武汉大学计算机学院
操作系统案例分析报告

以 Linux 0.12 为例

专 业 名 称 ：软件工程

课 程 名 称 ：操作系统

指 导 教 师 ：李蓉蓉

学 生 学 号 ：2021302111204

学 生 姓 名 ：朱俊杰

二〇二三年五月

目 录

1. Linux 系统概述

1.1 Linux 系统的发展历程

Linux 操作系统是 UNIX 操作系统的一种克隆系统。它诞生于 1991 年的 10 月 5 日(这是第一次正 式向外公布的时间)。此后借助于 Internet 网络，经过全世界各地计算机爱好者的共同努力，现已成为当 今世界上使用最多的一种 UNIX 类操作系统，并且使用人数还在迅猛增长。

Linux 操作系统的诞生、发展和成长过程依赖于以下五个重要支柱:UNIX 操作系统、MINIX 操作 系统、GNU 计划、POSIX 标准和 Internet 网络。

Linux 操作系统是 UNIX 操作系统的一个克隆版本。UNIX 操作系统是美国贝尔实验室的 Ken.Thompson 和 Dennis Ritchie 于 1969 年夏在 DEC PDP-7 小型计算机上开发的一个分时操作系统。 Ken Thompson 为了能在闲置不用的 PDP-7 计算机上运行他非常喜欢的星际旅行(Space travel)游 戏，于是在 1969 年夏天乘他夫人回家乡加利福尼亚渡假期间，在一个月内开发出了 UNIX 操作系统的原 型。当时使用的是 BCPL 语言(基本组合编程语言)，后经 Dennis Ritchie 于 1972 年用移植性很强的 C 语言进行了改写，使得 UNIX 系统在大专院校得到了推广。

MINIX 系统是由 Andrew S. Tanenbaum(AST)开发的。AST 是在荷兰 Amsterdam 的 Vrije 大学数学 与计算机科学系统工作，是 ACM 和 IEEE 的资深会员(全世界也只有很少人是两会的资深会员)。共发表 了 100 多篇文章，5 本计算机书籍。MINIX 是他 1987 年编制的，主要用于学生学习操作系统原理。到 1991 年时版本是 1.5。目前主要 有两个版本在使用:1.5 版和 2.0 版。当时该操作系统在大学使用是免费的，但其他用途则不是。当然目 前 MINIX 系统已经是免费的，可以从许多 FTP 上下载。

GNU 计划和自由软件基金会 FSF(the Free Software Foundation)是由 Richard M. Stallman 于 1984 年一 手创办的。旨在开发一个类似 UNIX 并且是自由软件的完整操作系统:GNU 系统(GNU 是"GNU's Not Unix"的递归缩写，它的发音为"guh-NEW")。各种使用 Linux 作为核心的 GNU 操作系统正在被广泛的使 用。虽然这些系统通常被称作"Linux"，但是 Stallman 认为，严格地说，它们应该被称为 GNU/Linux 系 统。

POSIX(Portable Operating System Interface for Computing Systems)是由 IEEE 和 ISO/IEC 开发的一 簇标准。该标准是基于现有的 UNIX 实践和经验，描述了操作系统的调用服务接口。用于保证编制的应 用程序可以在源代码一级上在多种操作系统上移植和运行。它是在 1980 年早期一个 UNIX 用户组 (usr/group)的早期工作基础上取得的。该 UNIX 用户组原来试图将 AT&T

的 System V 操作系统和 Berkeley CSRG 的 BSD 操作系统的调用接口之间的区别重新调和集成。并于 1984 年定制出了/usr/group 标准。

1.2 Linux 系统发展的时间线

1991 年：Linus Torvalds 在芬兰赫尔辛基大学发布了 Linux 内核的最初版本，作为一个免费的操作系统内核。

1992 年：Linux 内核开始吸引开发者的关注，他们开始为内核编写驱动程序和其他功能。

1993 年：Slackware Linux 成为第一个广泛使用的 Linux 发行版之一，它通过将 Linux 内核与其他软件包和工具集合在一起，提供了一个完整的操作系统。

1994 年：Debian Linux 发布，它是一个免费的、开源的 Linux 发行版，以稳定性和广泛的软件包支持而闻名。

1996 年：Red Hat Linux 发布，它是一个商业化的 Linux 发行版，提供了商业支持和服务。

1998 年：Linux 内核的 2.2 版本发布，引入了许多重要的改进和功能。

2000 年：Ubuntu Linux 发布，它是一个用户友好的 Linux 发行版，致力于提供易于使用和易于安装的操作系统。

2003 年：Linux 内核的 2.6 版本发布，引入了许多新的特性和改进，包括更好的多处理器支持和性能优化。

2004 年：Red Hat Enterprise Linux（RHEL）发布，它是 Red Hat 针对企业市场的商业 Linux 发行版。

2011 年：Linux 内核的 3.0 版本发布，虽然版本号发生了变化，但没有引入重大的技术变革。

2011 年：Linux 基金会成立，它是一个非营利组织，致力于推广和支持 Linux 和开源软件。

2014 年：Linux 内核的 3.16 版本发布，引入了许多新的功能和改进。

2015 年：Docker 技术的出现使得容器化应用程序的部署和管理变得更加简单和高效。

2015 年：CentOS 宣布与 Red Hat 合作，以将开发重点转移到 CentOS Stream 项目上，CentOS Stream 成为 RHEL 的预发行版本。

