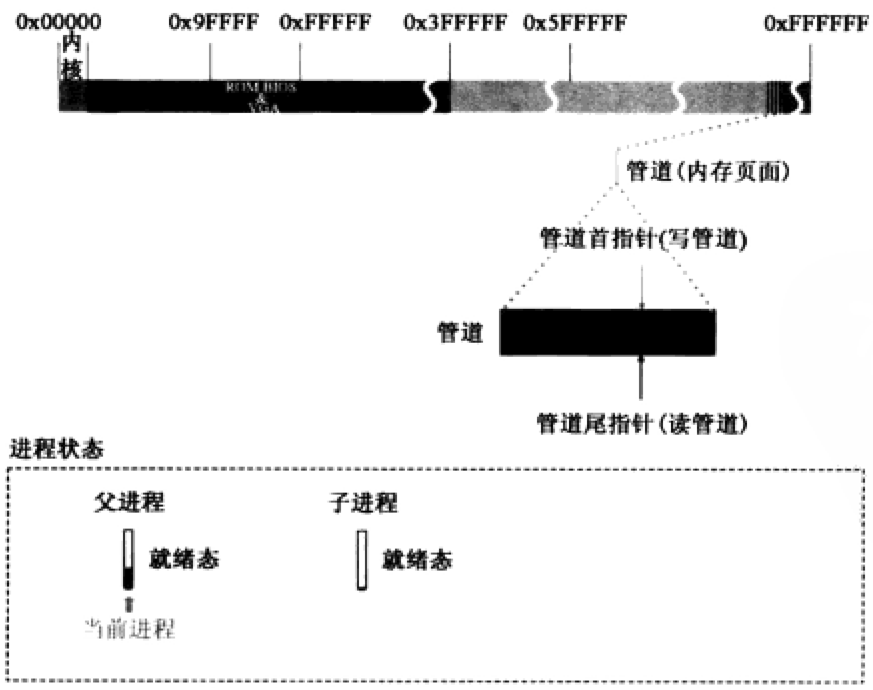


图 7-16　写管道进程继续写入数据

7.1.16　写管道进程挂起切换到读管道进程执行

管道中再次被写满后，写管道进程又要被系统挂起了，之后就切换到读管道进程去执行。

Linux 0.11 重新分配时间片的原则是当所有处于就绪态的进程时间片均为 0 时，才分配时间片，由于此时读管道进程是唯一处于就绪态的进程，并且它的时间片也用完了（我们在 7.1.14 节中已经说明），符合 Linux 0.11 操作系统重新分配时间片的原则，所以此时系统会重新给它们分配时间片，执行代码如下：

```
// 代码路径：kernel/schde.c：
void schedule(void)
{
    ...
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                    (*p)->priority;
    ...
}
```

然后，读管道进程继续执行。

读管道进程继续读取一部分数据的过程如图 7-17 所示：

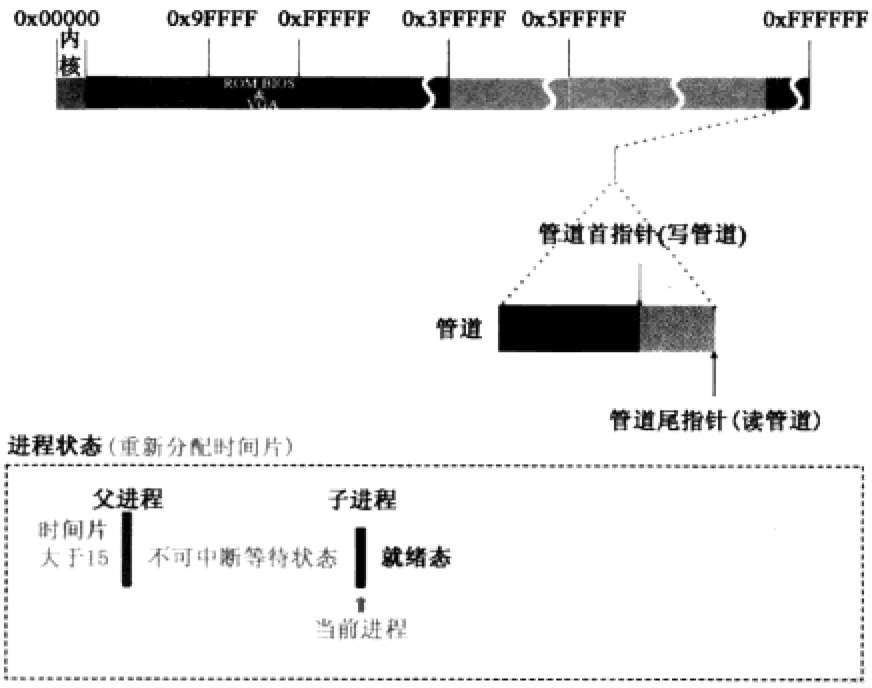


图 7-17　读管道进程继续从管道中读取数据

7.1.17　读管道进程继续执行，直到把管道中的数据读完

读管道进程开始执行后，将继续把管道中的数据读出，当操作到管道尾端后，也会将读管道指针从管道尾端移动至管道首端，并从首端继续读取管道中的内容，直至将数据彻底读完，执行代码如下：

```
// 代码路径：fs/pipe.c：
int read_pipe(struct m_inode * inode, char * buf, int count)
{
    ...
    while (count>0) {
        while (!(size=PIPE_SIZE(*inode))) {
```

```
                wake_up(&inode->i_wait);
                if (inode->i_count != 2) /* are there any writers? */
                    return read;
                sleep_on(&inode->i_wait);
            }
            chars = PAGE_SIZE-PIPE_TAIL(*inode);
            if (chars > count)
                chars = count;
            if (chars > size)
                chars = size;
            count -= chars;
            read += chars;
            size = PIPE_TAIL(*inode);
            PIPE_TAIL(*inode) += chars;
            PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
            while (chars-->0)
                put_fs_byte(((char *)inode->i_size)[size++],buf++);
    }
    ...
}
```

该过程如图 7-18 所示：

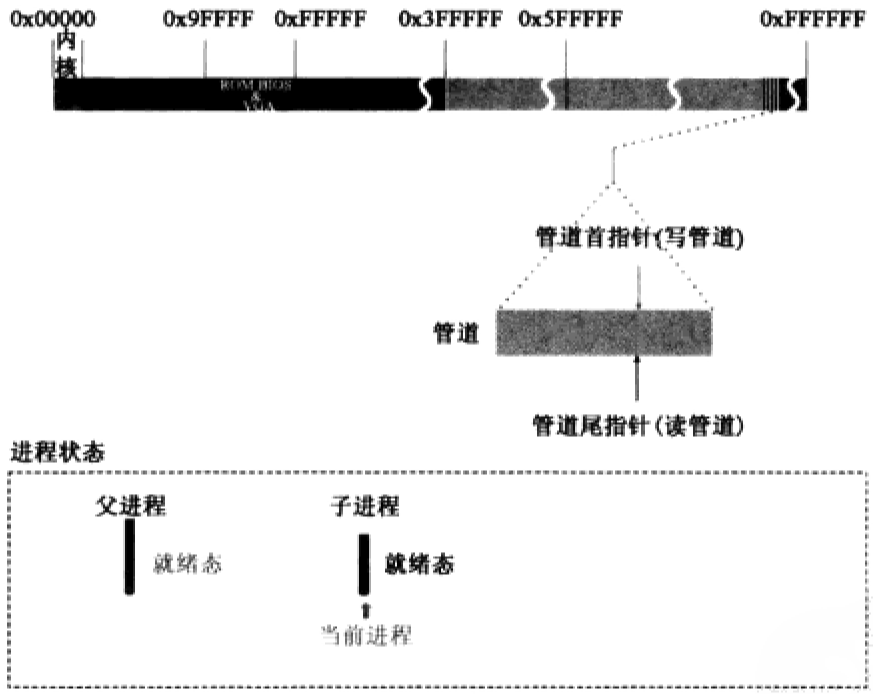


图 7-18　读管道进程将管道中的数据读完

7.1.18　读取完成后，读进程挂起，写进程继续执行

管道中的数据被读完后，读管道进程将再次被挂起，并切换到写管道进程去执行。

这就是 Linux 0.11 中管道操作的全部过程，通过对这个过程的介绍我们得知，系统是遵循如下原则对管道操作的进程进行组织管理和协调的：

- 对于读管道操作，如果管道中有数据，它就可以读出数据，如果没有数据，系统就会将它挂起。
- 对于写管道操作，如果管道中有空间，它就可以写入数据，如果没有空间，系统就会将它挂起。

这样，读管道进程和写管道进程虽然都在同一个内存页面中进行操作，但系统把它们数据操作错误的可能性都排除了，再加上 7.1.14 节中的介绍得知，时钟中断的发生也不会对数据操作造成影响，所以数据永远不会出现混乱。

7.2 信号机制

信号机制是 Linux 0.11 为进程提供的一套"局部的类中断机制"，即在进程执行的过程中，如果系统发现某个进程接收到了信号，就暂时打断进程的执行，转而去执行该进程的信号处理程序，处理完毕后，再从进程"被打断"之处继续执行。

本节将分为两个部分对信号机制进行详细的介绍。

第一部分：通过实例 2 的执行过程，对系统以及进程自身处理信号的过程进行详细的介绍。

第二部分：系统通过对信号的分析来改变进程的执行状态。

这里先开始介绍第一部分：实例 2。

这是一个关于"信号的发送、接收以及处理"的实例，我们将以此来介绍对系统以及进程自身处理信号的过程。

不妨假设有两个用户进程，一个用来接收及处理信号，这个进程的名字叫做 processig，它所对应的程序源代码如下：

```
 #include <stdio.h>
 #include <signal.h>

 void sig_usr(int signo)
 {
 if (signo == SIGUSR1)
         printf("received SIGUSR1\n");
     else
        printf("received %d\n", signo);
        signal(SIGUSR1, sig_usr);
}

int main(int argc, char **argv)
{
    signal(SIGUSR1, sig_usr);
    for (;;)
       pause();
     return 0;
}
```

一个用来发送信号，这个进程的名字叫做 sendsig，它所对应的源代码如下：

```
#include <stdio.h>
 int main(int argc, char **argv)
 {
     int pid, ret, signo;
     int i;
```

```
        if (argc != 3)
        {
            printf("Usage: sensig <signo> <pid>\n");
            return -1;
        }
        signo = atoi(argv[1]);
        pid = atoi(argv[2]);

        ret = kill(pid, signo);
        for (i=0;i<1000000;i++)

        if (ret != 0)
            printf("send signal error\n");

        return 0;
}
```

系统需要具备以下三个功能，以支持信号机制：

❑ 系统要支持进程对信号的发送和接收

系统在每个进程的管理结构 task_struct 中都设置了用以接收信号的数据成员 signal（信号位图），每个进程接收到的信号都“按位”存储在这个数据结构中。系统支持两种方式给进程发送信号：一种方式是其中一个进程通过调用特定的库函数给另一个进程发送信号；另一种方式是用户通过键盘输入信息产生键盘中断后，中断服务程序给进程发送信号。这两种方式的信号发送原理都是相同的，都是通过设置信号位图 signal 上的信号位来实现的。

本实例将结合第一种方式，即一个进程给另一个进程发送信号来展现系统对信号的发送和接收过程。

❑ 系统要能够及时地检测到进程接收到的信号

系统通过两种方式来检测进程是否接收到信号：一种方式是在系统调用返回之前检测当前进程是否接收到信号；另一种方式是时钟中断产生后，其中断服务程序执行结束之前，检测当前进程是否接收到信号。

这两种信号检测方式的本质都是相同的。本实例将结合第一种方式来展现系统对接收到的信号的检测。

❑ 系统要支持进程对信号进行处理

系统要能够保证，当用户进程不需要处理信号时，信号处理函数完全不参与用户进程的执行；当用户进程需要处理信号时，进程的程序将暂时停止执行，转而去执行信号处理函数，待信号处理函数执行完毕后，进程程序将从“暂停的现场处”继续执行。

本实例将从“用户自定义的信号处理函数与进程进行绑定”、“系统对信号的预处理”和“信号处理完毕后进程现场的恢复”这三方面入手来展现系统是如何做到这些的。

下面我们就来介绍这两个进程都是如何开始运行的，现在用户处于 shell 界面下。

第一步：运行 processig 程序，输入指令如下：

```
[/usr/root]# ./processsig &
```

```
<160>
[/usr/root]#
```

这样 processig 进程就开始运行了，这里也就得知 processig 这个进程的进程号是 160。

第二步：运行 sendsig 程序发送信号 SIGUSR1 给进程 signal：

```
[/usr/root]# ./sendsig 10 160
received SIGUSR1
[/usr/root]#
```

现在，我们开始分别介绍这两个进程的执行情况，由于 processig 进程是先开始执行的，所以我们先来介绍它的执行情况。

7.2.1 processig 进程开始执行

processig 进程开始执行，进入 main 函数后，先要将用户自定义的信号处理函数与 processig 进程相绑定，这样才能使信号处理函数在“需要”执行的时候正常执行。

用户程序是通过调用 signal 函数来实现这个绑定的。这个函数是库函数，它执行后会产生软中断 int0x80，并映射到 sys_signal 这个系统调用函数去执行，这个函数的功能是将用户自定义的信号处理函数 sig_usr 与 processig 进程相绑定。这意味着，只要 processig 进程接收到 SIGUSR1 信号，就调用 sig_usr 函数来处理该信号，绑定工作就是通过该函数来完成的。

进入 sys_signal 函数后，系统要在绑定之前先检测用户指定的信号是否合理，由于 Linux 0.11 中只能默认处理 32 种信号，而且默认忽略 SIGKILL 这个信号，所以只要用户给出的信号不符合这些要求，系统将不能处理。通过 signal(SIGUSR1, sig_usr); 这行用户程序的代码我们得知，用户传入的参数是 SIGUSR1，因此检测通过，执行代码如下：

```
// 代码路径：kernel/signal.c：
int sys_signal(int signum, long handler, long restorer)
{
    ...
    if (signum<1 || signum>32 || signum==SIGKILL)
    return -1;
    ...
}
```

检测完毕后，开始对 processig 进程管理结构中的 sigaction[32] 结构进行设置，执行代码如下：

```
// 代码路径：kernel/signal.c：
int sys_signal(int signum, long handler, long restorer)
{
    ...
    tmp.sa_handler = (void (*)(int)) handler;
    tmp.sa_mask = 0;
    tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
    tmp.sa_restorer = (void (*)(void)) restorer;
    handler = (long) current->sigaction[signum-1].sa_handler;
    current->sigaction[signum-1] = tmp;
```

```
    return handler;
    ...
}
```

其中，"tmp.sa_handler = (void (*)(int)) handler"这行代码实施了绑定工作。

在当前情况下，handler 参数就是 processig 进程程序中"signal(SIGUSR1, sig_usr)"这行代码的 sig_usr 函数的地址，该设置意味着，只要将来进程接到信号，就由 sig_usr 这个函数来处理。

"current->sigaction[signum-1] = tmp"这行代码确定了将由 sigaction[signum-1] 这一项为 SIGUSR1 信号提供服务。

每个进程管理结构中都有 sigaction[32] 这个管理结构，该管理结构有 32 个成员，正好对应默认的 32 种信号，sigaction[32] 中每一个成员都会为每一种信号的处理提供一套服务。

此时的设置如图 7-19 所示：

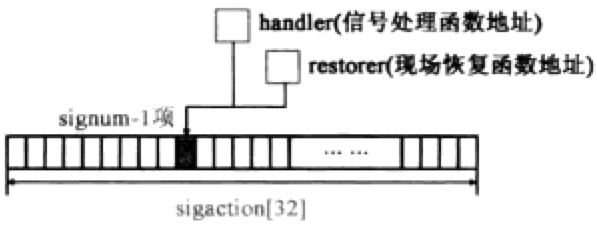


图 7-19 设置用户进程信号处理地址

系统就是通过这样的方式将用户自定义的信号处理函数与进程进行了绑定。

值得注意的是，图中的 restorer 函数也在 sys_signal 函数中绑定了，这个函数的功能也很重要，我们将在 7.2.6 节中详细介绍它。

processig 进程的状态及其代码在内存中的情况如图 7-20 所示，此时 processig 进程处于就绪态：

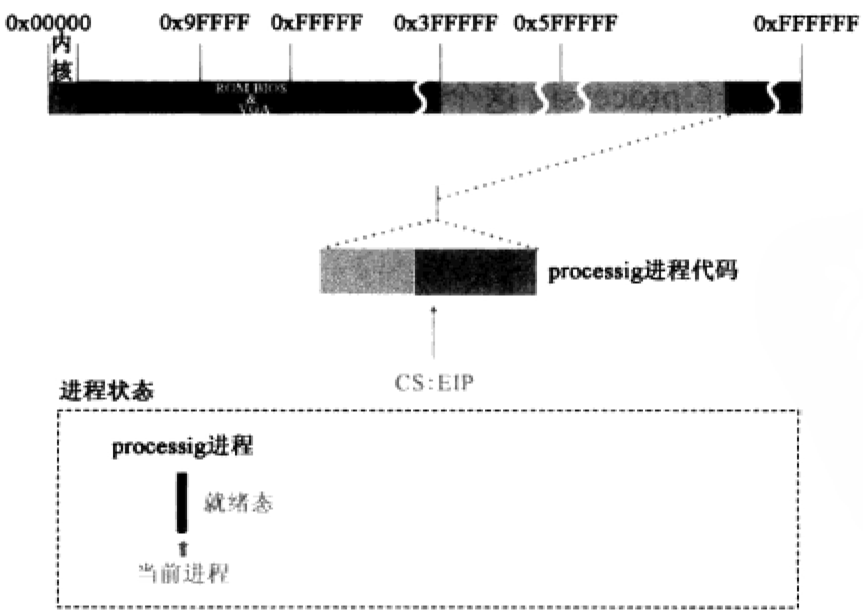


图 7-20 两个进程在内存中的情况

7.2.2 processig 进程进入可中断等待状态

在 processig 进程的程序中，为了体现信号对进程执行状态的影响，我们特意调用了 pause 函

数。这个函数最终将导致该进程被设置为“可中断等待状态”，等到该进程接收到信号后，它的状态将由“可中断等待状态”转换为“就绪状态”，这一过程我们将在 7.2.3 节中详细介绍。下面我们来介绍 pause 函数的执行过程。

执行完 signal 函数后，将返回到 processig 进程的用户空间继续执行，会执行到 for 循环中，调用 pause 函数，这个函数会映射到系统调用函数 sys_pause 中，执行代码如下：

```
// 代码路径：kernel/sched.c：
int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    return 0;
}
```

processig 进程将被设置为可中断等待状态，如图 7-21 所示：

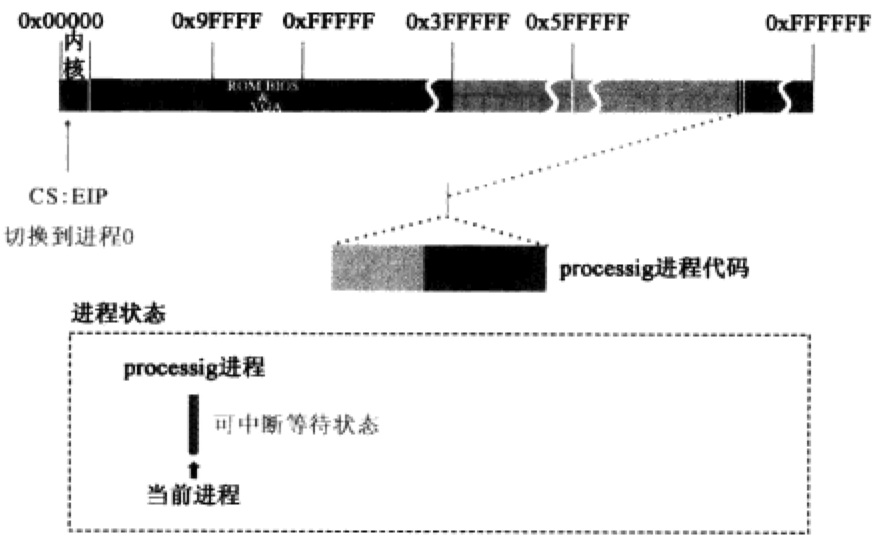


图 7-21　processig 进程进入可中断等待状态

由于在 shell 界面下只运行了 processig 这个用户进程，因此当前系统中所有进程都是非就绪状态的，所以接下来在调用 schedule 函数后会默认切换到进程 0。