2018 年：Linux 内核的 4.17 版本发布，引入了许多新的功能和改进，包括对 Spectre 和 Meltdown 漏洞的修复。

2019 年：Linux 内核的 5.0 版本发布，引入了许多新的功能和改进，包括对显卡和处理器的增强支持。

2021 年：Linux 内核的 5.12 版本发布，引入了许多新的功能和改进，包括对新的硬件平台的支持和性能优化。

1.3 Linux 系统的扩展应用

Linux 系统作为一个开放源代码的操作系统，具有广泛的应用领域和扩展应用。以下是 Linux 系统的一些扩展应用：

1. 服务器：Linux 在服务器领域广泛应用，包括 Web 服务器（如 Apache、Nginx）、邮件服务器（如 Postfix、Sendmail）、文件服务器（如 Samba）、数据库服务器（如 MySQL、PostgreSQL）等。Linux 的稳定性、可靠性和安全性使其成为构建服务器架构的首选。

2. 超级计算机：许多世界级超级计算机采用 Linux 作为操作系统。Linux 提供了高度的可扩展性和并行处理能力，适用于科学计算、天气预报、能源研究等领域的大规模计算需求。

3. 嵌入式系统：Linux 在嵌入式系统中得到广泛应用，例如智能手机、平板电脑、网络路由器、智能电视、汽车娱乐系统等。Linux 的开源性和可定制性使得开发者可以根据具体需求定制和优化系统。

4. 个人电脑：Linux 提供了许多用户友好的桌面环境，如 GNOME、KDE、Unity 等，以及多个发行版，如 Ubuntu、Fedora、Debian 等。这些发行版为用户提供了一个功能丰富、可定制的桌面操作系统选择。

5. 虚拟化和云计算：Linux 在虚拟化和云计算领域得到广泛应用。开源的虚拟化平台如 KVM 和 Xen 使用 Linux 作为宿主操作系统，提供了高效的虚拟机管理和资源分配。同时，Linux 还是云计算平台如 OpenStack 和 Kubernetes 的核心组件。

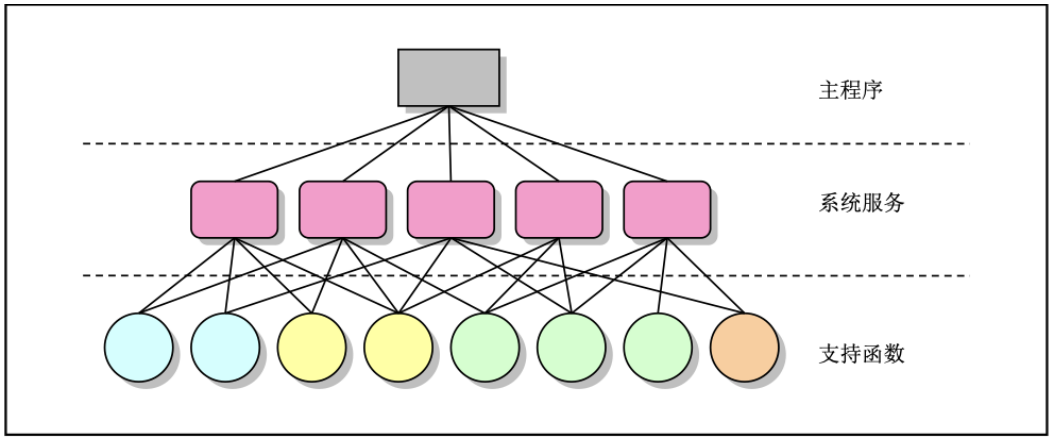
6. 安全性和网络：Linux 在网络安全领域具有重要地位。许多网络设备和防火墙采用 Linux 系统来提供网络安全功能。同时，许多安全工具和软件如 Snort、Wireshark、Metasploit 等都是基于 Linux 开发的。

7. 软件开发和编程：Linux 是许多软件开发人员和程序员的首选平台。Linux 提供了丰富的开发工具和编程环境，如 GCC 编译器、Python、Ruby、Java 等。同时，开源的开发框架和库也为开发者提供了广泛的选择和支持。

1.4 Linux0.12 的内核架构

1.4.1 Linux 内核模式

在单内核模式的系统中，操作系统所提供服务的流程为:应用主程序使用指定的参数值执行系统调 用指令(int x80)，使 CPU 从用户态(User Mode)切换到核心态(Kernel Model)，然后操作系统根据具体 的参数值调用特定的系统调用服务程序，而这些服务程序则根据需要再调用底层的一些支持函数以完成 特定的功能。在完成了应用程序所要求的服务后，操作系统又使 CPU 从核心态切换回用户态，从而返回 到应用程序中继续执行后面的指令。因此概要地讲，单内核模式的内核也可粗略地分为三个层次:调用服务的主程序层、执行系统调用的服务层和支持系统调用的底层函数。见图所示。单内核模式的主要优点是内核代码结构紧凑、执行速度快，不足之处主要是层次结构性不强。