对应代码如下：

```
// 代码路径：kernel/sched.c：
void schedule(void)
{
    ...
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
    }
```

```
    if (c) break;
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p)
            (*p)->counter = ((*p)->counter >> 1) +
            (*p)->priority;
    }
    switch_to(next);
}
```

通过代码可以看出，在当前情况下，next 的值一定为 0，所以会默认切换到进程 0 去执行。

7.2.3　sendsig 进程开始执行并向 processig 进程发信号

现在，接收信号的 processig 进程已经暂时被挂起了，我们开始让 sendsig 进程执行。该进程就会给 processig 进程发送信号，然后切换到 processig 进程去执行。我们来看具体的过程。

用户在 shell 界面下输入“./sendsig 10 160”这条命令，意思是让 sendsig 进程开始执行，并传递“10”和“160”这两个参数，这两个参数的意义是：“10”代表 SIGUSR1 信号，160 代表将要接收该信号的进程的进程号，此时表示即将要给 processig 进程发送 SIGUSR1 信号。

sendsig 进程开始执行后，会执行到“ret = kill(pid, signo)”这一行，其中 kill 是个库函数，最终会映射到 sys_kill 这个函数中去执行，在这个函数中，将参照“10”和“160”这两个参数给 processig 进程发送 SIGUSR1 信号，执行代码如下：

```
// 代码路径：kernel/exit.c：
    int sys_kill(int pid,int sig)
{
    ...
    else if (pid>0) while (--p > &FIRST_TASK) {
        if (*p && (*p)->pid == pid)
            if (err=send_sig(sig,*p,0))
                retval = err;
    ...
}
```

其中 send_sig 这个函数负责具体的信号发送工作，它会在 processig 进程的信号位图 signal 中找到 SIGUSR1 这一信号对应的位置，然后将其置 1，以此来发送信号，执行代码如下：

```
// 代码路径：kernel/exit.c：
static inline int send_sig(long sig,struct task_struct * p,int priv)
{
     ...
     p->signal |= (1<<(sig-1));
     ...
}
```

将 SIGUSR1 信号发送给 processig 信号的过程及该过程对 processig 进程管理结构中相应字段的影响如图 7-22 所示。

之后，就返回到 sendsig 用户进程空间内了，返回后进入 for 循环执行，在 1000000 次循环的过程中，时钟中断会不断地产生，sendsig 进程的时间片将被削减为 0，于是导致进程切换，

schedule 函数开始执行，对应代码如下：

```
// 代码路径：kernel/sched.c：
void schedule(void)
{
    ...
    if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
    ...
}
```

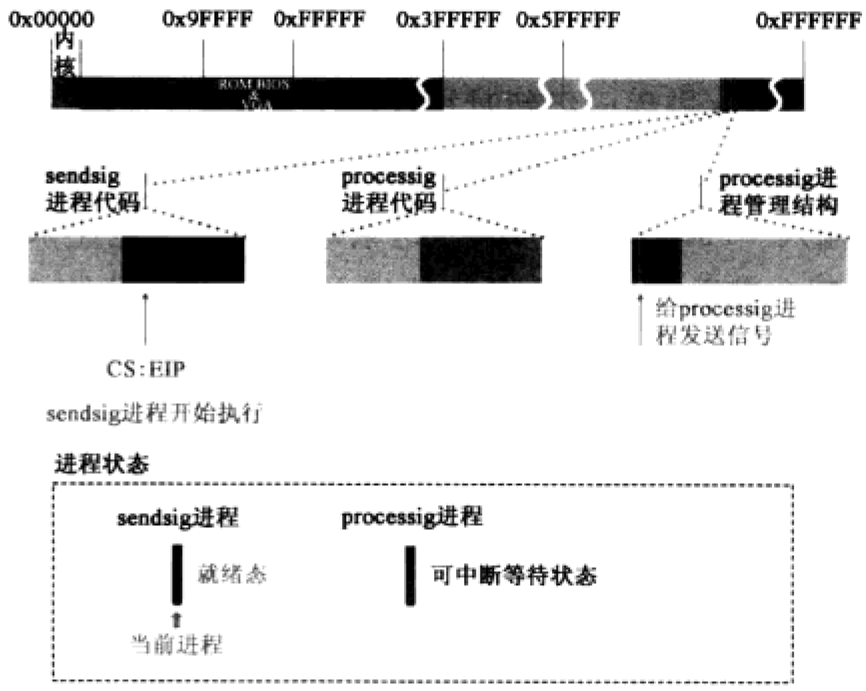


图 7-22 给 processig 进程发送信号

由于 sendsig 进程给 processig 进程发送了信号，所以，当遍历到 processig 进程时，它的信号位图 signal 上就会有所体现，同时，processig 进程还是可中断等待状态，所以，它就会被改设为就绪状态。该过程如图 7-23 所示：

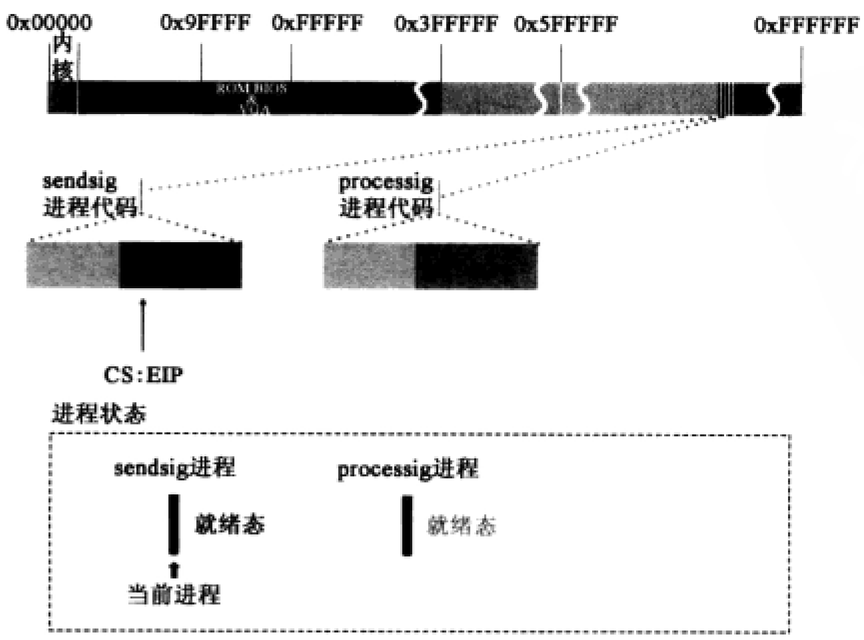


图 7-23 由于 processig 进程收到信号，因此被置为就绪态

上段代码的意思是：如果遍历到的进程在信号位图上有信号被置位，而且处于可中断等待状态，系统就把它改设为就绪状态。现在的情况是，sendsig 进程给 processig 进程发送了信号，同时，processig 进程还是可中断等待状态，两个条件全部成立，所以 processig 进程就会被改设为就绪状态。

等到第二次遍历时，就会切换到 processig 进程去执行，执行代码如下：

```
//代码路径：kernel/sched.c：
void schedule(void)
{
    ...
    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                    (*p)->priority;
    }
    switch_to(next);
}
```

7.2.4 系统检测当前进程接收到信号并准备处理

这里我们来介绍系统是如何检测到当前进程是否接收到信号的。

processig 进程开始执行后，会继续在 for 循环中执行 pause 函数，由于这个函数最终会映射到 sys_pause 这个系统调用函数中去执行，所以当系统调用返回时，就一定会执行到 ret_from_sys_call: 标号处，并最终调用 do_signal 函数，开始着手处理 processig 进程的信号，执行代码如下：

```
//代码路径：kernel/system_call.s：
    ...
    ret_from_sys_call:
   movl _current,%eax           # task[0] cannot have signals
   cmpl _task,%eax
   je 3f
   cmpw $0x0f,CS(%esp)          # was old code segment supervisor ?
   jne 3f
   cmpw $0x17,OLDSS(%esp)       # was stack segment = 0x17 ?
   jne 3f
   movl signal(%eax),%ebx
   movl blocked(%eax),%ecx
   notl %ecx
   andl %ebx,%ecx
```

```
bsfl %ecx,%ecx
je 3f
btrl %ecx,%ebx
movl %ebx,signal(%eax)
incl %ecx
pushl %ecx
call _do_signal
 ...
```

7.2.5 系统检测信号处理函数指针挂接是否正常

现在开始介绍信号处理之前的准备工作。

进入 do_signal 函数后，先要对 processig 进程的信号处理函数进行判定，我们在 7.2.1 节中介绍过，processig 进程的信号处理函数指针被加载到了进程管理结构的 sigaction[32] 结构中，如图 7-24 所示：

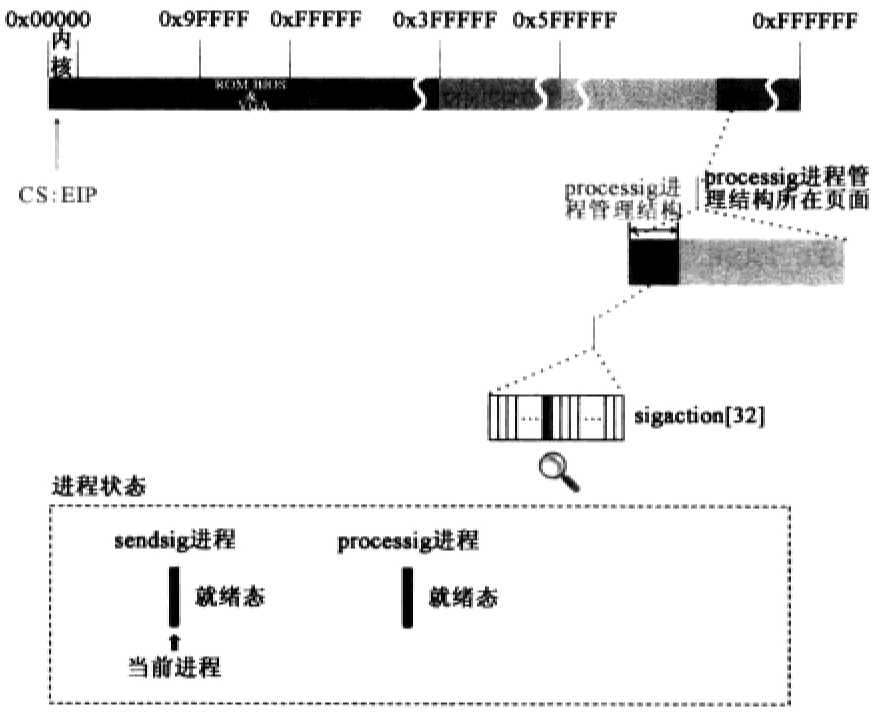


图 7-24 信号处理函数在 sigaction 结构中的位置

现在它开始发挥作用了，如果这个指针为空，该进程就很有可能退出。当然，此时的情况是，这个指针肯定不为空，它指向了 processig 进程的信号处理函数 sig_usr。检测工作对应的代码如下：

```
// 代码路径：kernel/signal.c：
      void do_signal(long signr,long eax, long ebx, long ecx, long edx,
        long fs, long es, long ds,
        long eip, long cs, long eflags,
        unsigned long * esp, long ss)
      {
        ...
        struct sigaction * sa = current->sigaction + signr - 1;
        ...
        sa_handler = (unsigned long) sa->sa_handler;
        if (sa_handler==1)
```

```
            return;
        if (!sa_handler) {
            if (signr==SIGCHLD)
                return;
        else
            do_exit(1<<(signr-1));
        }
        ...
}
```

7.2.6 调整 processig 进程的内核栈结构，使之先执行信号处理函数

这里继续介绍系统为信号处理所做的准备，以及这些准备工作在信号处理过程中产生的作用。

这里所做的准备工作的核心目的是对用户栈中的数据进行调整，使得此次系统调用返回后会“首先”执行 processig 进程的“信号处理函数”，然后再从用户进程“中断位置”继续执行。在 7.2.4 节中所介绍的 pause 函数执行后产生 int 0x80 软中断的下一条指令处就是这个用户进程的“中断位置”（当然，如果不需要处理信号，直接返回到“中断位置”处就可以了，但现在要先处理信号问题，再回“中断位置”）。

通过 2.2 节和 2.3 节的介绍我们得知，软中断产生后，CPU 将自动在当前进程“内核栈”中保存用户进程执行的“指令和数据”，其中包括 eip、cs、eflags、esp、ss 这些寄存器的值，这样，只要系统调用返回，就要依据“内核栈”中这些保存的寄存器值，来决定返回到用户空间的“哪个位置”继续执行。

面对这种情况，Linux 0.11 就在此次系统调用返回前，先把这些“内核栈”中保存的寄存器值备份在当前进程的“用户栈”中（内核有能力访问到所有物理内存空间，做这件事不成问题），然后对“内核栈”中的这些原有的寄存器值进行更改，这就使得系统调用函数返回后，首先根据“内核栈”中最新更改的数据跳转到用户空间的信号处理函数处执行，进入用户空间后，“用户栈”就该发挥作用了，等到信号处理完毕后，再通过前面备份在用户空间的“指令和数据”，返回到“中断位置”处执行。

这就是信号处理的全部策略，下面我们来看具体的实现代码：

```
// 代码路径：kernel/signal.c：
void do_signal(long signr,long eax, long ebx, long ecx, long edx,
        long fs, long es, long ds,
        long eip, long cs, long eflags,
        unsigned long * esp, long ss)
      {
        ...
        if (sa->sa_flags & SA_ONESHOT)
         sa->sa_handler = NULL;
         *(&eip) = sa_handler;
         longs = (sa->sa_flags & SA_NOMASK)?7:8;
         *(&esp) -= longs;
         verify_area(esp,longs*4);
         tmp_esp=esp;
         put_fs_long((long) sa->sa_restorer,tmp_esp++);
         put_fs_long(signr,tmp_esp++);
```

```
        if (!(sa->sa_flags & SA_NOMASK))
                put_fs_long(current->blocked,tmp_esp++);
        put_fs_long(eax,tmp_esp++);
        put_fs_long(ecx,tmp_esp++);
        put_fs_long(edx,tmp_esp++);
        put_fs_long(eflags,tmp_esp++);
        put_fs_long(old_eip,tmp_esp++);
        current->blocked |= sa->sa_mask;
}
```

上述修改内核栈和用户栈的过程，以及用户栈空间和内核栈空间的数据调整前后的变化如图 7-25 和图 7-26 所示：

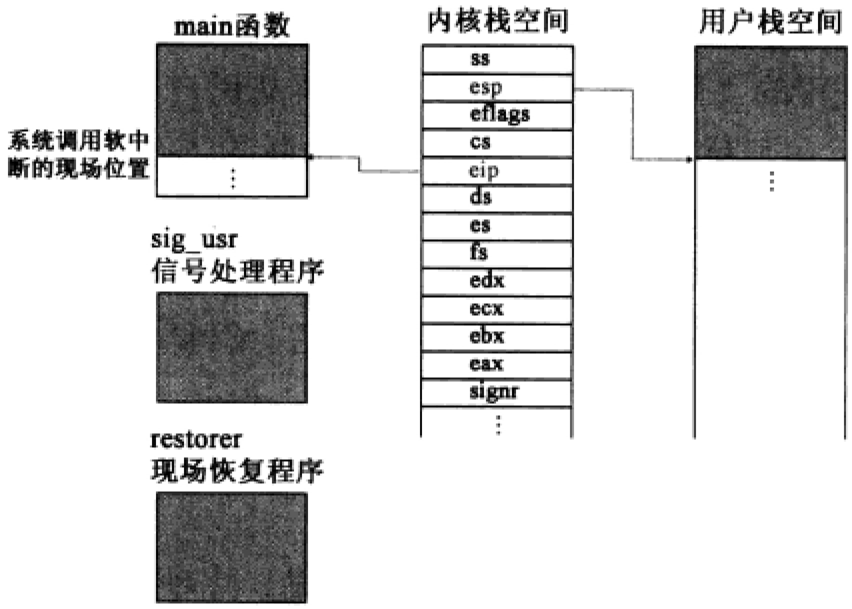


图 7-25　调整之前内核栈的现场数据及其意义

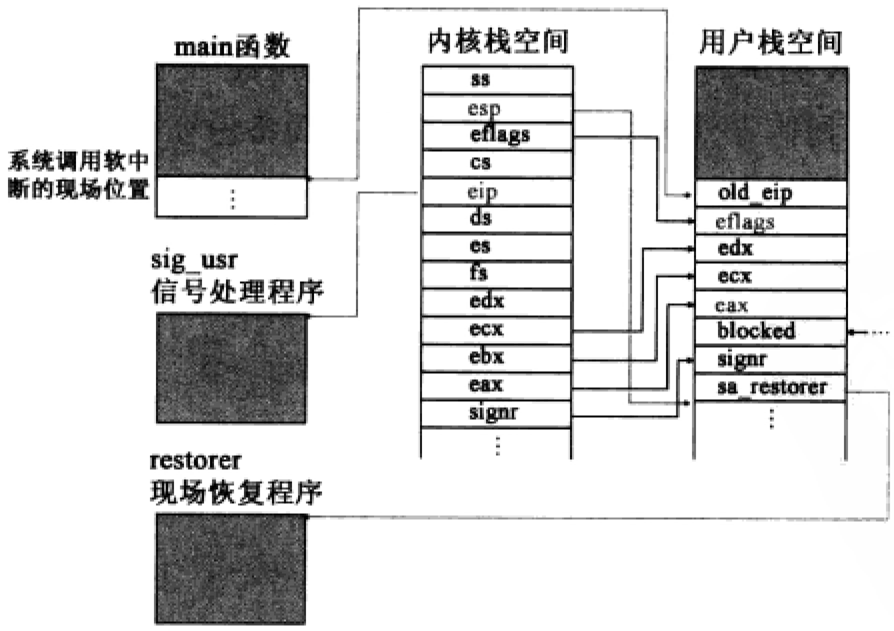


图 7-26　调整后的内核栈的现场数据及其意义

其中“*(&eip) = sa_handler”这一行是在系统调用返回后对 eip 位置的调整，使其指向 processig 进程的信号处理函数 sig_usr。

“*(&esp) −= longs”这一行是对“用户栈”空间的栈顶指针 esp 进行调整，使栈顶指针向栈

底的反方向移动，以便接下来在用户栈空间中备份数据。

```
put_fs_long((long) sa->sa_restorer,tmp_esp++);
put_fs_long(signr,tmp_esp++);
if (!(sa->sa_flags & SA_NOMASK))
    put_fs_long(current->blocked,tmp_esp++);
put_fs_long(eax,tmp_esp++);
put_fs_long(ecx,tmp_esp++);
put_fs_long(edx,tmp_esp++);
put_fs_long(eflags,tmp_esp++);
put_fs_long(old_eip,tmp_esp++);
```

信号预处理工作到这里已经完成，这些代码全部都是向用户栈空间中备份的数据，接下来我们来看这些数据在此次系统调用返回后都是如何被应用的，它们对“信号处理函数的执行”以及对“执行后进程现场的恢复”都产生了哪些影响。