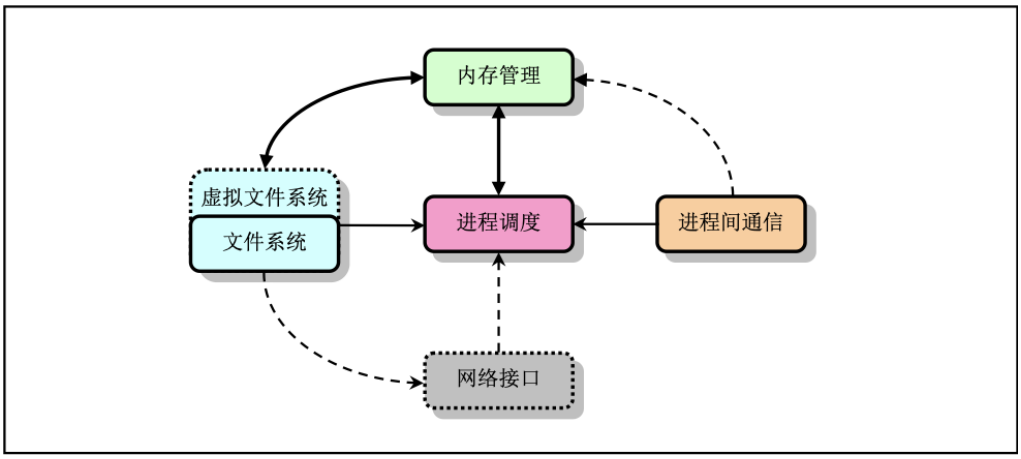
对于微内核结构模式，其主要特点是系统功能的模块化和消息传递。系统核心提供一个最基本的硬 件抽象管理层和关键系统服务功能。这些关键功能主要进程/线程间通信服务、虚拟内存管理和进程调度 等。操作系统其余功能则以各种模块化的形式在用户空间中运行。因此，微内核结构的优点是系统服务 偶合度低，便于系统的改进、扩展和移植。主要缺点则是在系统运行期间需要通过消息传递方式，在微 核心和系统各服务进程模块之间进行大量消息传递和同步操作，而这些操作会造成通信资源耗费和时间 上的延迟。典型的微内核结构的系统有 MINIX 操作系统和采用 Mach 内核的 Mac OS 系统。

1.4.2 Linux 内核系统体系结构

Linux 内核主要由 5 个模块构成，它们分别是:进程调度模块、内存管理模块、文件系统模块、进 程间通信模块和网络接口模块。

进程调度模块用来负责控制进程对 CPU 资源的使用。所采取的调度策略是各进程能够公平合理地访 问 CPU，同时保证内核能及时地执行硬件操作。内存管理模块用于确保所有进程能够安全地共享机器主 内存区，同时，内存管理模块还支持虚拟内存管理方式，使得 Linux 支持进程使用比实际内存空间更多 的内存容量。并可以利用文件系统把暂时不用的内存数据块交换到外部存储设备上去，当需要时再交换 回来。文件系统模块用于支持对外部设备的驱动和存储。虚拟文件系统模块通过向所有的外部存储设备 提供一个通用的文件接口，隐藏了各种硬件设备的不同细节。从而提供并支持与其他操作系统兼容的多 种文件系统格式。进程间通信模块子系统用于支持多种进程间的信息交换方式。网络接口模块提供对多 种网络通信标准的访问并支持许多网络硬件。

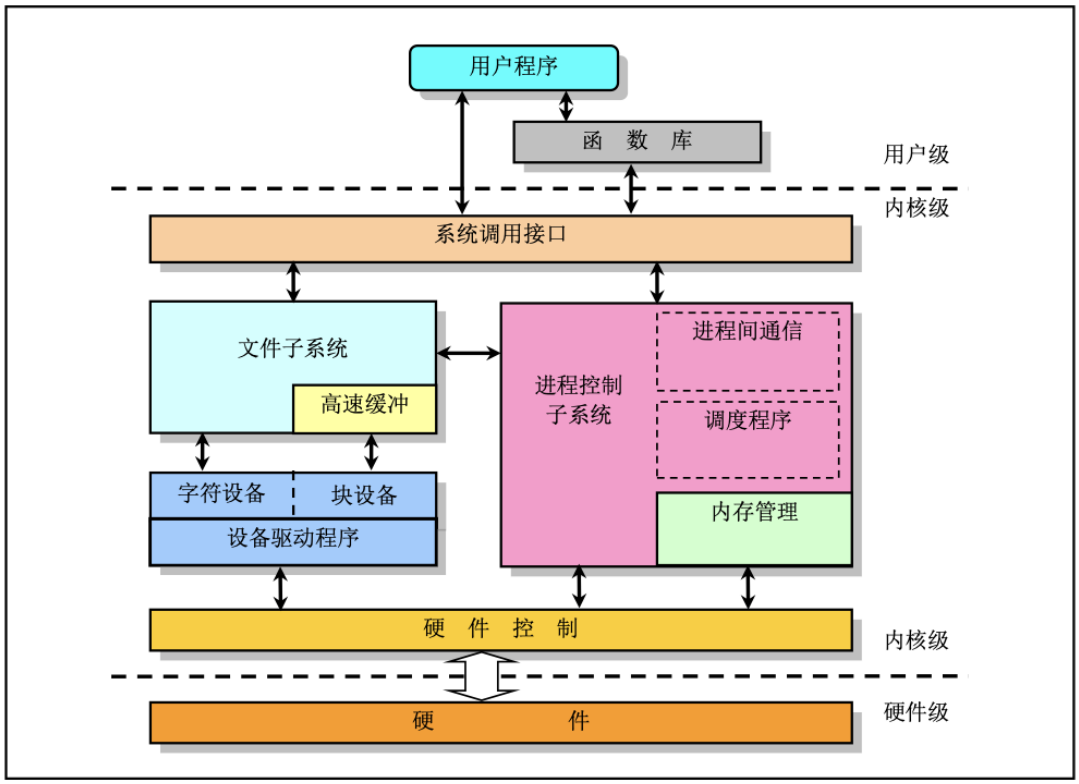
这几个模块之间的依赖关系见图 5-3 所示。其中的连线代表它们之间的依赖关系，虚线和虚框部分 表示 Linux 0.12 中还未实现的部分(从 Linux 0.95 版才开始逐步实现虚拟文件系统，而网络接口的支持 到 0.96 版才有)。



由图可以看出，所有的模块都与进程调度模块存在依赖关系。因为它们都需要依靠进程调度程序来 挂起(暂停)或重新运行它们的进程。通常，一个模块会在等待硬件操作期间被挂起，而在操作完成后 才可继续运行。例如，当一个进程试图将一数据块写到软盘上去时，软盘驱动程序就可能在启动软盘旋 转期间将该进程置为挂起等待状态，而在软盘进入到正常转速后再使得该进程能继续运行。另外 3 个模 块也是由于类似的原因而与进程调度模块存在依赖关系。

其他几个模块的依赖关系有些不太明显，但同样也很重要。进程调度子系统需要使用内存管理来调 整一特定进程所使用的物理内存空间。进程间通信子系统则需要依靠内存管理器来支持共享内存通信机 制。这种通信机制允许两个进程访问内存的同一个区域以进行进程间信息的交换。虚拟文件系统也会使 用网络接口来支持网络文件系统(NFS)，同样也能使用内存管理子系统提供内存虚拟盘(ramdisk)设 备。而内存管理子系统也会使用文件系统来支持内存数据块的交换操作。

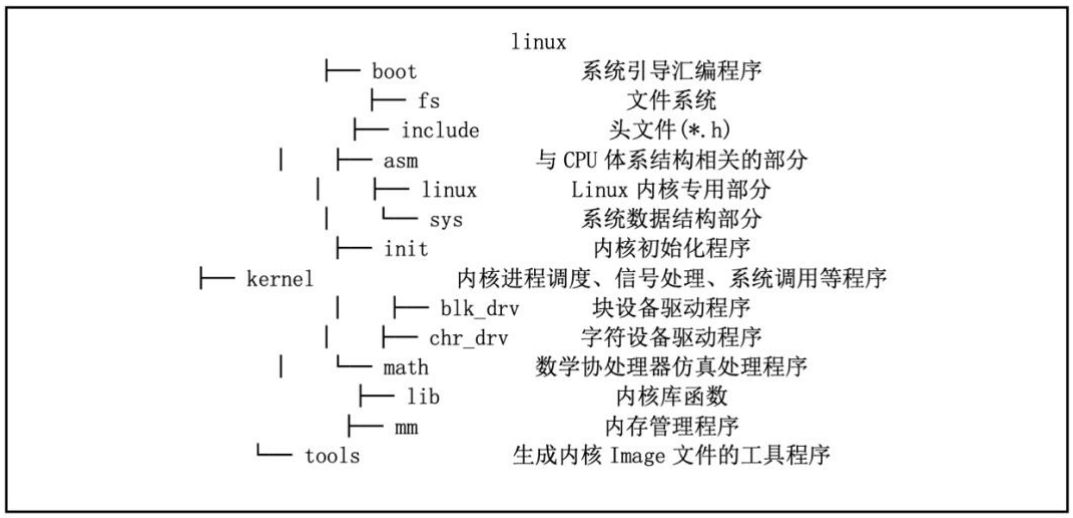
若从单内核模式结构模型出发，我们还可以根据 Linux 0.12 内核源代码的结构将内核主要模块绘制成下图所示的框图结构。



其中内核级中的几个方框，除了硬件控制方框以外，其他粗线方框分别对应内核源代码的目录组织结构。

除了这些图中已经给出的依赖关系以外，所有这些模块还会依赖于内核中的通用资源。这些资源包括内核所有子系统都会调用的内存分配和收回函数、打印警告或出错信息函数以及一些系统调试函数。

1.4.3 Linux 内核源代码目录结构



2. Linux 系统的进程和线程

2.1 Linux 进程及线程描述

在 Linux 系统中，进程和线程是并发执行的基本单位，它们扮演着重要的角色。

1. 进程（Process）：进程是运行中的程序的实例。每个进程都有自己的地址空间、内存、文件描述符和其他资源。进程是独立执行的，它们可以创建其他进程，通过进程间通信（IPC）进行相互通信和数据交换。在 Linux 中，进程由一个唯一的进程标识符（PID）来标识。

2. 线程（Thread）：线程是进程的执行单元，它是进程内的一个独立控制流。与进程相比，线程共享同一进程的地址空间和资源，包括文件描述符、信号处理器等。线程可以并发执行，允许多个线程在同一进程内同时执行不同的任务。在 Linux 中，线程也具有唯一的线程标识符（TID）。