由于在 7.2.1 节中已经将信号处理函数 sig_usr 与 processig 进程进行了绑定，因此系统调用返回后，就会到 processig 进程的 sig_usr 函数处执行。sig_usr 这个函数在最后会执行 signal(SIGUSR1, sig_usr); 这一行代码，重新设置该进程的信号处理函数指针，以便下次使用。

signal 函数执行完毕后，一定会执行“ret”这一行代码，于是此时处于栈顶的 sa->sa_restorer 所代表的函数地址值就发挥作用了，ret 的本质就是用栈顶指针的地址值来设置 EIP，然后跳转到 EIP 指向的地址位置去执行。此时就应该跳转到 sa->sa_restorer 所代表的函数地址值位置去执行了。

我们在 7.2.1 节的最后讲到，还将一个叫做 restorer 的函数地址绑定在了 sigaction 结构中，即“tmp.sa_restorer = (void (*)(void)) restorer”这行代码。restorer 是一个库函数的地址，它是由 signal 这个库函数传递下来的实参，这个库函数将来会在信号处理工作结束后恢复用户进程执行的“指令和数据”，并最终跳转到用户程序的“中断位置”处执行。

现在信号已经处理完了，restorer 函数开始工作，我们先来看一下这个函数的代码：

```
.globl ____sig_restore
.globl ____masksig_restore
____sig_restore:
addl $4,%esp
popl %eax
popl %ecx
popl %edx
popfl
ret

____masksig_restore:
addl $4,%esp
call ____ssetmask
addl $4,%esp
popl %eax
popl %ecx
popl %edx
popfl
ret
```

通过这段源代码，可以非常明显地看出来，这个库函数就是用来协助恢复用户进程执行的“指令和数据”。以下这几行代码：

```
popl %eax
popl %ecx
popl %edx
popfl
```

中使用的数据，正是

```
put_fs_long(eax,tmp_esp++);
put_fs_long(ecx,tmp_esp++);
put_fs_long(edx,tmp_esp++);
put_fs_long(eflags,tmp_esp++);
```

这几行代码在用户栈中备份的数据。

现场恢复后，就要返回到现场执行。

注意看最后一行汇编“ret”，由于 ret 的本质就是用当前栈顶的值设置 EIP，并使程序跳转到 EIP 指向的位置去执行，很显然，经过一系列清栈操作后，当前栈顶的数值就是“put_fs_long(old_eip,tmp_esp++)”这行代码设置的，这个 old_eip 就是 pause 函数为了映射到 sys_pause 函数，产生软中断 int0x80 的下一行代码，即 processig 进程的“中断位置”。所以，ret 执行后，信号就处理完毕了，并最终回到 pause 中去继续执行。

这就是 Linux 0.11 中信号处理的全部过程。

7.2.7 信号对进程执行状态的影响

现在，我们已经对信号机制有了一定的了解，接下来将介绍本节的第二部分内容，即分别以进程的“可中断等待状态”和“不可中断等待状态”为例，进一步分析“信号”对进程执行状态的影响。

可中断等待状态案例：

```
#include <stdio.h>
void main ()
{
    exit();
}
```

shell 进程创建了这个用户进程（该进程就自然成为了 shell 进程的子进程）后，又被设置为可中断等待状态，现在，这个用户进程就要退出了，我们以此为例来介绍信号对进程执行状态的影响。

7.2.7.1 用户进程退出并向 shell 进程发送信号

用户进程先调用 exit 函数来处理自己退出前的一些事务，包括将自己的程序所占用的内存页面释放、解除该进程与所操作文件的关系等，然后，用户进程给 shell 进程发送“子进程退出”信号，通知 shell 进程自己即将退出了，最后将自己设置为僵死状态并调用 schedule 函数，准备

进程切换，对应代码如下：

```
// 代码路径：kernel/exit.c：
int do_exit(long code)
{
      ...
      tell_father(current->father);
      schedule();
      ...
}
static void tell_father(int pid)
{
    ...
    task[i]->signal |= (1<<(SIGCHLD-1));
    ...
}
```

用户进程退出过程中发送信号以及将自身设置为僵死状态的过程如图 7-27 所示：

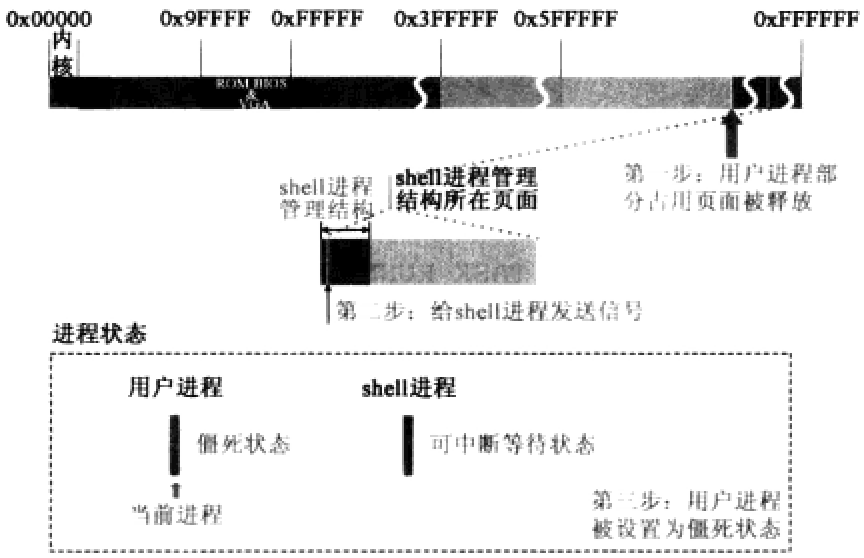


图 7-27　用户进程向 shell 进程发信号

7.2.7.2　shell 进程被唤醒并调度执行

进入 schedule 函数后，先对所有进程进行第一次遍历，如果发现哪个进程接收到了指定的信号，而且该进程还是可中断等待状态，那么就将该进程设置为就绪状态。系统通过遍历得知，shell 进程符合此条件，于是 shell 进程就被设置为就绪状态。如图 7-28 所示：

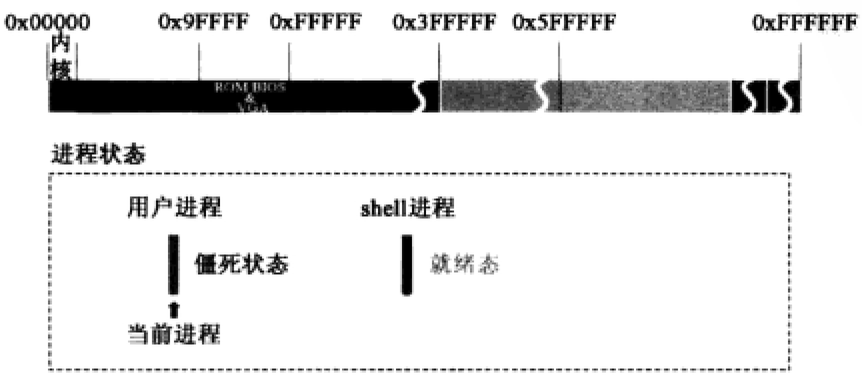


图 7-28　将 shell 进程设置为就绪状态

对应代码如下：

```
// 代码路径：kernel/sched.c：
void schedule(void)
{
    ...
    if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
    ...
}
```

之后，第二次遍历所有进程，假设当前只有 shell 进程是就绪状态，于是切换到 shell 进程去执行，执行代码如下：

```
// 代码路径：kernel/sched.c：
void schedule(void)
{
    ...
    while (1) {
            c = -1;
            next = 0;
            i = NR_TASKS;
            p = &task[NR_TASKS];
            while (--i) {
                  if (!*--p)
                      continue;
                  if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                      c = (*p)->counter, next = i;
            }
            if (c) break;
            for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                  if (*p)
                      (*p)->counter = ((*p)->counter >> 1) +
                          (*p)->priority;
     }
     switch_to(next);
}
```

7.2.7.3 shell 进程进行善后处理

shell 进程开始执行后，调用 wait 函数，一经发现接收到的信号确实是子进程的退出信号，则对此用户进程的退出进行善后处理，包括将用户进程管理结构所占用的页面释放掉，等等，如图 7-29 所示：

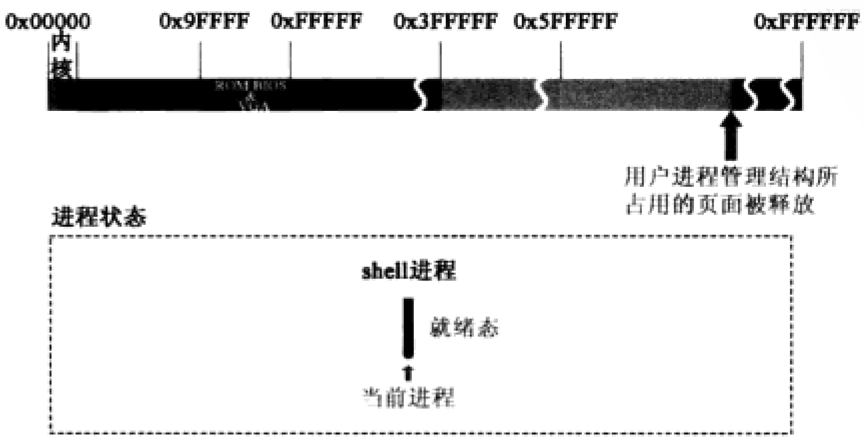


图 7-29 shell 进程处理用户进程退出后的善后工作

执行代码如下：

```
// 代码路径：kernel/exit.c：
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    ...
    if (!(current->signal &= ~(1<<(SIGCHLD-1))))
    ...
}
```

7.2.7.4　shell 进程再次被挂起

之后 shell 进程继续执行，从 tty0 这个终端设备文件上读取数据，我们假设此时用户并没有通过键盘输入任何信息，这样 shell 进程什么数据都没有读到，于是 shell 进程将被设置为可中断等待状态，等待着下次被唤醒。如图 7-30 所示：

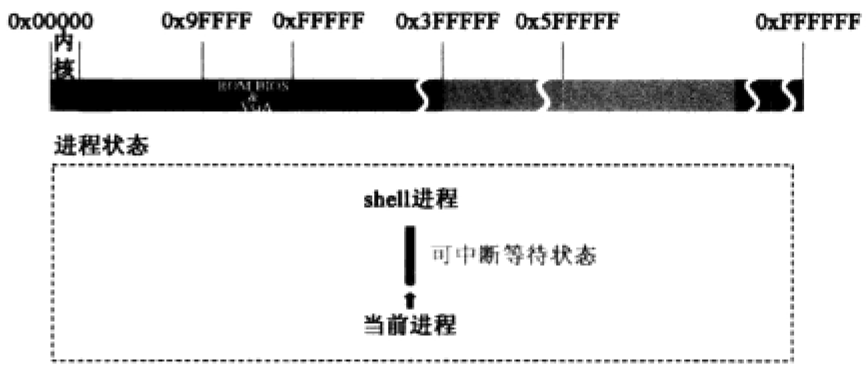


图 7-30　shell 进程再次进入可中断等待状态

由此可见：对处于可中断等待状态的进程而言，可以通过给它发信号，使系统在调用 schedule 函数进行进程切换时发现该信号，然后将其改设为就绪状态，之后它将根据发送的信号做出相应处理。

接下来，我们来讲解进程的不可中断等待状态。我们假设现在系统中有三个用户进程，分别是进程 A、进程 B 和进程 C，它们现在都处于就绪状态，其中进程 B 是进程 A 的子进程，进程 A 正在运行，我们以此为例来介绍信号对进程执行状态的影响。

进程 A 和进程 B 案例程序：

```
main()
{
    char buffer[12000];
int fd = open("/mnt/user/hello.txt", O_RDWR ,0644));
// 读文件
read(fd,buffer,sizeof(buffer));
  if (!fork())
      exit();// 进程 B（子进程）的代码
  return;
}
```

进程 C 案例程序：

```
main()
{
```

```
    int  i,j;
    for(i=0;i<1000000;i++)
        for(i=0;i<1000000;i++)
}
```

7.2.7.5 进程 A 由于等待读盘而被挂起

进程 A 需要读硬盘，于是调用 read 函数产生软中断，并最终映射到 sys_read 函数去执行。经过一系列函数调用后，发送读盘命令，返回后，进程 A 被设置为不可中断等待状态。这是因为进程 A 下一步的执行需要此次读盘的数据来支持，在数据被读出之前，这个进程无论收到什么信号，都不能被唤醒。这是由进程 A 的执行需求决定的，如果被唤醒了，它将操作缓冲区里面的数据，而此时缓冲区中的数据并没有从硬盘读出，这将引起数据混乱，如图 7-31 所示：

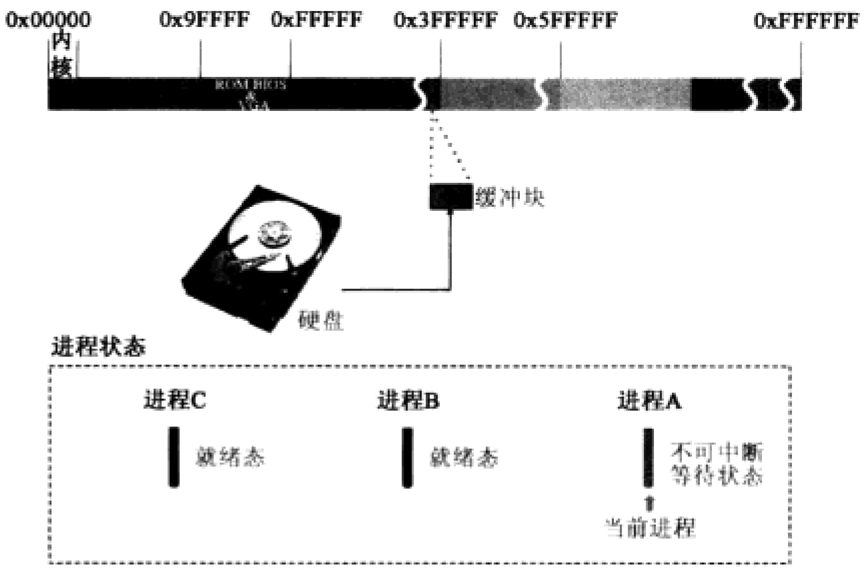


图 7-31 进程 A 挂起

进程 A 执行并最终挂起，对应的代码如下所示：

```
// 代码路径：fs/buffer.c：
struct buffer_head * bread(int dev,int block)
{
    ...
    wait_on_buffer(bh);
    ...
}
static inline void wait_on_buffer(struct buffer_head * bh)/* 不是接口 */
{
    ...
    sleep_on(&bh->b_wait);
    ...
}
// 代码路径：kernel/sched.c：
void sleep_on(struct task_struct **p)
{
    ...
    current->state = TASK_UNINTERRUPTIBLE;
    ...
}
```

7.2.7.6　进程 A 切换到进程 B 执行

之后也要调用 schedule 函数，最终会切换到其他进程执行，我们假设切换到进程 A 的子进程——进程 B 去执行，进程 B 执行后，准备退出，于是进程 B 被设置为僵死状态，之后给进程 A 发信号，通知进程 A 自己将要退出了，最后调用 schedule 函数，准备进程切换，如图 7-32 所示：

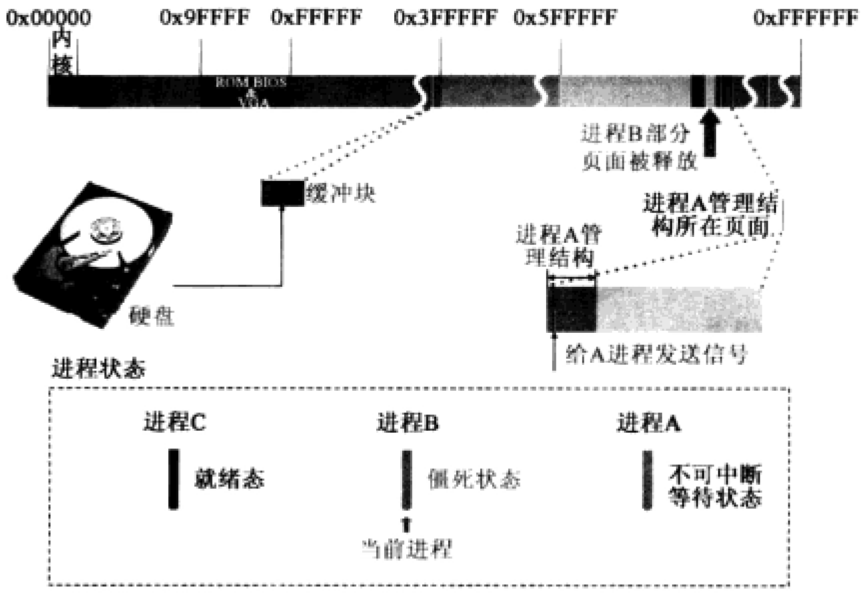


图 7-32　进程 B 退出并给进程 A 发信号

执行代码如下：

```
// 代码路径：kernel/exit.c：
int do_exit(long code)
{
      ...
      current->state = TASK_ZOMBIE;
      ...
      tell_father(current->father);
      schedule();
      ...
}
static void tell_father(int pid)
{
     ...
     task[i]->signal |= (1<<(SIGCHLD-1));
     ...
}
```

7.2.7.7　进程 A 虽收到信号但无法唤醒

进入 schedule 函数后，第一次遍历所有进程，此时进程 A 虽然接收到了信号，但由于它是不可中断等待状态，所以它并不会被改设为就绪态，于是此次只能切换到进程 C 去执行，如图 7-33 所示。

执行代码如下：

```
// 代码路径：kernel/exit.c：
void schedule(void)
{
```

```
    ...
    if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                  (*p)->state==TASK_INTERRUPTIBLE)
                         (*p)->state=TASK_RUNNING;
    ...
}
```

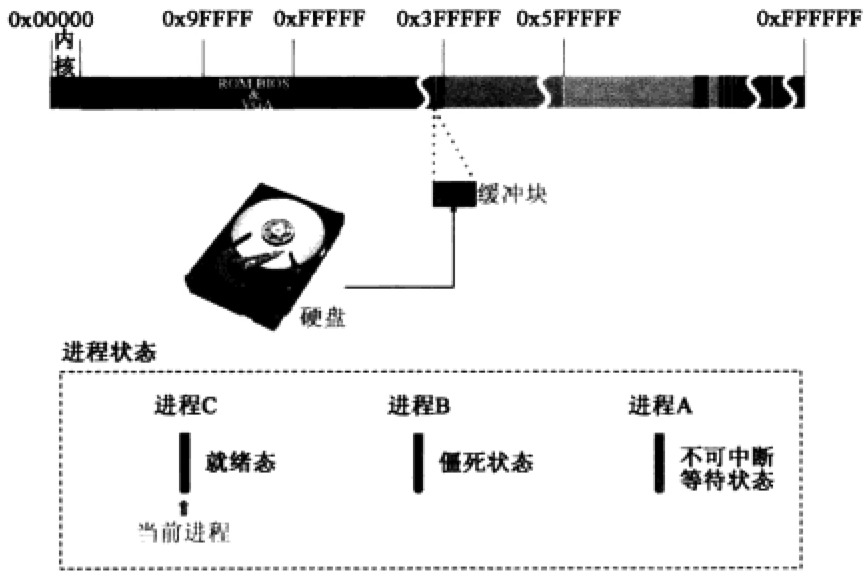


图 7-33 对于不可中断等待状态无法用信号唤醒

7.2.7.8 由于外设数据读取完成，进程 A 被唤醒

进程 C 执行了一段时间后，进程 A 指定的数据从硬盘上读出了，于是硬盘中断服务程序会将进程 A 强行设置为就绪状态（这也是将处于不可中断等待状态的进程改设为就绪状态的唯一方法），如图 7-34 右下角所示，执行代码如下：

```
// 代码路径：kernel/blk_dev/blk.h：
extern inline void end_request(int uptodate)
{
    ...
    unlock_buffer(CURRENT->bh);
    ...
}
extern inline void unlock_buffer(struct buffer_head * bh)
{
    ...
    wake_up(&bh->b_wait);
}
```

这样进程 A 就具备执行的能力了，但这并不等于进程 A 马上就执行。硬盘中断程序返回后，仍然是进程 C 继续执行，如图 7-34 左下角所示。

7.2.7.9 切换到进程 A 执行并处理信号

进程 C 的时间片用完了，又要进行进程切换了，这里进入 schedule 函数后，发现只有进程 A 是就绪状态，于是切换到进程 A 去执行。进程 A 开始执行后，先要对刚才从硬盘上读出来的数据进行处理，然后，此次 sys_read 就执行完了。此时，软中断准备返回，在返回之前，先要检查一下进程 A 是否有接收到任何信号（注意，所有系统调用函数返回前，都要检测一下当前进

程是不是接收到信号了，这不是特例，而是通用法则)。果然，检查到进程 A 接收到了信号，于是将对处理该信号的服务程序入口地址进行处理，以便于一旦此次软中断返回，就由信号处理程序处理该信号（现在的情况是，要将进程 B 退出的善后事务彻底处理完），然后进程 A 再从刚才 read 函数的后面继续执行。具体的执行代码我们在前面都已经说明了，如图 7-35 所示。

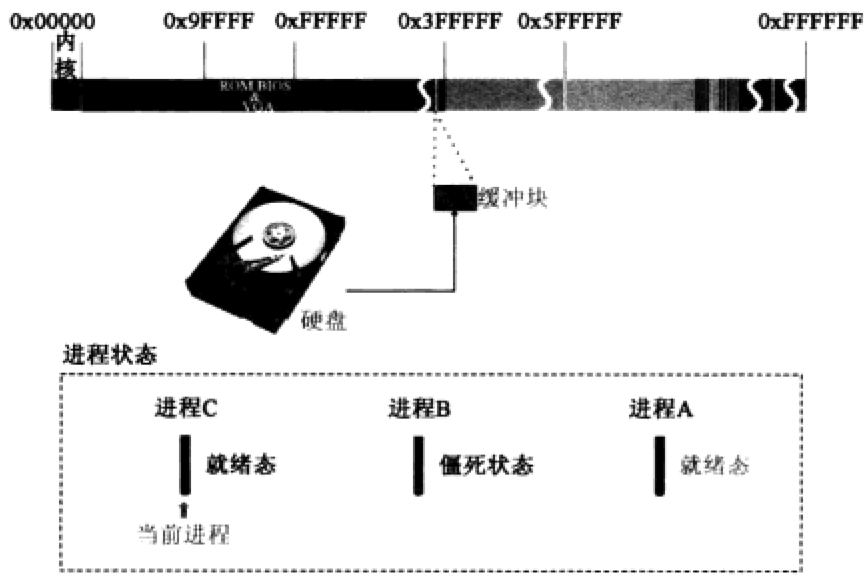


图 7-34　进程 A 被唤醒

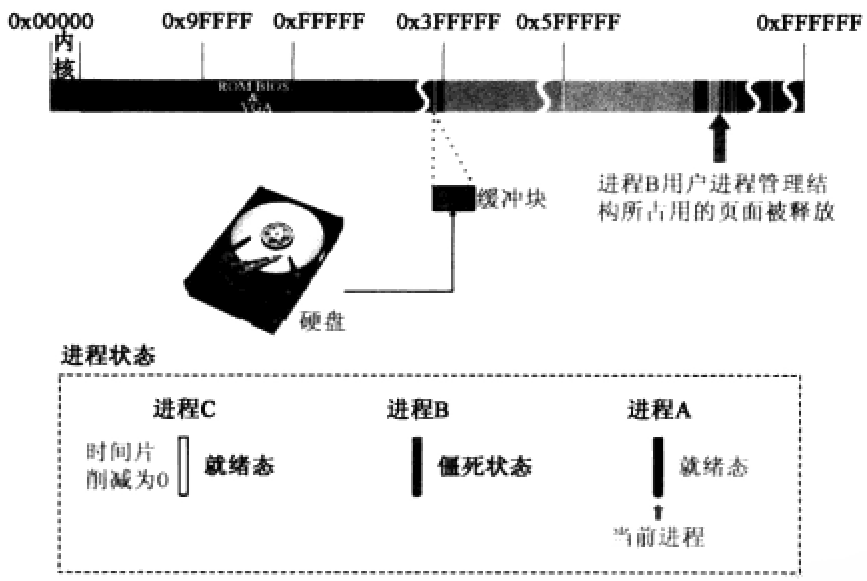


图 7-35　进程 A 执行并处理信号

由此可见：对处于不可中断等待状态的进程而言，除非直接将其设置为就绪状态，没有任何办法将它的状态改设为就绪态，接不接到信号都没有用。

7.3　小结

本章以“多进程操作管道”和“进程间发送信号”为例，对 IPC 问题进行了讲解。

在“多进程操作管道”案例中，操作系统先在文件管理表和 i 节点管理表中，为管道文件进行注册，并在内存中申请一个页面作为管道。管道建立后，两个进程就开始对它进行数据操作，其中一个进程从管道中读取数据，另一个向管道中写入数据，即便发生时钟中断打断了进程的执

行，数据也同样不会产生混乱。

在“进程间发送信号”案例中，系统调用执行完毕前，都要处理当前进程的信号，如果确定接收到信号，就进行一些设置准备，系统调用返回后，立即执行信号处理函数，然后再去执行进程刚才产生系统调用软中断后的代码。此外，操作系统每次进行进程调度时，都会对所有进程进行检测，如果进程处于可中断等待状态且接收到了信号，就会将其设置为就绪态，如果进程处于不可中断等待状态，即便它接收到了信号，状态也不会被改变。

第 8 章

操作系统的设计指导思想

普天之下，莫非王土；率土之滨，莫非王臣。

——《诗经 • 小雅》

前 7 章详细分析和讲解了 Linux 操作系统的运行原理与工作机制，本章将尝试从设计者的视角探讨操作系统的设计指导思想。

8.1 运行一个最简单的程序，看操作系统为程序运行做了哪些工作

透彻理解操作系统的好方法之一就是查看当一个最简单的程序在计算机上运行时，操作系统都做了些什么。

我们以一个 C 语言版的“hello world”为例：

```
#include <stdio.h>
void main()
{
    printf("hello world\n");
}
```

这段程序被编译、链接之后会生成一个可执行文件。我们在 Linux 操作系统上运行这个程序，最终会在屏幕上输出“hello world”。表面上看，在屏幕上显示的“hello world”都是我们写的程序的功劳，其实我们写的程序只起到了很小的作用。很明显的是，这个程序里使用了 C 语言的库函数 printf，因为本书的主题是操作系统，所以我们暂不讨论库函数。

我们概略地浏览一下 Linux 0.11 操作系统为 hello 程序的运行都做了些什么。下面的小字是对操作系统所做工作的概要描述，涉及操作系统的代码在 1 万行以上，除进程间通信之外，几乎涉及操作系统的方方面面。我们写这段文字的目的，是想让读者对运行一个最简单的程序，操作系统都做了哪些工作有一个直观的了解。

案例描述：硬盘上有一个名叫 hello 的可执行文件，该文件的源程序如前所述。

现在系统已经处于怠速状态了，用户准备通过键入一条指令 ./hello，使硬盘上该文件中的程序加载并执行，最终将 hello world 这个字符串显示出来。

第一步：用户输入命令，shell 进程被唤醒，对命令进行解析。

为实现这一步，系统将至少进行如下准备工作：

（1）用户敲击键盘后，键入的信息记录在终端设备文件 tty0 上。

如果系统要以文件的形式对终端设备进行操作，首先要构建一整套文件系统，之后加载该文件系统，以便在此基础上对文件进行操作。这套文件系统包括“超级块”、“逻辑块位图”、“i 节点位图”、“文件 i 节点”、“数据块”等。其次，还要根据文件的不同功能对数据进行分类，包括普通文件、设备文件、目录文件等。tty0 就属于设备文件，有了这些准备才有可能对终端设备文件 tty0 进行操作。

（2）敲击键盘后，还要产生键盘中断信号，系统要能够对键盘中断信号进行处理。

首先，这个中断信号会通过可编程中断控制器 8259A，所以要对 8259A 这个中断控制器进行设置；然后，信号会被传达给 CPU，CPU 要通过中断描述符表寄存器 IDTR 找到内存中的中断描述符表，再通过搜索中断描述符表找到键盘中断处理程序，并执行该程序。要实现这些操作，就要构建一整套中断服务体系，其中包括对中断描述符表寄存器 IDTR 进行设置和建立一个中断描述符表，用以和中断服务程序相挂接，然后还要编写中断服务程序，以便能够为具体的中断服务。此外，还要将这些中断服务程序与中断描述符表相挂接。

（3）中断服务程序开始执行后，唤醒 shell 进程，之后，通过进程调度机制，由进程 0 切换到 shell 进程去执行。

这需要系统建立一整套进程管理机制，就 shell 来说，要创建进程并加载 shell 程序，这样才能构建人机交互界面；同时，还要创建一个进程 0，并在其他进程都不处于就绪状态时切换到进程 0 去执行，而且只要一旦有进程被唤醒，就又立即切换到该进程执行。这个机制要适用于操作系统中的所有进程。既然要支持多进程执行，那么就还要设计一套进程轮询机制，即产生时钟中断，导致进程切换，这个机制里面又有很多的问题需要考虑，比如时钟中断服务程序的设计和 8253 定时器的设置，等等。

（4）shell 进程通过执行自己的程序从 tty0 这个终端设备文件上读取用户键入的指令信息，然后解析该指令，并准备进行相应的处理。当然，这条指令不是敲击一次键盘就能输入的，每次敲击键盘，都会重复上述动作，然后 shell 进程再次睡眠，并等待下一次键盘中断的产生。

到这里为止，系统仅仅是对用户键入的命令进行响应，正式的处理还没有开始呢。以上介绍的这些准备工作，都只是针对具体的步骤进行了最简单的介绍，这些准备工作的背后，又有着更多的准备工作，严格来讲，本书第 1 章、第 2 章中所介绍的准备工作，几乎都要派上用场，才能执行这第一步。

第二步：shell 程序解析出用户命令后，调用 fork 函数创建一个用户进程，以便对 hello 文件的程序进行控制。

系统在这里至少要为用户进程创建一套进程管理结构，每套进程都要有一套这样的结构，以便控制将来加载的程序。这套结构十分复杂，包括时间片、优先级、进程状态、进程对应的文件、进程的任务状态描述符表 TSS，以及进程的局部数据描述符表 LDT，等等。这其中的每一

项又与系统的运行有着千丝万缕的关系。比如说 TSS，它里面存放着当前进程运行时所有寄存器中的数据，一旦发生时钟中断，系统就将当前各个寄存器中的数据存储在 TSS 中，同时用即将切换到的进程中的 TSS 中的数据来设置各个寄存器中的值，最后再切换。可见，这个 TSS 中的数据是进程切换的根本保障。再比如 LDT，它里面存放着当前进程的代码段描述符和数据段描述符，这两个描述符都直接控制着进程所控制的程序，而进程运行的根本目的就是执行用户的程序。

另外，每个进程都会有 TSS 和 LDT，为了便于管理，还需要再设计一套数据结构，就是全局描述符表 GDT，所有进程的 TSS 和 LDT 的索引都存放在这个 GDT 中，系统为了方便操作 GDT 表，并进一步操作 LDT 和 TSS，还要对 CPU 中关于这三个表的专用寄存器进行设置，它们就是全局描述符表寄存器、局部数据寄存器和任务状态寄存器。

但仅有这些，也还是远远不够的，系统启动之初是实模式，各个段寄存器中都是实际的地址值，直到进入保护模式，段寄存器中的数值才变成了段选择符，这样 GDT 表才能参与应用，所以系统还要为实模式到保护模式的转换做全方位的准备工作。

以上这些只是针对 TSS 和 LDT 进行的展开分析，进程管理结构中其他成员与系统之间同样有着紧密的关系，例如进程调度的最基本方式就是通过时间片轮转，而时间片轮转的最重要参考数据就是当前进程的时间片。再例如，只有进程才能够操作文件，所以进程就要与文件全面建立关系，包括文件的 i 节点、文件管理表中的表项、进程自身的文件管理指针表，等等。

创建进程，就必须创建进程管理结构，而进程管理结构中的成员一个都不能少，都要创建并设置。除了进程管理结构，创建进程时候，还要为新进程复制页表和创建页目录项，这些都与内存页面的应用有直接关系，而内存的应用策略，又是整个操作系统中最复杂的应用策略之一。

第三步：新进程创建完毕后，加载 hello world 文件对应的程序。

要完成这一步，进程就要在两个方面进行全方位的准备：一个是文件，一个是内存。Hello world 程序一定是以可执行文件的方式存储在硬盘上的，所以，在文件加载之前一定要检测文件是否可用，主要表现在对文件 i 节点的检测和对文件头的检测两个方面。i 节点是文件的管理信息，只要涉及 i 节点，就一定离不开对 i 节点的查找，于是就要解析文件路径、操作目录文件和目录项、操作 i 节点表等，一件事情都不能少做。文件头存储在数据块中，要操作数据块又离不开逻辑块位图的支持，这样一来，整个文件系统中涉及的全部内容都要用到了。

具备了载入文件的条件后，就要将 hello world 文件载入到内存中了，这样系统就要解决所有与内存相关的问题，包括要与原来进程共享的页面解除关系，这就涉及页面引用次数、页面三级管理机制（页目录表、页表、页面）、页面数据（只读 / 可读可写）等一系列问题，系统就要为此建立页写保护等机制来解决这些问题。

但仅仅解决这些问题还远远不够，程序的加载也是很讲究策略的，其中最重要的就是缺页中断机制，即必须根据需要来分析是不是需要申请新的页面来加载程序的内容，为此又要对许多数据进行判断，这样才能确定加载的必要性，比如线性地址所对应的物理地址是否被映射到了线性地址空间内等，这就需要一套物理地址到线性地址的映射方案。另外，缺页中断机制的设计也很

有讲究，缺页并不等于一定要把外设的程序加载进来。比如，由于压栈导致的缺页同样要申请新的页面来载入数据，但这与外设一点关系都没有。这些都是缺页中断机制设计时需要全面考虑的问题。

总之，hello world 程序的加载，几乎涉及文件管理与内存管理的各个方面。而且，以上所述还仅仅是针对 hello world 这一个进程加载所进行的最基本介绍，Linux 是要支持多进程执行的，每个进程都有可能加载自己的程序，而文件和内存又是所有进程可以共用的资源，它们之间还存在着更为复杂的管理关系。比如，两个进程加载同一个 hello world 文件时，要不要共享，如何共享，共享后页面的引用计数如何计算，读写属性如何确定，等等。

第四步：hello world 程序开始执行，将“hello world”字符串显示在屏幕上。

hello world 程序加载进内存后就要开始执行了，这个程序比较简单，就是将 hello world 这个字符串显示到屏幕上。但是，即便如此，系统也要为此做很多的工作，其中最主要的就是关于显示方面的工作，比如，显卡属性如何确定，显卡是单色还是彩色；显存位置如何确定，显示在屏幕上的位置又如何确定；如果字符数量过多，要不要滚动显示，如何滚动显示，等等。这些问题都要操作系统来做，而且直接与显示器的底层交互。

从这些小字文档中，我们不难看出，即使是运行一个最简单的程序，操作系统也要作非常多的工作。我们反向思考一下，似乎也可以得出下面的结论：如果没有操作系统，即使是在屏幕上显示一行“hello world”，我们都要写大量复杂的、具有操作系统所具备的功能的程序。毫不夸张地说，没有操作系统我们甚至无法把程序加载到计算机中，更谈不上得到运行结果了。

那么，操作系统究竟为应用程序的运行都做了些什么？

经过综合分析，我们归纳出：操作系统的一部分任务是为应用程序的运行提供使用硬盘、显示器、键盘等外设的基础程序，或者说操作系统为应用程序的运行提供了对外设的支持。如果操作系统不写这些支持程序，应用程序就必须写这些程序，而且所有应用程序都要写的这部分程序的内容也都差不多，所以，我们也可以把操作系统看成是所有应用程序共有的部分。

像 Linux 这样的现代操作系统，不仅为应用程序提供了对外设的支持，还支持多个程序同时运行，这就要求操作系统不但要支持外设，还必须对运行的多个程序进行有效的组织、管理和协调，防止某个程序独占 CPU、内存、外设等资源，使得其他程序无法正常运行。此外，还要防止正在运行的程序之间相互读写和相互覆盖，确保所有程序正确运行。最关键的是，操作系统不能被应用程序直接读写、更不能被应用程序覆盖。

8.2 操作系统的设计指导思想——主奴机制

这些看似合情合理的要求背后似乎隐藏着这样一个问题：应用程序是程序，操作系统也是程序，操作系统程序凭什么能对应用程序进行组织、管理和协调而不受应用程序损害呢？

我们认为凭的是特权机制。要想让操作系统做到能够对应用程序进行组织、管理和协调，同

时又不受到损害，最有效的方法就是在操作系统与应用程序之间、应用程序与应用程序之间进行有效的分离，同时还要做到操作系统能随意访问应用程序，应用程序不能访问操作系统，应用程序之间也不能互相访问。

这意味着，操作系统必须能够做到，如果它要让某个应用程序在内存中的什么位置运行，该应用程序就得老老实实地在那运行，操作系统应在内存中为应用程序划出清晰的边界，应用程序不能越雷池半步。操作系统允许应用程序占用 CPU 运行多长时间，应用程序就只能运行多长时间，运行完这点时间后，还得将 CPU 的使用权规规矩矩交还给操作系统，没有权利私自扣留 CPU 的使用权。如果某个应用程序要想使用外设，不能直接向外设伸手，得向操作系统请示和申请。如果操作系统认为可以让这个应用程序使用外设，就让它使用；如果操作系统认为不应该让这个应用程序使用外设，就不让它使用……

在这样的特权机制下，操作系统和应用程序的关系就成了主子和奴才的关系，为了便于记忆，我们把这种特权机制称为主奴机制。

8.2.1 主奴机制中的进程及进程创建机制

8.2.1.1 程序边界与进程

为了实现主奴机制，首先要在操作系统内核程序与应用程序之间，应用程序与应用程序之间建立有效的边界。

现实生活中的物体，大部分种类都是有自然边界的，比如我们周围的房子、桌椅板凳，以及我们自身，房子、桌椅板凳都有天然的边界；我们有皮肤，这是我们的天然边界，这些天然的边界可以有效地防止人与人之间、物与物之间、人与物之间的融合，以保持独立、完整的形体，同时也保持了独立的、完整的特性。现实生活中也有一些物质没有天然的、确定的边界，比如气体和液体等。不同的气体之间因为没有边界很容易相互融合，不分你我。我们想要盛水，最有效的方法就是用杯子或瓶子之类有边界的容器。

程序代码在计算机中的情况与气体和液体的情况类似，它也没有天然的、确定的边界，这就需要操作系统人为确定边界，起到类似于容器的分离和承载作用。为此，现代操作系统的设计者提出了进程的概念，用 task_struct 数据结构来实现明确地划分边界等作用。tast_struct 是进程最主要的标志，从操作系统的角度看，进程就是运行中的接受操作系统组织、管理和协调的程序。

8.2.1.2 进程创建

从技术上讲，创建进程的方法不止一种，Linux 操作系统的进程创建采用的是对象创建模式。对象创建就是用已有的对象创建新的对象，用已有的进程创建新的进程，也就是所谓的父子进程创建机制。从本质上讲，创建进程最主要的就是创建 task_struct。父子进程创建的主要机制就是从父进程的 task_struct 复制一份作为子进程的 task_struct。

从逻辑上很容易反推出，父子进程创建机制意味着最初的父进程必须是独立存在，这就是进程 0。不难理解，进程 0 不能由父子进程创建机制创建，所以只能由操作系统设计者手工编写进