Linux 系统中的线程和进程有以下特点和优势：

1. 轻量级：相对于进程而言，线程的创建和切换开销较小，因为线程共享相同的地址空间和资源。

2. 共享资源：线程可以共享相同的内存、文件描述符和其他进程资源，这样可以更高效地进行数据共享和通信。

3. 并发执行：多个线程可以并发执行，提高系统的响应性和效率。

4. 灵活性：线程可以在同一进程内进行通信和同步，更方便地实现协作和并发编程。

5. 编程模型：通过线程，可以更容易地实现并行计算、多任务处理和事件驱动编程。

在 Linux 中，进程和线程由操作系统内核进行管理和调度。通过系统调用和线程库（如 pthread 库），可以创建和操作进程和线程。常用的线程库包括 POSIX 线程库（pthread）、OpenMP 等。

总结起来，进程和线程是 Linux 系统中并发执行的基本单位，进程是程序的实例，拥有独立的资源，而线程是进程内的执行单元，共享相同的资源。它们在多任务处理、并行计算和协作编程等方面发挥着重要作用。

2.2 进程调度策略

调度程序：schedule()函数首先扫描任务数组。通过比较每个就绪态(TASK_RUNNING)任务的运行时间递减 滴答计数 counter 的值来确定当前哪个进程运行的时间最少。哪一个的值大，就表示运行时间还不长，于 是就选中该进程，并使用任务切换宏函数切换到该进程运行。

如果此时所有处于 TASK_RUNNING 状态进程的时间片都已经用完，系统就会根据每个进程的优先 权值 priority，对系统中所有进程(包括正在睡眠的进程)重新计算每个任务需要运行的时间片值 counter。 计算的公式是:

$$counter = \frac{counter}{2} + priority$$

这样对于正在睡眠的进程当它们被唤醒时就具有较高的时间片 counter 值。然后 schedule()函数重新扫描 任务数组中所有处于 TASK_RUNNING 状态的进程，并重复上述过程，直到选择出一个进程为止。最后 调用 switch_to()执行实际的进程切换操作。

如果此时没有其他进程可运行，系统就会选择进程 0 运行。对于 Linux 0.12 来说，进程 0 会调用 pause() 把自己置为可中断的睡眠状态并再次调用 schedule()。不过在调度进程运行时，schedule()并不在意进程 0 处于什么状态。只要系统空闲就调度进程 0 运行。

进程切换：每当选择出一个新的可运行进程时，schedule()函数就会调用定义在 include/asm/system.h 中的 switch_to()宏执行实际进程切换操作。该宏会把 CPU 的当前进程状态(上下文)替换成新进程的状态。 在进行切换之前，switch_to()首先检查要切换到的进程是否就是当前进程，如果是则什么也不做，直接 退出。否则就首先把内核全局变量 current 置为新任务的指针，然后长跳转到新任务的任务状态段 TSS 组成的地址处，造成 CPU 执行任务切换操作。此时 CPU 会把其所有寄存器的状态保存到当前任务寄存 器 TR 中 TSS 段选择符所指向的当前进程任务数据结构的 tss 结构中，然后把新任务状态段选择符所指向的新任务数据结构中 tss 结构中的寄存器信息恢复到 CPU 中，系统就正式开始运行新切换的任务了。这个过程可参见图所示：

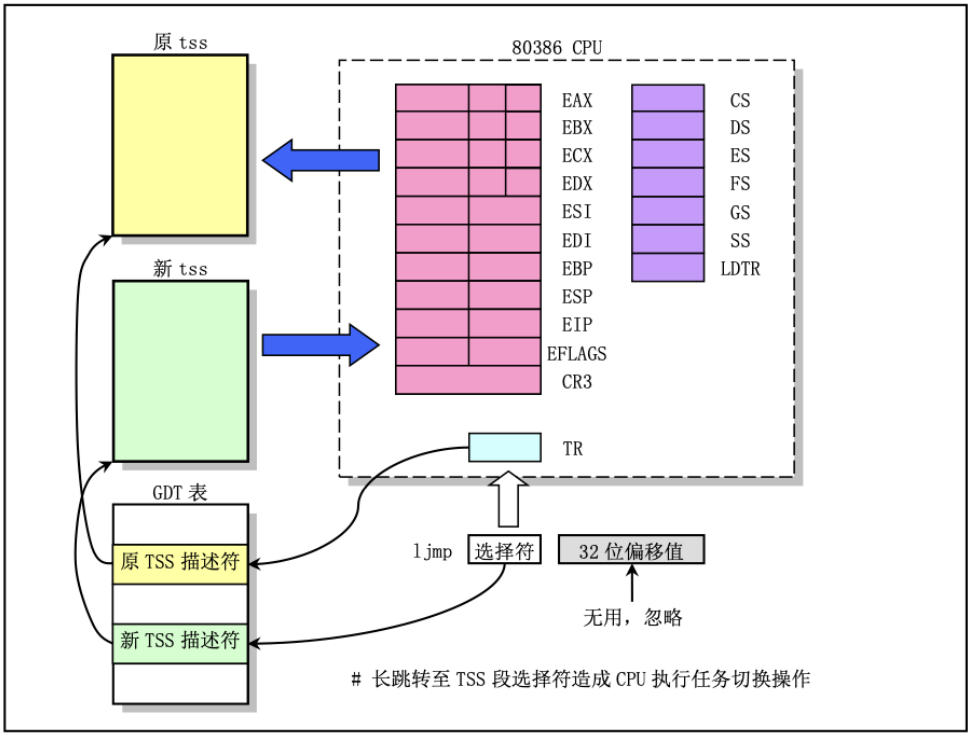


图 5-23 任务切换操作示意图

在 Linux 0.12 版本中，进程调度策略使用的是固定优先级调度（Fixed Priority Scheduling）。

具体的进程调度策略如下：

1. 每个进程都有一个静态的优先级（priority），取值范围为 0 到 39，数字越小表示优先级越高。

2. 每个进程都有一个时间片（timeslice），用于执行。时间片的长度取决于进程的优先级，优先级越高，时间片越长。

3. 当一个进程的时间片用完后，调度器会将其挂起，然后选择一个优先级最高的就绪进程来执行。如果有多个优先级相同的就绪进程，则采用轮转（round-robin）的方式，依次执行它们。