程 0 的 task_struct。有了进程 0，父子进程创建机制就可以用进程 0 作为父进程创建子进程。

有了进程，就有了主奴机制的组织、管理和协调的对象。下面我们将分析操作系统中的主奴机制是如何实现的。

8.2.1.3 操作系统的设计如何体现主奴机制

操作系统内核与用户进程之间的关系应该设计为主奴关系，以实现主奴机制。实现了主奴机制，操作系统才能稳定地运行。

操作系统的设计者使用了一整套设计方案来实现主奴机制。下面我们从操作系统设计者的角度，分三个方面来剖析操作系统的设计思想中是如何体现主奴机制的。第一个方面，进程调度体现的主奴机制；第二个方面，内存管理体现的主奴机制；第三个方面，文件系统体现的主奴机制。

8.2.1.4 操作系统在进程调度中体现的主奴机制

操作系统进行进程调度时，对待内核和进程的方式是截然不同的，准确地说，进程调度就是内核操作的。当发生时钟中断时，触发调度程序，调度程序判断当前进程的时间片是否用完，如果用完，马上进行调度，不论这个进程的工作是否完成，立即挂起这个进程，调度其他进程运行。如果是内核，调度程序判断确认后就返回，内核继续运行，直到工作完成为止，不论其占用 CPU 的时间有多长，都是如此，所有的用户进程都要停下来，一直等待内核运行结束。由此可见，系统掌握了“进程调度的权力”后，该权力只针对它的奴才——进程，而不能针对主子本身——内核。

Linux 操作系统每次分配给进程的运行时间是由若干时间片组成的，分配多少时间要由内核决定，内核决定多少就是多少，进程想要增加时间片，连商量的机会都没有。一旦时间片用完，CPU 的使用权就要交还内核。值得注意的是，这个“交还”不是通过协商或轮流坐庄，而是强行收回！如果进程运行结束，就算还有剩余的时间片，操作系统也会收回 CPU 的使用权，不会等待。

如果不采用主奴机制，而是把进程调度设计成进程自觉、主动地将执行权限上交至操作系统，那么，进程什么时候上交 CPU 的使用权以及是否上交，直接取决于进程的程序设计，操作系统几乎无法控制。更可怕的是，如果进程是恶意程序，或者发生异常僵死，操作系统很可能永远无法回收 CPU 的使用权，会导致整个系统瘫痪。

主奴机制的设计保证了应用程序占用 CPU 资源的情况最多只影响操作系统分配给它的那些时间片，这些时间片用完后，执行权限自然回归操作系统，回归之后，操作系统就可以处理进程，甚至将出错的进程强行退出。

具体的技术细节在第 5 章已经详细讲解过了。

8.2.2 操作系统在内存管理中的主奴机制

前面讲过，进程最主要的标志就是 task_struct 数据结构。在 task_struct 数据结构中，用 start_code、end_code、end_data、brk、start_stack 明确地定义了进程的边界，任何未经允许的跨

越边界行为都将被制止。有了清晰的边界，就保证了进程不能直接越界访问操作系统内核，也保证了进程间不能直接相互访问。这是主奴机制的体现。

Linux 0.11 的内核和用户进程都采用了分页机制，但是采用的管理数据却是两套：一套是针对内核使用的，范围是 0MB ～ 16MB 的全部内存空间；另一套是针对用户进程的，分页范围仅限于 1MB ～ 16MB 空间内（1MB 以内的是内核空间），如图 8-1 所示。

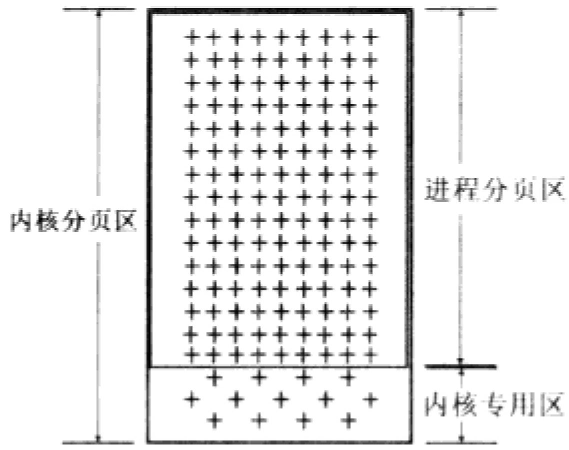


图 8-1　内核与用户区分页示意图

从图 8-1 中可以清楚地看出，只要操作系统内核代码在内核专用区，进程是无论如何也访问不到的，而内核的访问范围却囊括整个内存空间。我独占的地方你来不了，你的地方我随便去，因为你的地方就是我的地方。真是"普天之下，莫非王土"，其结果必然是"率土之滨，莫非王臣"！

此外，用户进程只能面对一个逻辑地址，不能直接使用物理地址，需要使用内存的时候，由操作系统内核转化为一个线性地址，再根据第二套 1MB ～ 16MB 的管理数据方案转化为实际物理地址。内核将整个内存划分为统一大小的内存块——页。进程运行时，操作系统给进程分配页，如果操作系统给进程分配了两个以上的页，这些页不一定是相邻的，而且通常是不相邻的。这些页具体分配在内存的什么地方，进程是不知道的，分了几页进程也不知道。准确地说，进程甚至都感觉不到分页，在进程看来，它使用的是一个连续的逻辑地址内存空间。

进程不知道自己在哪，更不可能知道别的进程在哪，根本不可能进行进程间的互访。内核代码肯定是放在内核专用区里，从图 8-1 可以明显地看出位置在进程的访问极限之外，所以进程不可能访问内核。进程存储区在内核的访问范围之内，内核就可以随意访问进程的内存空间。这是典型的主奴关系！

具体的技术细节在第 2 章和第 5 章已经详细讲解过了。

8.2.3　操作系统在文件系统中体现的主奴机制

以写文件时申请磁盘空间为例，当用户需要向磁盘写文件时，首先需要向内核提出申请，说明自己是哪个进程、自己需要的资源大小，以及资源的读写权限。内核接到申请之后，会结合当前的磁盘已占用资源、缓冲区的实际情况决定是否立即满足该用户进程的请求。如果有多个进程

都在申请资源，内核会决定让某一个程序先获得资源，而让其他进程处于等待状态，并且要对等待队列的排序做出管理，以达到内核认为的最好标准。如果当前的资源已经无法满足要求，内核还会驳回进程的申请。这里也充分体现了内核和用户进程的主奴机制，用户进程的工作一旦涉及底层，是没有权利直接向资源伸手的，需要先向内核“提申请”、“打报告”。内核掌控着各种硬件资源，负责对申请资源的众多进程进行组织、管理和协调。具体的技术细节在 4.1 节中有详细的讲解。

8.3 实现主奴机制的三种关键技术

上面我们从三个方面详细分析了主奴机制的设计指导思想在操作系统中的体现。下面，我们将讲解操作系统的设计者是靠什么实现主奴机制的。我们认为，主要依靠三项关键技术：保护和分页、特权级、中断。这三项技术有一个共同特点，都是依托 CPU 提供的硬件机制。

8.3.1 保护和分页

我们在第 1 章就讲过，Linux 0.11 打开了 pe 和 pg，即打开了保护模式和分页机制。打开保护模式后，CPU 的寻址模式发生了实质性的变化，以代码寻址为例，实模式时是 CS:IP，在保护模式下 IP 变为 EIP，更关键的变化是 CS 由直接的代码段基址变为代码段选择符，通过解析代码段选择符可获得 GDT 中指定的代码段描述符，进一步解析才能获得代码段的基址。

还有两个变化是深刻的：一个是段限长，一个是特权级。

实模式下的 CS 虽然是代码段的段基址，但 CS 只是负责看管代码段的起始位置，Intel 的 CPU 没有设计负责看管代码段结尾地址的段尾寄存器。虽然有 64KB 的段长，但不得不允许其他段覆盖，因为在实际使用中经常出现代码段远小于 64KB 的情形，为了不浪费内存，只好允许覆盖。

保护模式除了段基址之外，还有段限长，这样既兼容了实模式只有一个段寄存器的情况，又相当于增加了一个段尾寄存器。既有效地防止了对代码段的覆盖，又防止了代码段自身的访问超限，明显增强了保护作用。

对主奴机制影响深远的是特权级。

从第 1 章开始，我们多次提到 CS 的最后两位就是特权级。Intel 从硬件上禁止低特权级代码段的代码使用一些关键性的指令，如：LGDT、LLDT、LTR、LIDT。另外，Intel 还提供了机会，允许操作系统的设计者通过一些特权级的设置，禁止用户进程使用 cli、sti 等对掌控局面至关重要的关键性指令。

有了这些硬件作基础，操作系统就可以把内核设计为最高特权级，把用户进程设计为最低特权级。这样，操作系统设计者就能做到让操作系统内核可以执行一切指令，想做什么就作什么。操作系统可以访问 GDT、LDT、TR，而 GDT、LDT 是逻辑地址形成线性地址的关键，也就是说操作系统能够掌控线性地址，而用户进程则不能，用户进程只能使用逻辑地址，且用户进程的逻

辑地址要由内核转换为线性地址。物理地址是由内核将线性地址转换而成的，不知道线性地址就不知道物理地址，也就是说，操作系统内核可以访问任何物理地址。这样，对于用户进程来说，它只能“感觉”到在访问一个逻辑地址的内存空间，如同访问“真实的内存空间”一样，而实际的逻辑地址到物理地址的映射要由操作系统内核来安排，操作系统把用户进程需要访问的内存安排在内存的什么地方，完全是随心所欲，而用户进程甚至都不知道实际访问的物理地址在哪。

Linux 0.11 操作系统的用户进程线性地址空间的设计方案是：把 4GB 的线性地址空间等分为 64 份，每份 64MB，每个进程一份，每个进程的逻辑地址空间是 64MB。也就是说，在用户进程看来，有 64MB 的内存可以使用，这样 64MB × 64 = 4GB。由于用户进程能访问的空间不可能超出 64MB，每个进程的线性地址空间又没有任何交叠。这样，从理论上讲，在线性地址空间的层面，任何用户进程之间的直接相互寻址和访问都是不可能的了。用户进程之间的相互访问都不可能，更不可能访问操作系统内核了。

这样，保护模式给操作系统的设计者提供了机会，使操作系统的设计者有可能做到，用户进程不能访问操作系统内核，也不能相互访问，而操作系统内核能够访问任何用户进程。这是主奴机制的一个体现。

分页的前提是保护模式，也就是说，pe 和 pg 必须同时打开，不存在没有 pe 的 pg，可以说分页和保护是一体的。分页机制同样依托于 CPU 的硬件，在提高了内存空间使用效率的同时，也使操作系统的设计者能够实现用户进程之间不能互访，更不可能访问内核，而内核可以任意访问用户进程。

在分页机制下，理论上使线性地址等价物理地址的分页方法只有一种，下面我们详细讲解这种方法的原理和操作。

我们用一个简单的线性方程来表示线性地址和物理地址之间的关系。

$y = kx + b$

这里 x 代表线性地址，y 代表物理地址。k 是线性地址和物理地址的比例关系，因为线性地址和物理地址的单位都是字节，如果增长的方向相同，线性地址和物理地址的比例关系就是 1，也就是

$k = 1$

这样，

$y = kx + b$

就成为：

$y = x + b$

不难看出，只要

$b = 0$

就可以实现

$y = x$

也就是物理地址等于线性地址。

操作系统内核要想做到在分页机制下实现线性地址等于物理地址，就一定要把操作系统内核分页的起始位置放在物理地址的起始位置，这是关键！

回顾一下第 1 章的图 1-23 以及 5.7 节讲到的内容，才能真正地理解内核分页从内存的起始位置开始的深刻意义，其作用就是 b = 0，使线性地址等于物理地址！

对于内核来说，就是要直接面对物理内存，最直接的方式就是线性地址和物理地址一一对应。当内核需要访问内存的时候，就可以直接访问物理内存，而不是像进程那样，被操作系统绕过来、绕过去，绕得晕头转向，连自己究竟在物理内存中的具体什么位置都不知道。

为了在分页机制的前提下满足这项要求，操作系统特别设计了一套页表专门供内核使用，这套页表的值刚好可以使线性地址的值和物理地址的值一一对应，而且这个映射范围不是局限于内核的 1MB 空间本身的，而是包含所有 16MB 的空间，也就是说，内核可以直接访问任何一个进程的内存空间。具体方案的技术细节已经在第 1 章图 1-38 至图 1-40 以及 5.7.2 节详细讲解过。

用户进程只能面对一个逻辑地址，需要使用内存的时候，首先由操作系统内核转化为一个线性地址，再由内核根据第二套 1MB ～ 16MB 的管理方案转化为实际物理地址。在这里，线性地址和物理地址的转换与直接映射截然相反。

首先，用户进程内存页面的分配是从物理内存的高地址端开始的，随着程序的执行向物理内存的低地址端发展，可以粗略认为是与线性地址方向反向分配页面，可以粗略地看成是 k ＝ −1，而且 b 自然不可能为零。操作系统完全根据多进程实际运行的需要给进程临时分配页面，页面分配在物理内存的什么地方事先完全无法预料，准确地说，就是操作系统内核也无法预料，看上去很像随机分配页面。这样做的效果是将用户进程的逻辑地址与实际物理内存完全隔离，便于内核对内存做统一管理。这里就充分显示了内核这个主子可以直接操作、管理内存，对全部内存有使用、管理权限，知道每一个进程的实际内存究竟分配在哪里。同时，也充分体现了用户进程——奴才，不知道自己实际使用的内存在哪里，没有权限访问其他进程的空间。内核代码及所有进程的物理内存分布完全由内核掌控，实现了主奴机制。

具体技术细节已经在 5.3 节和 5.5 节详细讲解过。

图 8-2 是内核分页及用户进程分页的原理示意图。

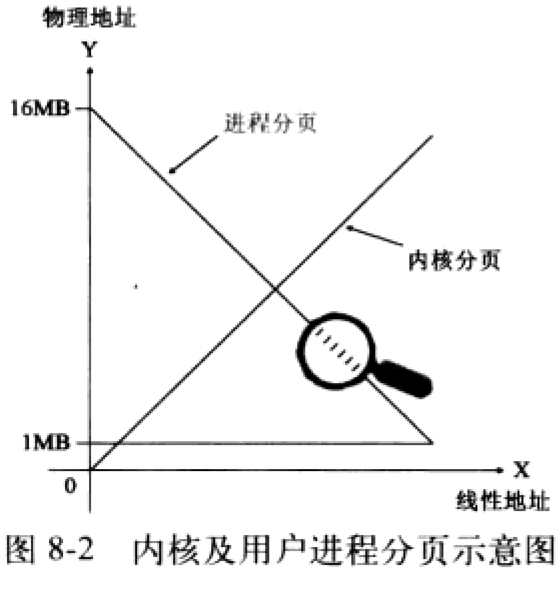


图 8-2　内核及用户进程分页示意图

8.3.2 特权级

特权级主要是依托 CPU 硬件提供的保护模式，着眼点在"段"，在所有段选择符的最后两位标识特权级，最终影响的是段选择符决定的段，这些段选择符包括 CS、SS、DS、ES、FS 和 GS。特权级影响的范围是"段"，这是关键点！

对于 Linux 操作系统而言，通常所说的内核态、用户态，准确的说法是某个代码段、或者某个数据段、或者某个栈段……当前的特权级或者是 0 级，也就是内核特权级；或者是 3 级，也就是用户特权级。

内核特权级可以在任何条件下执行所有的指令，操作系统能够做到用户特权级不能执行那些可能颠覆内核特权级的指令。就 CPU 硬件而言，既可以使所有代码都处于内核特权级，也可以使所有代码都处于用户特权级，如图 8-3 所示。

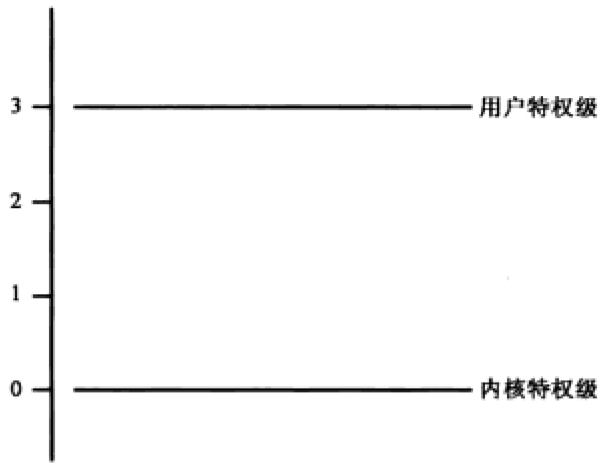


图 8-3 内核与用户特权级示意图

但是，一旦计算机中的所有代码都处于用户特权级，高特权级的指令就永远无法使用了，这也恰恰是特权级设计者的初衷。另外，所有代码都处于同一个特权级，很容易出现互访和覆盖的混乱现象，而且一旦出现混乱就非常麻烦。

操作系统的设计者都是顶级聪明人，当然不会做这种蠢事。他们会把操作系统内核代码设计为高特权级，把用户进程代码设计为低特权级。这样，内核就可以执行所有的指令，而用户进程则不能，再一次体现了主奴机制。

8.3.3 中断

操作系统和用户程序运行起来后，计算机中的用户特权级和内核特权级的代码、数据频繁交替出现，这是因为有特权级的转换。Intel 的 CPU 提供了几种转换方法，Linux 0.11 使用的主要方法是中断。通过中断和中断返回，Linux 0.11 实现了特权级之间的转换。图 8-4 形象地表示了中断所导致的特权级翻转。

下面我们详细讲解为什么中断技术能实现特权级的转换。

在我们看来，计算机中最重要的三件事就是：执行序、可识别、可预见。

我们就从执行序开始分析。图 8-5 是计算机程序执行序的分类示意图。

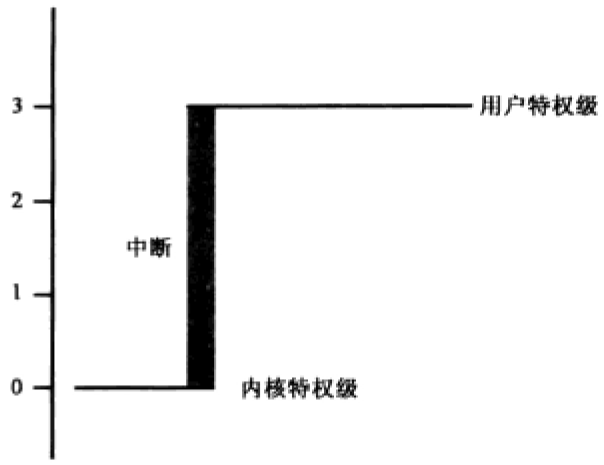


图 8-4 中断导致特权级翻转的示意图

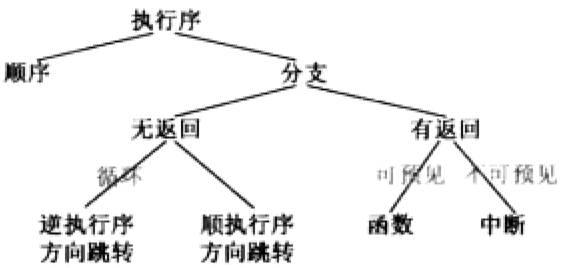


图 8-5 程序执行序示意图

首先，在计算机里的执行序有顺序和分支两种，顺序执行是靠 CPU 中的程序计数器 pc 自动累加实现的。每执行一条指令，pc 就自动累加，pc 和指令指针 ip 或 eip 联手，形成顺序执行序。除此之外，还有分支，分支又分无返回分支和有返回分支，无返回分支就是跳转。可以通过某种条件进行跳转，跳转以后不会返回，另一类就是跳转之后还要返回，这就是函数调用和中断，更广义的说法是调用子程序。这种情况下执行完子程序的操作之后，还要回到调用指令的下一行继续执行。回到调用指令下一行的前提是必须满足“能够回到调用指令的下一行”，因此需要保留执行调用指令的状态，这就是所谓的现场保护，实质上就是保存标志着 CPU 和内存运行状态的相关寄存器的值。等到调用结束后，再恢复现场，返回调用指令的下一行继续执行。从这个角度说，中断和函数有非常像的地方，对程序的设计者而言，二者的差别在于是否可预见，由此也致使玩法有重大区别。

函数的调用指令是程序设计者编写的，对程序设计者是可预见的，对于函数调用的保护动作也是可预见的。中断技术最初是因为解决外设硬件的 I/O 问题而发明的，后来又出现了模仿硬件中断的技术路线发明的软中断，方法是类似的。简而言之，中断对于操作系统的设计者来说是不可预见的，随时可能会有一个新的事件在原有的执行序不可预见的情况下切进来，打断原有的执行序，所以叫“中断”。

由于中断的发生不可预见，保护现场的任务不可能由程序员完成，只能由 CPU 硬件完成，实际上相当于是“硬件的 call”。回想一下第 2 章讲到的如下代码：

```
// 代码路径：kernel/fork.c
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
        long ebx,long ecx,long edx,
        long fs,long es,long ds,
        long eip,long cs,long eflags,long esp,long ss)
```

最后一行参数 long eip,long cs,long eflags,long esp,long ss 在调用之前根本找不到传参，不论是源代码还是反汇编代码，都无法找到传参的代码，也看不到任何压栈动作或将其他数据变成栈的操作，然而却能正确运行，实在让人费解，原因就是 copy_process 的调用源于 0x80 中断，这 5 个参数是 CPU 硬件压栈。注意，这 5 个参数的顺序与 Intel IA-32 手册中介绍的 CPU 硬件压栈顺序完全一致。这就是中断的特征。当中断服务程序执行结束，自然也要“硬件的 ret”—— iret。

不知大家是否发现，上面讲解的中断执行过程隐含着另一个特点，就是中断与普通的 call 有着很大的不同，普通的 call 似乎是沿着内存的地址“平着”“滑”到被调用位置。中断则不然，似乎是脱离了内存，通过 CPU 硬件“翻”到内存中另一处中断服务程序的位置。

普通 call 的这个特点对于编写一般的程序没有任何问题，但对于编写操作系统这样的底层系统软件却会带来致命的问题。当用户需要使用内核的系统调用代码时，如果用普通的 call 就能实现，就等价于用户程序可以随意访问操作系统内核，能访问就有可能修改，甚至可能覆盖。这严重背离了主奴机制，必将导致整个系统混乱。

中断通过 CPU 硬件“翻”到中断服务程序的特征引起了操作系统设计者的注意，当中断通过 CPU 的硬件“翻”时，CPU 硬件的设计者借此机会让 CPU 翻转了各个段的特权级，使中断成为用户特权级和内核特权级之间翻转的阶梯。除此之外，中断技术还有一个重要的特征，就是能够让硬件的信号直接切进来，对操作系统的调度而言，最重要的就是时钟中断。

如果没有时钟中断，可靠的进程调度是不可想象的。那样的话，操作系统只能设计成与用户进程协商 CPU 的使用权，只能等待用户进程自觉自愿把 CPU 的使用权交还，别无他法。

有了硬件的时钟中断就完全不一样了。时钟中断就像操作系统内核手中象征王权的斧钺，根本不与进程协商，时间一到，二话不说，强行将进程的执行序斩断，夺回 CPU 的使用权，行使主子的特权，体现主奴机制。

8.4 建立主奴机制的决定性因素——先机

到此为止，主奴机制似乎已经表述得很好了，只是有一个问题仍然无法解释，就是用户程序是程序，操作系统也是程序，用的是同一个 CPU，同一套指令集，为什么操作系统内核程序能用的指令用户程序就不能用呢？答案似乎是内核的特权级比用户程序的特权级高。如果进一步问：为什么内核程序能获得高特权级，而用户程序就不能呢？

我们认为，关键点是：先机！

计算机开机启动的时候是实模式，实模式没有特权级的概念。这时操作系统内核开始加载，正常情况下，此时不应该有 BIOS、操作系统以外的任何程序。当操作系统的启动程序打开 pe 的时候，特权级状态必须是最高特权级，否则，有一部分指令将永远无法使用。

这是非常关键的时刻！操作系统的设计者就是利用这个最有利的时间，以时间换特权，先霸占所有特权，并充分利用这些特权，创建进程。因为所有的进程都是操作系统直接或间接创建的，所以，操作系统有充分的条件和机会把进程的特权级降低。一旦进程的特权级被降低，就再

也无法翻身，除非操作系统程序代码设计错误，误把进程的特权级提上来。显然，操作系统设计者会仔细审查这类错误，并认真测试，消灭这类错误。假如操作系统的代码没有这类错误，进程一旦被创建，就再也无法获得内核特权级，只能终生为奴。由此可见，掌握先机对操作系统主奴机制的形成有着决定性作用。

反之，一些恶意程序，它们进入计算机的时机虽然晚于操作系统，但它们会想方设法利用操作系统设计上一切可以利用的漏洞，变被动为主动，抢占先机，一旦掌握先机，马上获得最高特权级，就可以为所欲为……有一类病毒，就是通过这一点利用操作系统的漏洞，想办法驻留到硬盘的系统引导区，甚至是 BIOS 中。大家根据本书前面讲解的原理就能理解，BIOS 和硬盘的系统引导区程序是先于操作系统进入内存的，这类病毒自然就先于操作系统进入了内存，一旦它们抢占了先机，获得了最高特权级，操作系统就非常麻烦了。

8.5 软件和硬件的关系：主机与进程、外设与文件

计算机分为主机和外设，主机包括 CPU、内存和总线，除了主机之外的硬盘、软驱、光驱、显示器、网卡等都是外设。因为，软件编程无法直接控制总线，所以，我们只关注主机中的 CPU 和内存。

主机进行的是运算，外设进行的是数据的输入、输出和数据的断电保存。

从根本上讲，用户使用计算机是为了解决用户的运算问题，用户运算的直接体现就是用户应用程序。从操作系统的角度看，运行中的用户应用程序就是用户进程，可以说，用户进程代表了用户的运算。

用户的运算需要外设的支持，首先需要键盘、硬盘、网络这类外设将应用程序及需要处理的数据输入主机进行运算。运算的结果需要显示器、打印机等外设输出，需要硬盘等外设进行断电保存和转移，或者驱动其他设备进行工作。以硬盘为例，操作系统将硬盘上存储的数据映射为文件，这些是我们熟知的。进一步拓展文件的概念，从外设上的数据拓展到外设本身，如键盘、显示器也被拓展为字符设备文件。可以说，文件代表用户使用的外设。

我们先展开讲解进程，再展开讲解文件。

8.5.1 非用户进程——进程 0、进程 1、shell 进程

我们先来思考一个问题，所有的操作系统都要有用户使用界面，就是所谓的 shell。Linux 0.11 的 shell 是由 shell 进程而不是由操作系统内核承担。shell 明显是一个操作系统的功能，为什么不是由内核而是由进程承担？

仔细思考一下，我们会发现，如果 Linux 操作系统只是用于个人计算机，由内核承担 shell 似乎也没什么不可以。考虑到 Linux 在服务器领域有很大的发展空间，服务器的操作系统有多 shell 的需求，这样算来，内核承担 shell 不划算，最好是由进程承担。

然而，shell 进程显然不能由普通的用户应用程序承担，比如由一个围棋程序承担 shell，显

然是不合适的。围棋程序本身是应用程序，需要 shell 加载。围棋程序自身成为 shell，看上去这个操作系统从始至终一直有一个不可退出的围棋，感觉很怪异。如果允许这样的围棋程序退出，就更可怕了，一旦围棋程序结束退出了，操作系统就再也没有 shell 啦，没有 shell 的操作系统就是一个不可使用的操作系统，有什么存在价值呢？

由此可见，shell 必须是一个专门为操作系统配套设计的特殊进程，从开始接受用户使用到关机，都不应该退出。

shell 的本质是用户界面程序，掌控的是显示屏、键盘等，这些都是外设。Linux 操作系统的进程创建机制是由父进程创建子进程，由此可推断，shell 的父进程必须有使用外设的能力，以及可用的外设环境，可以推论出父进程就应该是进程 1 那样的进程。外设肯定是由主机掌控的，所以，所有进程都必须有在主机中运行的能力，可以推论进程 1 的父进程就应该是进程 0 那样的进程。

现在，我们可以看得更清楚，第 2 章讲解的进程 0、进程 1、shell 进程的创建，就分别代表了主机、外设、特殊的外设——用户界面。这三个部分也恰恰是计算机宏观构成的三个部分。由此可见这三个进程的划分寓意深刻，如果将三个进程合并为一个进程，就不能清晰地表示这种结构。图 8-6 表示了进程 0、进程 1 及 shell 进程所对应的硬件。

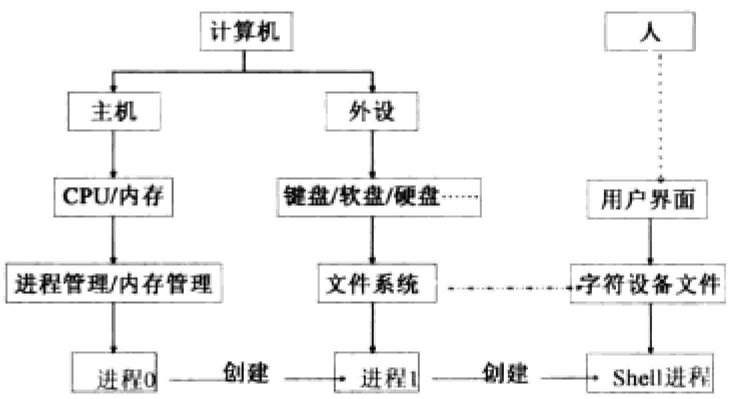


图 8-6　进程与硬件对应关系示意图

8.5.2　文件与数据存储

从前面章节的内容不难看出，虽然文件系统涉及的代码最多，几乎占了总代码量的一半，但相对而言，文件系统是最容易理解的。以硬盘为例，文件映射硬盘上存储的数据，硬盘的存储空间非常大，远远大于内存的存储量。但说到底，文件也就是一个数据的存储，硬盘可以看作计算机的数据仓库。存储工作虽然手续繁杂，但相对于运算工作而言要简单得多。而且繁杂的原因主要是因为硬盘的存储量非常大，而且要“以碎制碎”，如果用简单的管理方法，所需要的管理数据量就很大，而这部分数据既占用了硬盘的空间，又不是用户数据，为了尽可能减少这部分数据在硬盘空间的占有量，就需要用最少的管理数据管理最多的用户数据，操作系统的设计者提出了超级块、i 节点、逻辑块位图、i 节点位图等一整套的管理结构。而且这些结构还要涉及进程，导致文件系统变得非常繁杂。但总的来说文件代表存储的数据（也代表设备），所以相对运算而言

还是简单得多。

8.5.2.1 内存、硬盘、缓冲区：计算级存储、存储级存储、过渡态——缓冲区

主机中有内存，外设中有硬盘，表面上看，两者的功能都是存储，为什么还分成主机和外设？通常的说法是，内存速度快、价格高，容量小，不能断电保存，硬盘正好是互补的。

进一步追问，如果差别仅仅是断电保存，为什么用两种截然不同的管理方式来管理？操作系统管理内存用的是进程、分页、特权级，表……一大堆复杂的数据结构，管理硬盘用的是文件、i 节点、位图、块……差别非常大。

从外形上看，CPU 和内存是两种完全不同的器件。实际上，它们必须联合起来才能完成计算机中最重要的工作——计算。也就是说，计算是发生在 CPU 与内存之间的，即发生在主机中，仅有 CPU 是无法完成计算的。CPU 计算所需的指令在内存里，计算结果也要放在内存中。不仅如此，更重要的是，复杂的计算，CPU 不可能用一个指令完成，需要在内存中安排复杂的算法，如：把一个复杂的四则运算式转化为逆波兰式后，在内存中用栈操作的方式形成算法，CPU 和内存联合操作，最终取得计算结果。这个过程中，我们很难否认内存在进行"计算"，内存有存储功能，更有计算功能，是计算级存储。再看看硬盘，虽然功能也是存储，但丝毫看不出计算的迹象，是非常纯粹的存储，是存储级存储。

计算级存储比存储级存储多了"计算"，计算比存储复杂，管理信息自然要多一些。同样是管理文件，同样是用 i 节点这套方法，内存中的文件管理信息要比硬盘上的多出一些。硬盘上的文件管理信息就是为了存储，找得到、不出错就行。内存中的文件管理信息就不一样了，除了这些要求外，还要执行查找等操作，查找本身就是计算，所以内存中的管理信息是带有"计算"意义的文件管理信息。硬盘上的文件管理信息就是一本简单的数据"库存账"，硬盘本身并不执行查找运算，查找运算发生在主机中，所以硬盘不需要额外的运算管理信息。

让我们上升一个高度再来看看计算，可以看出这里说的计算也可以分为两类，一类计算是用户进程的计算，也就是用户程序的计算；另一类计算是内核为文件系统的运行所做的计算，这类计算与用户进程计算没有直接的关系。为了更容易看清楚，我们称参与用户进程计算的内存为全运算存储，称参与内核为文件系统的运行所做运算的内存为半运算存储，称在内存中完全模拟外设、没有任何运算的内存为无运算存储。

有了全运算存储、半运算存储、无运算存储的概念，就容易理解另外一个概念——缓冲区。

缓冲区在内存，是介于全运算存储和无运算存储之间的过渡态。并且，在操作文件过程中，例如在目录文件中查找某个目录项的操作，也是在缓冲区完成的。由于其具体操作是进行字符串比较，这肯定是一个运算过程，这些运算明显不是用户程序的运算，所以缓冲区是半运算存储。

用这个观点审视内存，可以发现内存中的"超级块管理表"和"i 节点管理表"（它们常驻于内核数据区），"逻辑块位图"和"i 节点位图"（它们常驻于缓冲区）等文件系统管理结构明显也是服务于半运算的。我们可以把这些管理结构所占的内存空间（无论是处于内核数据区还是缓冲区）统称为文件系统专用缓冲区。对比通常意义的缓冲区，我们可以看得更清楚，两个缓冲区都由内核控制、操作。通常意义的缓冲区针对的是进程，文件系统专用缓冲区针对的是文件系统。

反过来看，如果没有这两个缓冲区，外设的数据与内存直接交互，操作系统就得在本应是用户进程自己进行计算的全运算存储空间中进行针对文件系统的查找等计算，全运算与半运算搅在一起非常混乱。更麻烦的是，用户进程的全运算源于用户程序的代码，文件系统的半运算源于操作系统内核代码，内核代码和用户代码处理的数据都要在用户进程所属的内存空间操作，有悖于主奴机制。

再者，缓冲区还有一个作用就是共享，如果缓冲区中有一个进程读入了某个文件，其他进程读同样的文件时，可以共享缓冲区中的文件。没有缓冲区，只能每个进程读自己的文件，这可能造成内存中有同一文件的多份拷贝。换句话说，缓冲区有共享内存中只有一份拷贝的含义。

用上述概念考察一下虚拟盘，不难看出虚拟盘就是在内存中简单模拟外设，这类内存空间的特征是无运算存储。这类内存空间映射的不是文件，而是外设，比如软盘。不论这个软盘实际存储的数据有多少，哪怕只有 1KB，也要把整个软盘映射过来，显然浪费了内存空间。又由于虚拟盘是无运算存储，所以用户进程真正使用的时候，还要再经过半运算存储倒一遍手，所以应尽量不用无运算存储。

8.5.2.2　缓冲区的设计指导思想

在设计操作系统的缓冲区时，要求确保多进程读写数据的正确性，以及尽可能高的效率。内存之间的数据交互速度比内存与硬盘之间的数据交互速度快 2 ～ 3 个量级，缓冲区就在内存中，从数据流向的角度看，缓冲区处在用户进程与硬盘之间。为了实现正确、高效的要求，缓冲区的设计指导思想就是：1）要使数据的读写有序；2）让数据在缓冲区中停留的时间尽可能长，能用缓冲区中的数据，尽量用缓冲区中的数据。缓冲区中实在没有用户进程所需要的数据了，再从硬盘上读取数据到缓冲区。操作系统中与缓冲区有关的设计，都直接、间接地体现了这个指导思想。

为了实现这个设计指导思想，Linux 0.11 设计了一套数据和相关的函数，这些数据包括 hash_table、b_count、b_lock、b_dirt、*b_data、b_dev、b_blocknr、b_uptodate、b_wait……

用 *b_data 指向与进程进行数据交互的缓冲块，用 b_dev 和 b_blocknr 指定将要读写的文件的一个数据块所在的设备号和块号（以后我们简称为硬盘块），并将这一对缓冲块、硬盘块挂接在 hash_table 上，形成绑定关系，以此类推，hash_table 将所有需要读写的硬盘块与对应的缓冲块绑定，形成如图 8-7 所示的管理关系。

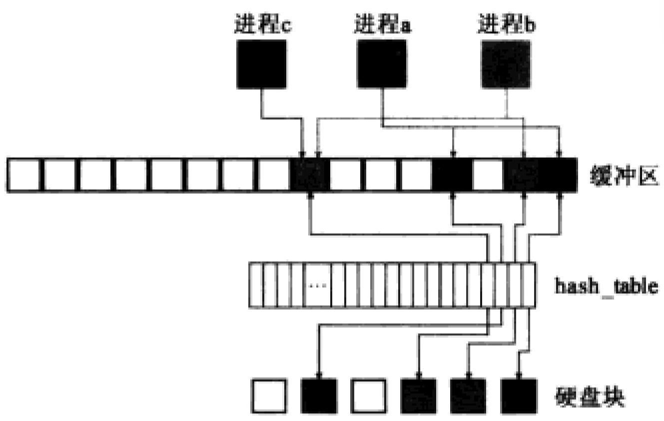


图 8-7　缓冲区、哈希表与硬盘数据块的管理对应关系示意图

当用户进程要进行文件读写操作时，并不一定是要对文件的所有数据都读写，操作系统通过

对文件的分析，确定要操作哪个硬盘块，用 b_dev 和 b_blocknr 标识。为了最大限度地提高缓冲块数据的复用性，此前的文件读写任务完成后，与硬盘块对应的缓冲块中的数据不会立即被清除掉。所以，执行新的文件读写任务时，操作系统会先在缓冲区的管理结构 hash_table 中查找，看看记载的和缓冲块对应的硬盘块与操作系统确定需要读写的硬盘块是否相吻合，只要相吻合，就证明需要操作的硬盘块的数据，很有可能不需要读盘了。根据能用缓冲区中的数据就尽量用缓冲区中的数据这个指导思想，能用现成的数据就用现成的。

然而，b_dev 和 b_blocknr 这两个字段吻合，只能说明缓冲区中有操作系统需要操作的硬盘块对应的缓冲块，并不等于缓冲块中的数据一定可以使用，因为这个缓冲块中的数据，有可能已经无效，比如说某个文件的内容已经全部被删除，那么这个文件“残存”的缓冲块中的数据还在，缓冲块与硬盘块的绑定关系也在，但已经不能再使用了。

为了解决这个问题，操作系统设置了一个 b_uptodate，当它的值为 1 时，说明缓冲块中的数据是有效的，可以直接使用，不用再次读盘了，如果为 0，说明该缓冲块中的数据无效，不能直接使用，还需要从硬盘块中读出新的数据，缓冲块才能使用。b_uptodate 对于读操作尤为重要，在硬盘块中的数据读入缓冲块后，操作系统的中断服务程序，就把 b_uptodate 设置为 1，表示该缓冲块中的数据此时已经有效。在新申请一个缓冲块时，b_uptodate 就会被设置为 0，表示缓冲块中的数据无效。

在操作系统看来，用户进程将数据写盘，其实就是操作系统将用户进程的数据写入缓冲区。缓冲区中的数据什么时候真正写到硬盘上去，由操作系统决定。换句话说，用户进程的数据写盘是分两步进行的，第一步，操作系统将用户数据写入缓冲区，并尽可能地停留在缓冲区，尽可能复用。第二步，操作系统适时写盘。这两步通常不是连续执行的，中间可能会有一个停顿。为了让操作系统在停顿后仍能够高效地同步缓冲区中的数据，且能避免不必要的同步，于是设计了 b_dirt，其作用就是标识了此前操作系统曾将用户进程的数据写入缓冲块中，改变了缓冲块中的数据。将其值设置为 1，表示所管理的缓冲块中的数据需要同步到硬盘上。等到操作系统将缓冲块中的数据同步到硬盘后，把该字段改为 0。