4. 进程可以通过系统调用（如 sleep()）主动让出 CPU，或者由于等待某些事件（如 I/O 操作）而被挂起。

需要注意的是，Linux 0.12 版本的进程调度策略相对简单，不具备现代操作系统中的复杂调度算法和特性。它主要侧重于按照静态优先级分配时间片，并采用轮转调度来实现多任务处理。

随着 Linux 内核的不断发展，进程调度策略得到了改进和优化。在当前的 Linux 内核版本中，使用的是完全公平调度（CFS）策略，它基于时间片和红黑树数据结构来实现更公平和高效的进程调度。

图 Linux0.12 kernel sched.c 文件中的调度函数

```c
/*
 *  'schedule()' is the scheduler function. This is GOOD CODE! There
 * probably won't be any reason to change this, as it should work well
 * in all circumstances (ie gives IO-bound processes good response etc).
 * The one thing you might take a look at is the signal-handler code here.
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;

/* check alarm, wake up any interruptible tasks that have got a signal */

    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) {
            if ((*p)->timeout && (*p)->timeout < jiffies) {
                (*p)->timeout = 0;
                if ((*p)->state == TASK_INTERRUPTIBLE)
                    (*p)->state = TASK_RUNNING;
            }
            if ((*p)->alarm && (*p)->alarm < jiffies) {
                (*p)->signal |= (1<<(SIGALRM-1));
                (*p)->alarm = 0;
            }
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
        }

/* this is the scheduler proper: */

    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);
}
```

3. linux 系统的内存管理

3.1 Linux 0.12 的内存管理技术

在 Linux 0.12 内核中，为了有效地使用机器中的物理内存，在系统初始化阶段内存被划分成几个功能区域，见图所示。

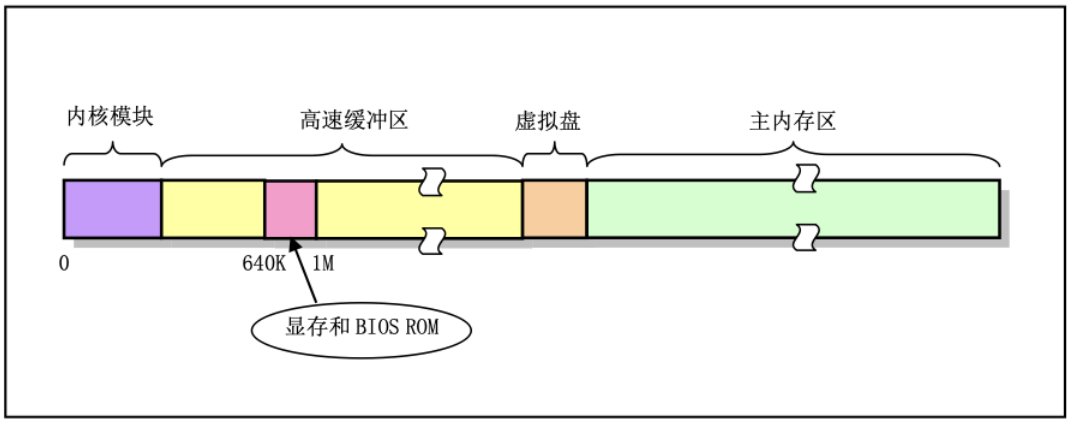


图 5-5 物理内存使用的功能分布图

其中，Linux 内核程序占据在物理内存的开始部分，接下来是供硬盘或软盘等块设备使用的高速缓 冲区部分(其中要扣除显示卡内存和 ROM BIOS 所占用的内存地址范围 640K--1MB)。当一个进程需要 读取块设备中的数据时，系统会首先把数据读到高速缓冲区中;当有数据需要写到块设备上去时，系统也是先将数据放到高速缓冲区中，然后由块设备驱动程序写到相应的设备上。内存的最后部分是供所有 程序可以随时申请和使用的主内存区。内核程序在使用主内存区时，也同样首先要向内核内存管理模块 提出申请，并在申请成功后方能使用。对于含有 RAM 虚拟盘的系统，主内存区头部还要划去一部分， 供虚拟盘存放数据。

由于计算机系统中所含的实际物理内存容量有限，因此 CPU 中通常都提供了内存管理机制对系统中的内存进行有效的管理。在 Intel 80386 及以后的 CPU 中提供了两种内存管理(地址变换)系统:内存 分段系统(Segmentation System)和分页系统(Paging System)。其中分页管理系统是可选择的，由系统程序员通过编程来确定是否采用。为了能有效地使用物理内存，Linux 系统同时采用了内存分段和分页管理机制。

Linux 0.12 内核中，在进行地址映射操作时，我们需要首先分清 3 种地址以及它们之间的变换概念: a. 程序(进程)的虚拟和逻辑地址;b. CPU 的线性地址;c. 实际物理内存地址。

虚拟地址(Virtual Address)是指由程序产生的由段选择符和段内偏移地址两个部分组成的地址。因 为这两部分组成的地址并没有直接用来访问物理内存，而是需要通过分段地址变换机制处理或映射后才 对应到物理内存地址上，因此这种地址被称为虚拟地址。虚拟地址空间由 GDT 映射的全局地址空间和 由 LDT 映射的局部地址空间组成。选择符的索引部分由 13 个比特位表示，加上区分 GDT 和 LDT 的 1 个比特位，因此 Intel 80X86 CPU 共可以索引 16384 个选择符。若每个段的长度都取最大值 4G，则最大 虚拟地址空间范围是 16384 * 4G = 64T。

逻辑地址(Logical Address)是指由程序产生的与段相关的偏移地址部分。在 Intel 保护模式下即是 指程序执行代码段限长内的偏移地址(假定代码段、数据段完全一样)。应用程序员仅需与逻辑地址打 交道，而分段和分页机制对他来说是完全透明的，仅由系统编程人员涉及。不过有些资料并不区分逻辑 地址和虚拟地址的概念，而是将它们统称为逻辑地址。

线性地址(Linear Address)是虚拟地址到物理地址变换之间的中间层，是处理器可寻址的内存空间 (称为线性地址空间)中的地址。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的 基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。

若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386 的线性地址空间容量为 4G。 物理地址(Physical Address)是指出现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址 变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

虚拟存储(或虚拟内存)(Virtual Memory)是指计算机呈现出要比实际拥有的内存大得多的内存量。因此它允许程序员编制并运行比实际系统拥有的内存大得多的程序。这使得许多大型项目也能够在 具有有限内存资源的系统上实现。一个很恰当的比喻是:你不需要很长的轨道就可以让一列火车从上海 开到北京。你只需要足够长的铁轨(比如说 3 公里)就可以完成这个任务。采取的方法是把后面的铁轨 立刻铺到火车的前面，只要你的操作足够快并能满足要求，列车就能象在一条完整的轨道上运行。这也 就是虚拟内存管理需要完成的任务。在 Linux 0.12 内核中，给每个程序(进程)都划分了总容量为 64MB 的虚拟内存空间。因此程序的逻辑地址范围是 0x0000000 到 0x4000000。

3.2 Linux 0.12 的内存管理策略

内存分段管理策略：

每个程序都可有若干个内存段组成。程序的逻辑地址(或称为虚拟地址)即是用于寻址这些段和段

中具体地址位置。在 Linux 0.12 中，程序逻辑地址到线性地址的变换过程使用了 CPU 的全局段描述符表 GDT 和局部段描述符表 LDT。由 GDT 映射的地址空间称为全局地址空间，由 LDT 映射的地址空间则称 为局部地址空间，而这两者构成了虚拟地址的空间。具体的使用方式见 5-8 图所示。

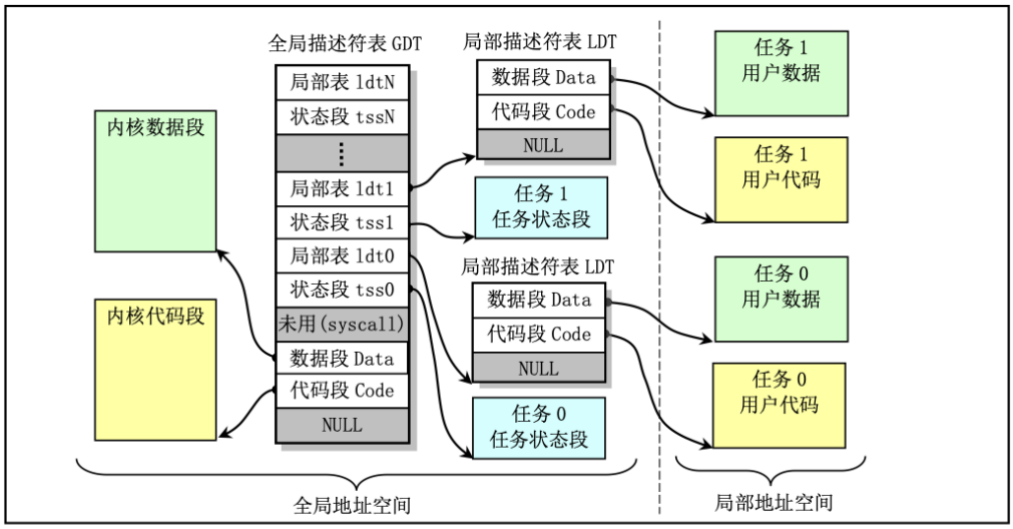


图 5-8 Linux 系统中虚拟地址空间分配图

图中画出了具有两个任务时的情况。可以看出，每个任务的局部描述符表 LDT 本身也是由 GDT 中 描述符定义的一个内存段，在该段中存放着对应任务的代码段和数据段描述符，因此 LDT 段很短，其段 限长通常只要大于 24 字节即可。同样，每个任务的任务状态段 TSS 也是由 GDT 中描述符定义的一个内 存段，其段限长也只要满足能够存放一个 TSS 数据结构就够了。