注意，写操作和读操作还不太一样，如果是读操作，而且缓冲块中没有现成的数据，那就只有从硬盘块中读取了，而且是立即读取，因为用户还在等待使用这些数据。写操作则不同，用户进程并不知道所谓的写盘其实只是将数据写到缓冲块中，缓冲块中的数据什么时候同步到硬盘，完全由操作系统酌情而定。在开始同步工作前，即使 b_dirt 已经为 1，如还需要往这个缓冲块中继续写入数据，仍然可行。将来操作系统只同步最终的数据，这也是缓冲区设计指导思想的另一种体现。

为了保证数据读写的正确性，必须确保数据读写的有序。比如，不能在缓冲块数据同步到硬盘块的同时，还在向这个缓冲块中写新的数据。b_lock 就起这个标识作用。在同步缓冲块之前，必须把这个缓冲块加锁，即 b_lock = 1，操作系统内核见到这个标志，就不再将进程的数据写入这个缓冲块中，这样在同步的过程中，缓冲块中的数据不会发生任何变化，确保同步结束时，缓冲块与对应的硬盘块中的数据是一致的。也就是说读写缓冲块与同步数据这两步操作不能同时进行，当 b_lock 设置为 0 时，只允许操作系统进行进程与缓冲块间的数据交互，当 b_lock 设置为

1 时，只允许操作系统进行硬盘与缓冲块间的数据交互。这样就能避免同时操作。

当缓冲块被加锁时，仍有其他进程需要与被加锁的缓冲块进行数据交互的可能，由于此时操作系统禁止任何进程与被加锁的缓冲块交互数据，所以操作系统只有将这样的进程挂起，切换到其他进程去执行，并用 *b_wait 指向被挂起的进程，以便缓冲块解锁后唤醒被挂起的进程。当需要与被加锁的缓冲块进行数据交互的进程不止一个时，*b_wait 实际上指向的是最后一个申请与被加锁的缓冲块进行数据交互的进程，其余的进程按序形成一个隐含的队列，如图 8-8 所示，其中等待队列如图中左上角所示。

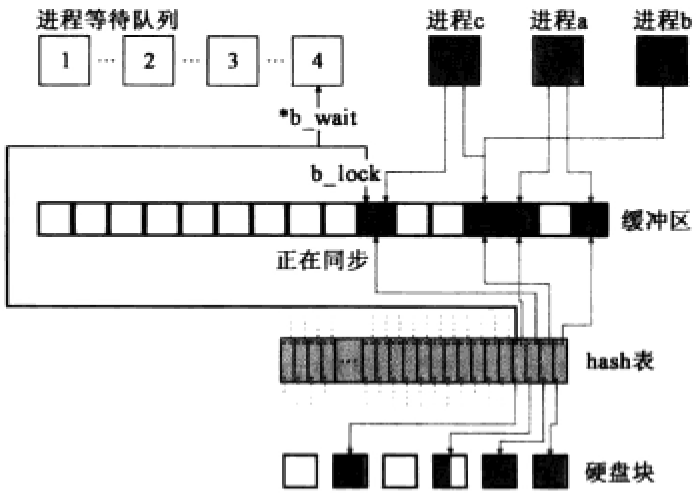


图 8-8 多进程访问设备文件时的状态示意图

当一个进程需要与硬盘进行数据交互时，操作系统首先遍历缓冲区的管理结构哈希表，如果通过哈希表找到现成的缓冲块，哪怕它还被其他进程使用着，只要数据有效，就先用这个现成的。用现成的就不用从硬盘块中读取数据，比操作硬盘划算得多。实在是找不到现成的，再申请一个空闲的缓冲块。b_count 就是缓冲块是否处于空闲状态的标志。事实上，可能有多个进程提出申请，需要与同一个缓冲块进行数据交互，每增加一个提出申请的进程，b_count 就加一，反之就减一。如果最后被削减为 0，说明这个缓冲块没有被引用，它就是空闲缓冲块。

有了这些措施，就从体系上保证了用户进程所要读写的数据与硬盘对应数据的一致性，并且实现了尽可能的高效。

下面我们讲解一下文件系统和进程的另一个连接点——管道。

8.5.2.3 利用文件系统实现进程间通信——管道

管道是进程间通信的一种方式，位于内存，是内存中的概念，奇怪的是管道不是用内存管理的方式进行管理的，而是用文件系统的管理方式来管理的，这是为什么？

进程管理的设计指导思想，就是要使进程之间完全独立和隔离，保护模式就是根据这个要求设计的。进程间通信意味着数据要跨越进程边界流动，如果采取直接交互的方式，显然违背了这个设计指导思想。怎样做才能使数据合理地跨越进程的保护边界进行流动，既实现进程间通信，又不破坏操作系统对进程边界的保护？

仔细分析，我们就会发现对于进程来说，文件是一种每个进程都可以访问的资源，换句话说，文件对进程而言是共享的。如果将进程间需要通信的数据以文件的形式作为中转站，多个进

程同时访问一个文件，有的进程写数据、有的进程读数据，就可以实现进程间彼此的数据传输。这么做既满足了进程间独立、隔离的基本思想，又实现了进程之间通信的功能。

然而，文件代表的是外设，CPU 与外设的通信速度比 CPU 与内存通信的速度慢 2 ～ 3 个量级。操作系统既然可以在内存中虚拟软盘，肯定也可以虚拟文件。在内存中虚拟一个专门用作进程间通信的文件，这就是管道。操作系统有了管道，就实现了既以文件作为进程间通信的中转站，又享受内存级通信的速度。由于管道来源于文件，管理方式自然与文件相同，这就是管道为什么通过文件系统管理的道理。

8.6 父子进程共享页面

当父进程创建子进程的时候，操作系统先将父进程的全部管理数据结构复制给子进程，在子进程加载自身的代码之前，分享父进程的代码，等到加载自己的代码后，才切断共享父进程代码的关系。为什么不在子进程刚刚创建完毕就切断这个关系？

因为此时的子进程没有任何自己的代码，加载自己代码的工作，也是需要代码的，这份代码按照 Linux 的规则，只在父进程中有，所以如果不能共享父进程的代码，就连加载自身代码的工作也无法完成。

共享父进程代码的机制为很多服务器程序提供了方便。既然允许共享，就应该允许子进程实实在在地执行父进程的代码，这样就面临父子进程同时使用同样的代码、数据的局面，极有可能造成数据混乱，为了避免这种情况的发生，顺理成章的提出页写保护机制，具体技术细节在第 5 章第 8 节讲解过。页写保护机制的设计指导思想是防止多进程访问共享数据导致的数据混乱。以此类推，对于所有访问共享数据（包括内存的、外设的）可能导致的数据混乱，解决的基本思路与此大致类似。

8.7 操作系统的全局中断与进程的局部中断——信号

我们在前面已经反复提到中断，中断对操作系统而言，其重要性怎么强调都不过分。下面，我们沿着中断的技术路线向前延伸，分析中断与 TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE 之间的关系。

在本书的第 1 章就讲到过 cli 关中断，我们知道 cli 可以使整个操作系统不能收到中断信号，等于关了整个操作系统的中断。

TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE，虽然它们名字里含有中断字样，却看不出它们和通常意义上的中断有什么直接的关系。

跟踪 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 的使用，可以发现与它们关系紧密的是信号，为什么和信号有关系的参数却起了一个和中断有关的名字？

回顾中断技术可以知道，中断技术发明的最初动机是为了避免操作系统频繁的主动轮询外设

的 I/O 状态，空耗主机资源。中断技术使操作系统由主动轮询变为被动响应，极大地降低了 I/O 过程中主机资源的开销，提高了运行效率。

对比分析中断和信号，可以看出信号明显是在模仿中断的技术路线，使进程间的沟通由主动轮询变为被动响应，同样大幅度减少了进程间沟通引起的操作系统的消耗，提高了整体运行效率。比如，shell 进程创建了一个子进程，如果子进程退出，理论上，应该由 shell 释放子进程的进程管理结构，为子进程的退出做善后工作。问题是，shell 怎么知道子进程要退出呢？很容易想到的方法是询问子进程是否退出。如果 shell 创建了几十个子进程，依照这个方法，就要对每一个子进程进行定期轮询，查看是否有子进程退出，而且不论有几个子进程要退出，即使一个要退出的子进程都没有，shell 为了及时处理子进程的退出，仍然要频繁地轮询子进程，这与中断技术发明之前，主机频繁轮询外设的 I/O 情形极其类似。操作系统的设计者借用中断的技术路线，设计了模拟中断的信号。我们对比一下就会发现，两者的技术路线非常地相似。两者的对应关系如图 8-9 所示：

中断	信号
cli	TASK_UNINTERRUPTIBLE
sti	TASK_INTERRUPTIBLE
IDT、中断向量	sigaction
中断服务程序	信号服务程序
……	……

图 8-9　中断与信号的对比示意图

可以发现，两者对称性很强，有很强的可比性。区别在于中断是针对整个操作系统的，信号是针对进程的。我们甚至可以把通常的中断看成是整个操作系统的“全局中断”，信号是进程的“局部中断”，这样可以更清楚地看到信号的本质及精髓。读者可以利用对比的方法，深入理解、掌握信号以及 TASK_UNINTERRUPTIBLE、TASK_INTERRUPTIBLE。

8.8　小结

操作系统设计指导思想的主线就是主奴机制，它是操作系统设计指导思想的核心！整个操作系统任何一个部分的设计都不能和主奴机制相悖。毫不夸张地说，全部操作系统的设计都是围绕着主奴机制展开的。

到此为止，我们已经把操作系统设计指导思想这一章的内容讲解完毕。当然，一个操作系统是非常复杂的，要想设计出一个可用的操作系统，仅凭这一章的内容是远远不够的，但本章的内容对帮助读者站在操作系统设计者的角度全面理解、把握操作系统仍然是充分的。

结束语

现在是全书的结尾，很高兴能在此和您见面。根据我们多年的教学经验，能在这和您见面，说明您在操作系统方面的水平已经不可小视！因为操作系统实在是太难了，绝大部分读者中途就已经放弃了。如果您还感到意犹未尽，就请您再回过头来重新阅读，细细品味！

“新设计团队”简介

本书由中科院的指导老师杨力祥先生带领的“新设计团队”共同完成，该团队一共分为 2 个小组：梁文峰组和武若冰组。团队成员介绍如下：

杨力祥：中国科学院研究生院信息学院教师。

梁文峰组：梁文峰，陈大钊，刘天厚，冯克。

武若冰组：武若冰，宋琦，张勇，袁泉，李坤龙，袁胜。

其中绝大部分成员是中国科学院研究生院已毕业的高材生，袁胜是中国移动集团卓望公司的高管。

此外，北京石油化工学院的在校学生杨丹雪参与了本书绘图的调色工作。

作者的话

梁文峰

操作系统的源代码错综复杂，千头万绪，但在我看来，操作系统并不是一堆源代码，而是一部精密的机器。看代码之前，眼前会先浮现出一部正在运转的机器，而代码提供的就是这部机器的“设计图纸”，它说明了：这部机器启动过程中要打开哪些开关，以及它们打开和关闭的先后顺序和相互之间的影响等。随着了解的不断深入，整部机器的架构和设计的细节在脑海中变得越来越清晰，在此过程中，有时候会发现自己对机器的某个部件的运转情况不太清楚，于是就会根据自己对机器情况的了解推测出其他应该会有与该部件的运转相配合的其他部件，于是拿出“设计图纸”对照，结果与推测的完全一致。

在了解这台机器的过程中，也会不断思考一些问题：比如，各零部件为什么要这样设计，为什么要以这样的顺序启动，启动后为什么要这样响应某些操作，更重要的是，这一切都是唯一的吗，有没有其他可能，后果又会是什么？

在我的导师杨力祥老师的帮助下，我不仅对操作系统中“是什么”有了全面的认识，而且对“为什么”有了体系性的提高。在我看来，理解“操作系统”这种机器的原理的过程与理解“汽车”、“坦克”、“战斗机”等这些机器的原理的过程在本质都是一样的。

本书是我们对这部机器的运转情况及其本质原理的生动描述，愿与您分享。

陈大钊

用"图"说话，这是本书的最大特色。的确，当你试图去描述某种有着复杂结构的事物时，图形远比文字更能让人理解。

有时候说了半天，还真不如画个图来得简单。书中包括 339 张图，每张图都是精心设计和绘制的，每条线的像素宽度和位置都是经过精确计算的。毫不夸张地说，这本书不仅是在讲解操作系统，更是在绘制操作系统！本着对读者负责的态度，本书在所有技术细节上都力求完美！

刘天厚

我很幸运在杨老师的指引下加入了"新设计团队"，又很幸运地与新设计团队的伙伴们一起共同努力完成了本书的创作。虽然操作系统是电脑中最最复杂的程序，几乎所有讲解操作系统的书都让人望而生畏，但是本书不是一本让你只看几页就看不下去并束之高阁的书，它极有可能是一本会被你翻烂的书。我们以抽丝剥茧的方式对 Linux 操作系统的内核设计原理进行了剖析，希望能帮助大家从宏观到微观上去认识整个操作系统。虽然 99% 的程序员一辈子也不会对操作系统的架构和设计进行修改，但是我们希望每个程序员都能从本书中或多或少地得到一些启发，成为那 1%。

冯克

操作系统在计算机中的地位举足轻重，但是它的学习难度也是非常大的。很多人学习操作系统的信心和决心，常常会被各种学习资料中密密麻麻的文字和庞杂的程序代码而击溃。

我所在的"新设计团队"在研究 Linux 操作系统时以它的真实的运行时序为主线，从一个全新的视角对操作系统进行了重新审视，于是我们的研究在短时间内取得了显著的成果。

我们在研究的过程中以图解的方式勾勒出了操作系统的真实运行时序，并根据这种思路完成了本书。通过书中精心绘制的、能真实反映操作系统运行时序的 300 多幅图，即使你不看操作系统的源代码，也可对操作系统的运行有初步的了解。当你深入分析每一幅图的细节时，结合与图相关的说明文字，你将会发现操作系统的原理与机制与真实的运行代码是无缝衔接的。如果理解了本书中的内容，操作系统不将再是虚无缥缈的，它的每一个动作都能在源代码中找到相应的实现。

希望此书的与众不同的讲解方式能给你带来全新的学习体验，希望它能帮你真正学懂操作系统，少走弯路。

武若冰

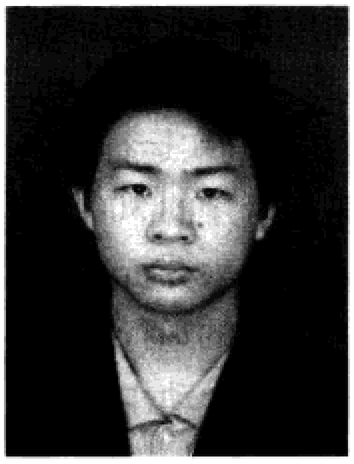
不懂汇编语言和 C 语言能看懂操作系统的源代码吗？你可能认为我在开玩笑，不过这对我来说并不是玩笑，而是事实。我是环境工程 / 环境化学科班出身，连谭浩强先生的入门级 C 语言教程都不曾读过。一年多前，当杨老师说让我和大家一起研究和学习操作系统时，那时我也曾彷徨过。好在杨老师的方法很独特，在他的指导下，我很快就入了门。书中的内容是我学习操作系统时的学习方法和思维方式的真实写照。

当你手捧这本书时，看过我的学习经历之后，你是不是对自己学习操作系统更加充满信心？本书独特的讲述方式、精心计算与绘制出的图片，以及对操作系统设计原理的精辟的讲解，将会成为你操作系统探秘之旅的指南针。

宋琦

这本书里面有很多内容都是其他同类书中不曾探讨的，比如：操作系统是如何进入怠速过程的？ main 函数竟然不是操作系统的起始点？内核程序和用户程序的等级为何如此森严？操作系统中居然还有“主奴机制”？这些结论不但生动有趣，而且对于理解操作系统有重大的意义。

然而，这并不仅是一本描述结论的书。本书真正的精华是它呈现出的一种体系性的思考方式，这种思考方式指导我们在纷繁复杂的操作系统源代码中理清了它的核心设计思想，然后我们根据这种思想构建了自己的理论体系，以指导实际工作和解决具体问题。这种思考方式不仅适用于操作系统，而且它是一切原发性思考的源动力。这种思考方式是对几十年来我们只学科学结论却忽略科学探索方式的填鸭式教育的突破。对操作系统的理解只是本书的作者们在这种思考方式下解决具体问题的一个很小的案例。这段经历犹如一段美妙的旅程，是一个充满了推论、探索、假设、证伪、构建体系的过程。如果你现在已经准备好了与我们一起分享这个过程，那么，祝你旅途愉快！



张勇

众所周知，操作系统是 PC 中最基础的软件，同时也是最复杂的软件之一，IT 从业者都以精通操作系统为荣。

然而，无数次地翻开操作系统的书籍，无数次踌躇满志地阅读 Linux 的源码，无数次的迷茫，无数次的痛定思痛，最终却是无数次的无奈。你是否如我一样，找不到头绪，被操作系统书籍中玄之又玄的理论搞晕，被庞杂的 Linux 源码所折磨？

操作系统理论的根源是什么？理论是如何指导实现的？代码又是怎样体现理论的？它们之间的相互关系是怎样的？面对操作系统庞杂的体系及复杂的关系，我们通过何种方式来驾驭它？你是否如我一样一筹莫展？如果是，请你翻开本书吧，你会发现真正理解操

作系统的设计思想和实现原理并没有想象中的那么难。我相信，跟随本书的讲解思路，你不仅能找到上述问题的答案，而且还能掌握体系性的思维方式，这对你今后研究任何复杂的事物都有莫大的益处。

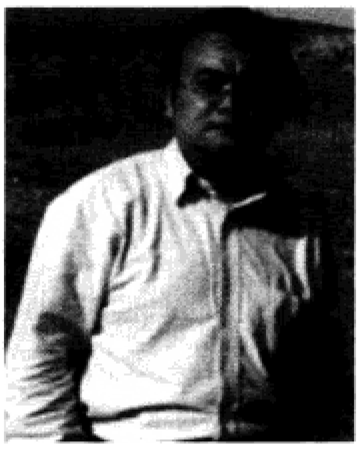
袁泉

我从小就有一个梦想，这辈子如果要是能在计算机领域做出哪怕一点点事，就不枉此生。带着这个梦想我一路走进了中国科学院研究生院，通过努力终于成为"新设计团队"中光荣的一员。写书的过程对于我来说不仅仅是像在解读一幅充满玄机、线索复杂、设计精妙的藏宝图，而更重要的是对我的人生进行了一次洗礼。通过做一件实实在在的、高水平的、有着巨大意义和格局的事，会让你所有的优秀和不优秀的品质得到一次全面且刻骨铭心的检阅。我忘不了那一次次充满严谨和灵感的团队会议，忘不了动辄连续几十小时盯在显示器前画图，忘不了杨老师给予我的那一次次耐心的、语重心长的人生教诲……这些都是伴随本书的诞生赐予我的宝贵人生财富，也将伴我迎接人生的下一次挑战。

李坤龙

从事硬件开发的朋友们有必要去学习和理解操作系统吗？我本人刚好是从事硬件 FPGA 开发工作的，参与撰写本书的经历让我感觉到，如果能够从体系上去理解操作系统，无论是从事嵌入式软件驱动开发的程序员，还是计算机硬件接口设计人员，都会获得方向性和系统性的指导。

杨力祥老师的思想深邃、方法独到，而且自成体系，是我见过的在世的大师级人物。如果你对体系、数学、物理、程序设计、操作系统、编译原理等感兴趣，我要祝贺你，你现在还有机会来到中科院研究生院聆听大师的教诲。



袁胜

读懂了操作系统的每一行源代码就是真正地理解了操作系统吗？能背得出每一个操作系统的函数就算真的明白了操作系统的设计原理吗？相信你在看这本书的过程中会有很多次"哦，原来是这样啊！"的恍然大悟，甚至会进一步产生想要自己去修改和完善操作系统源代码的冲动。

与市面上所有关于操作系统的书相比，本书的视角和写作方式都非常独特，但又是那么的自然，带给我们一种全新的学习体验。本书对计算机从加电启动，到操作系统准备就绪，再到运行用户程序的整个过程进行了详细的讲解，不仅有精辟的源代码分析，还有直观、通俗易懂的图解说明。

我认为这本书更深层次的内涵是它极具"驾驭力"的传授方式：以计算机学科中最为复杂的

操作系统为实例，向大家展示了，当面对浩繁纷杂的难题时，如何以一种系统性的思考方式去探究问题的本原。这本书不仅解读了操作系统的理论体系，更是传授了解决难题的创造思维方式和方法。通过阅读本书，相信读者不仅可以更深刻地理解操作系统的设计原理，而且还可以领悟到这种驾驭力所包含的功力，这种功力的提升不正是我们在具体技术知识之外的更深层次追求吗？我本人是这本书的最早受益者之一，通过参与这本书的编写，我在管理方面的功力得以提升，同时我尝试着在实际的日常工作中不断地去运用这种“驾驭力”，不断地感悟着它的精妙所在。以阿尔卑斯山风景区的一句精典广告语结束我的话：“请慢慢走，别忘记欣赏！”

联系作者

如果你对本书有意见或建议，欢迎你通过邮箱：xsjlinuxOS@163.com 与作者联系。

附录　搭建 Linux 0.11 系统环境

为了更好地学习 Linux 0.11 的内核，我们需要搭建一个基于 Linux 0.11 的操作平台。 这个操作平台包含三部分内容：

- boch 仿真器：boch 是不错的调试系统软件的工具，你可以从 http://bochs.sourceforge.net/ 中获取最新的版本。
- Linux 0.11 根文件系统：因为 Linux 0.11 的内核只支持 Minix 类型文件系统，所以该文件系统必须是 Minix 格式的。该文件系统中包括了 Linux 0.11 启动过程中运行的脚本及命令文件。根文件系统可以是软盘根文件系统，也可以是硬盘根文件系统。由于 Linux 早期的都是从软盘启动的，因此我们选择软盘根文件系统。

可以从 http://www.oldlinux.org/Linux.old/images/rootimage-0.11-20040305 获得。

- Linux 0.11 内核启动映像文件：就是内核代码编译后生成的 image 文件。在生成 image 文件时，必须在内核代码中指定从软盘挂载根文件系统，这样才能与软盘根文件系统挂接上。可以从 http://www.oldlinux.org/Linux.old/images/bootimage-0.11-20040305 获得。

一、boch 仿真器的安装及配置

首先下载 Bochs-2.4.exe 并安装完成，将 rootimage-0.11-20040305 和 bootimage-0.11-20040305 保存都在 Linux0.11 目录下。

然后，在 linux0.11 目录下新建一个 boch 的配置文件 bochsrc_fd.bxrc，内容如下：

```
megs: 32
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
vga: extension=vbe
floppya: 1_44=bootimage-0.11-20040305, status=inserted
floppyb: 1_44=rootimage-0.11-20040305, status=inserted
boot: a
log: bochsout.txt
mouse: enabled=0
ips: 15000000
vga_update_interval: 150000
i440fxsupport: enabled=1
```

二、运行

双击 bochsrc_fd.bxrc 文件，显示界面如图 1 和图 2 所示。

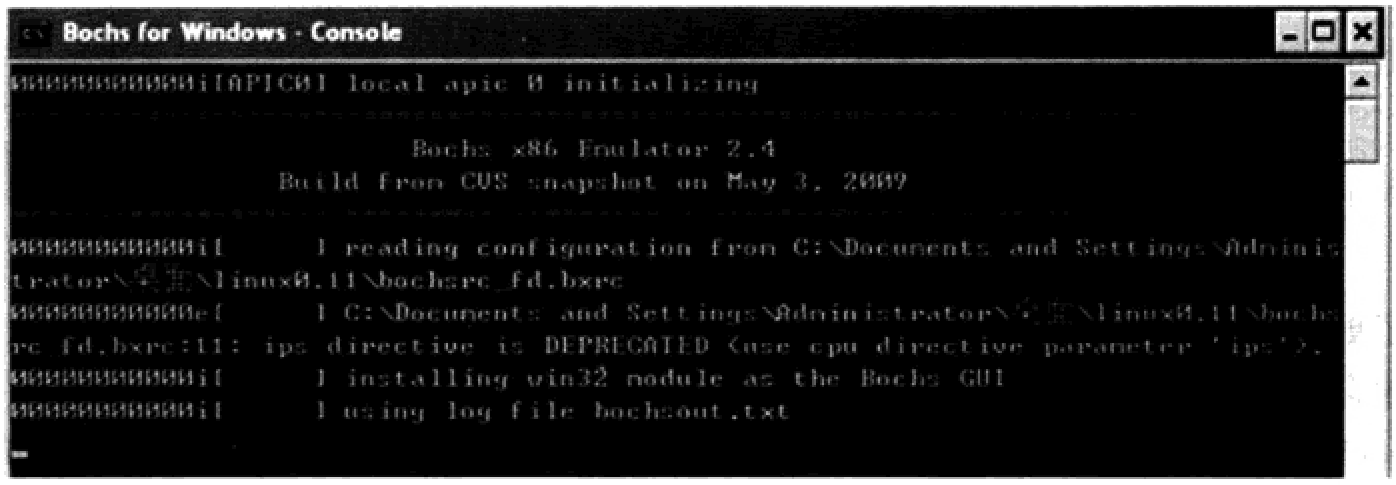


图 1　辅助控制台

图 2　主显示界面

当出现“Insert root floppy and press ENTER”时，说明此时内核要加载根文件系统。由于在内核映像 bootimage-0.11-20040305 中是从第一个软盘启动（fd0）的，因此我们需要重新设置仿真器的 A：盘。点击仿真器图标 A：盘，设置如图 3 所示。

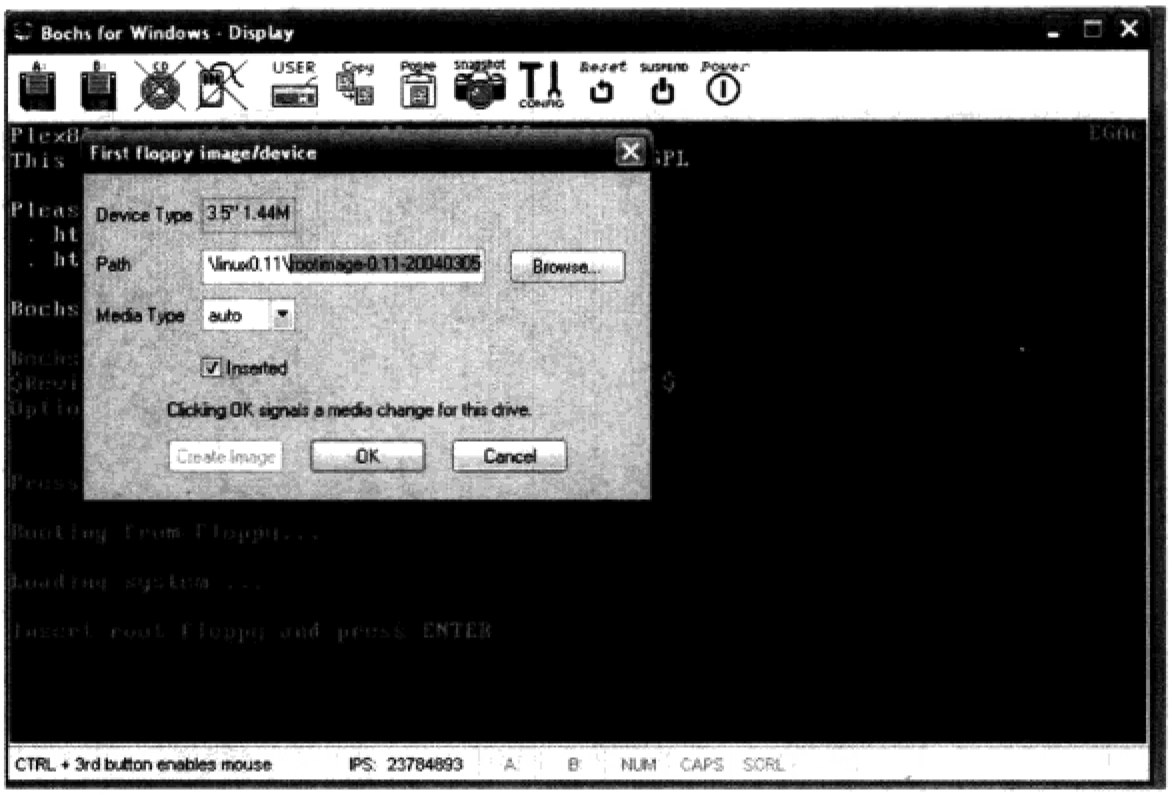


图 3 重新设置仿真器 A 盘路径

确认后，按回车键，内核继续启动。等出现 "[/usr/root]#" 时，表明 Linux 0.11 系统启动成功，如图 4 所示。

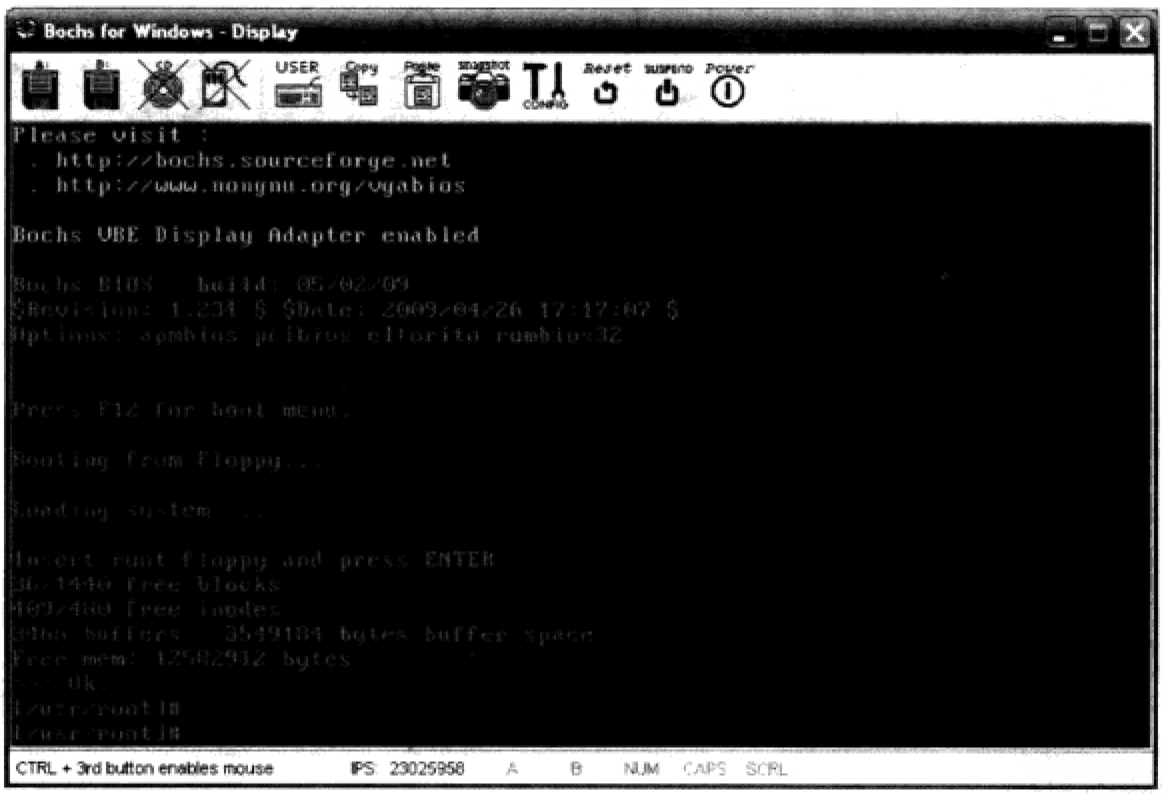


图 4 Linux 0.11 系统启动成功

完成上述准备工作后，就可以正常使用 Linux0.11 系统了。读者可以执行一些命令或运行一些程序。图 5 和图 6 展示了如何利用 Linux 0.11 执行一个程序。

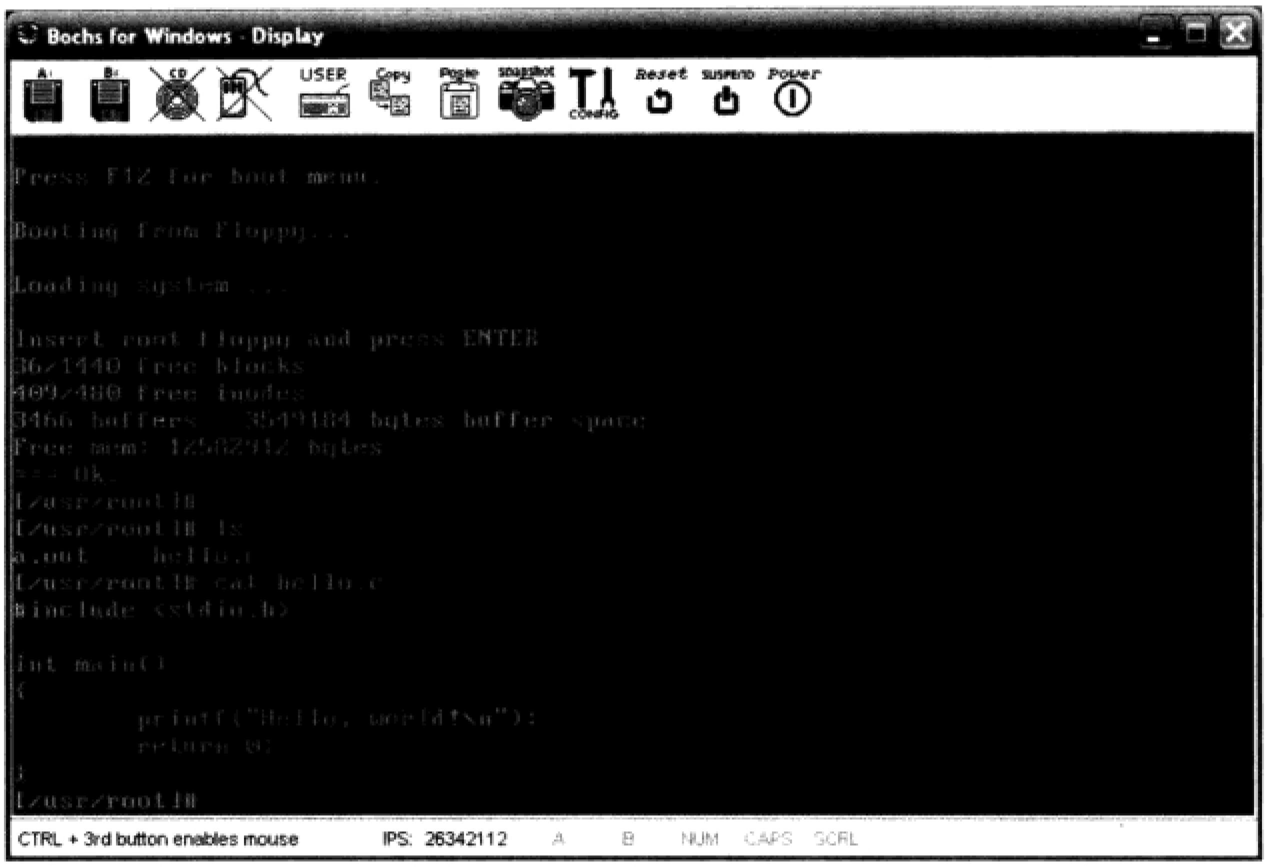


图 5 利用 cat 命令参看 hello.c 文件内容

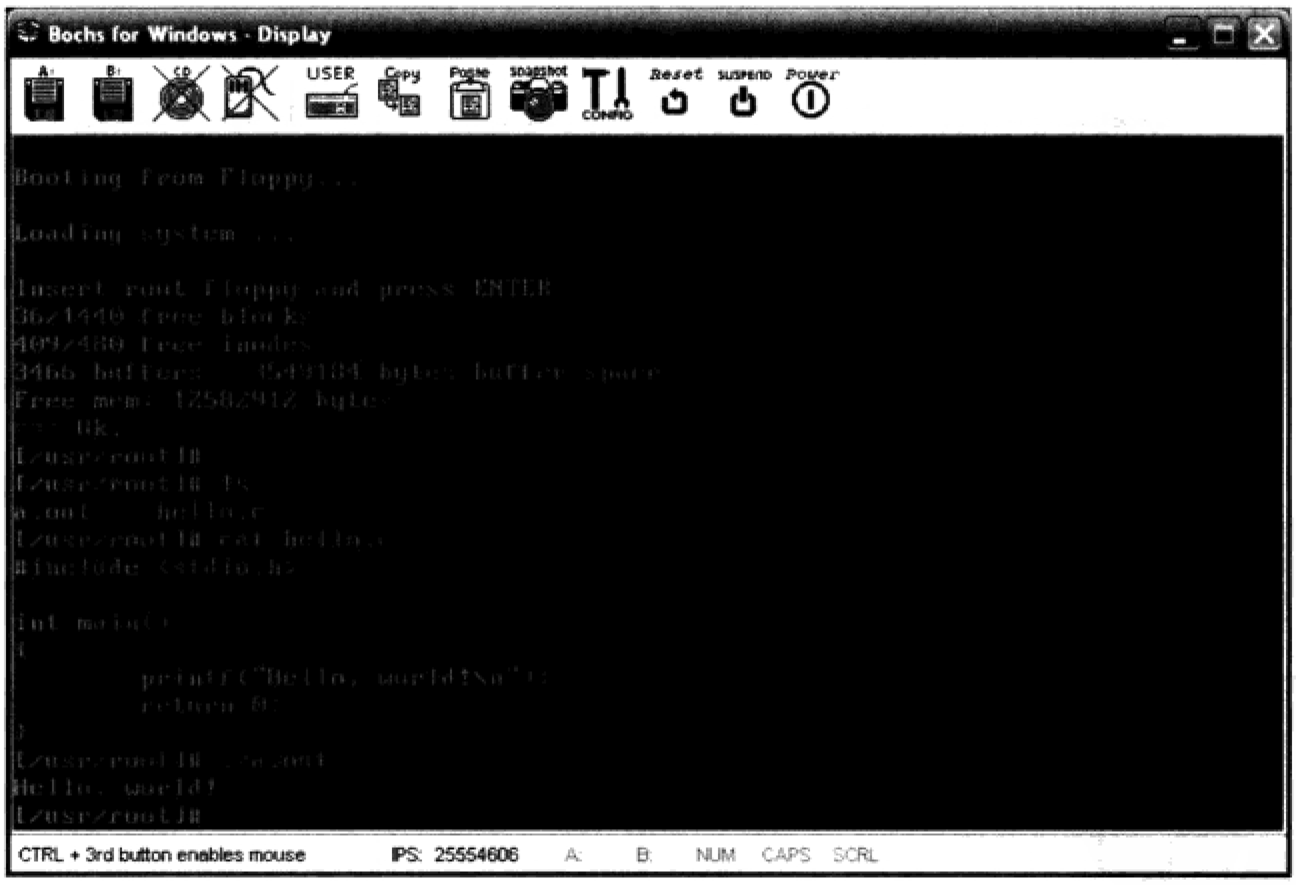


图 6 执行 a.out 程序