内存分页管理策略：

内存分页管理机制的基本原理是将 CPU 整个线性内存区域划分成 4096 字节为 1 页的内存页面。程 序申请使用内存时，系统就以内存页为单位进行分配。内存分页机制的实现方式与分段机制很相似，但 并不如分段机制那么完善。因为分页机制是在分段机制之上实现的，所以其结果是对系统内存具有非常 灵活的控制权，并且在分段机制的内存保护上更增加了分页保护机制。为了

在 80X86 保护模式下使用分 页机制，需要把控制寄存器 CR0 的最高比特位(位 31)置位。

在使用这种内存分页管理方法时，每个执行中的进程(任务)可以使用比实际内存容量大得多的连 续地址空间。为了在使用分页机制的条件下把线性地址映射到容量相对很小的物理内存空间上，80386 使用了页目录表和页表。页目录表项与页表项格式基本相同，都占用 4 个字节，并且每个页目录表或页 表必须只能包含 1024 个页表项。因此一个页目录表或一个页表分别共占用 1 页内存。页目录项和页表项 的小区别在于页表项有个已写位 D(Dirty)，而页目录项则没有。

线性地址到物理地址的变换过程见图 5-9 所示。图中控制寄存器 CR3 保存着是当前页目录表在物理 内存中的基地址(因此 CR3 也被称为页目录基地址寄存器 PDBR)。32 位的线性地址被分成三个部分， 分别用来在页目录表和页表中定位对应的页目录项和页表项以及在对应的物理内存页面中指定页面内的 偏移位置。因为 1 个页表可有 1024 项，因此一个页表最多可以映射 1024 * 4KB = 4MB 内存;又因为一 个页目录表最多有 1024 项，对应 1024 个二级页表，因此一个页目录表最多可以映射 1024 * 4MB = 4GB 容量的内存。即一个页目录表就可以映射整个线性地址空间范围。

由于 Linux 0.1x 系统中内核和所有任务都共用同一个页目录表，使得任何时刻处理器线性地址空间 到物理地址空间的映射函数都一样。因此为了让内核和所有任务都不互相重叠和干扰，它们都必须从虚 拟地址空间映射到线性地址空间的不同位置，即占用不同的线性地址空间范围。

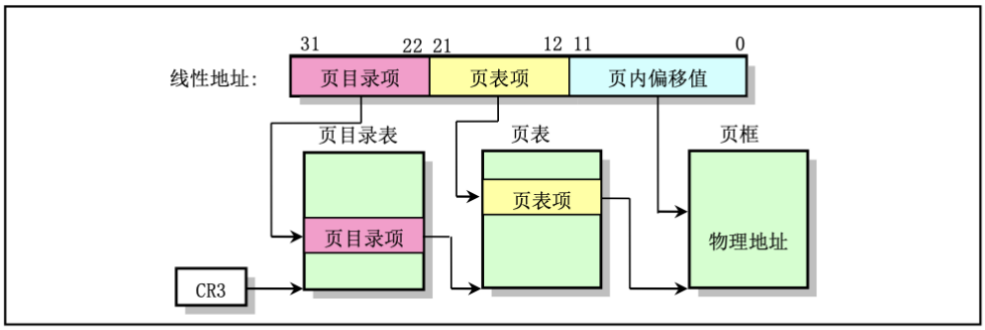


图 5-9 线性地址到物理地址的变换示意图

对于 Linux 0.12 系统，内核设置全局描述符表 GDT 中的段描述符项数最大为 256，其中 2 项空闲、 2 项系统使用，每个进程使用两项。因此，此时系统可以最多容纳(256-4)/2 =126 个任务，并且虚拟地址 范围是 ((256-4)/2)* 64MB 约等于 8G。但 0.12 内核中人工定义最大任务数 NR_TASKS = 64 个，每个任

务逻辑地址范围是 64M，并且各个任务在线性地址空间中的起始位置是 (任务号)*64MB。因此全部任务 所使用的线性地址空间范围是 64MB*64 =4G，见图 5-10 所示。图中示出了当系统具有 4 个任务时的情 况。内核代码段和数据段被映射到线性地址空间的开始 16MB 部分，并且代码和数据段都映射到同一个区域，完全互相重叠。而第 1 个任务(任务 0)是由内核“人工”启动运行的，其代码和数据包含在内 核代码和数据中，因此该任务所占用的线性地址空间范围比较特殊。任务 0 的代码段和数据段的长度是 从线性地址 0 开始的 640KB 范围，其代码和数据段也完全重叠，并且与内核代码段和数据段有重叠的部 分。实际上，Linux 0.12 中所有任务的指令空间 I(Instruction)和数据空间 D(Data)都合用一块内存， 即一个进程的所有代码、数据和堆栈部分都处于同一内存段中，也即是 I&D 不分离的一种使用方式。

任务 1 的线性地址空间范围也只有从 64MB 开始的 640KB 长度。它们之间的详细对应关系见后面说 明。任务 2 和任务 3 分别被映射线性地址 128MB 和 192MB 开始的地方，并且它们的逻辑地址范围均是 64MB。由于 4G 地址空间范围正好是 CPU 的线性地址空间范围和可寻址的最大物理地址空间范围，而 且在把任务 0 和任务 1 的逻辑地址范围看作 64MB 时，系统中同时可有任务的逻辑地址范围总和也是 4GB， 因此在 0.12 内核中比较容易混淆三种地址概念。

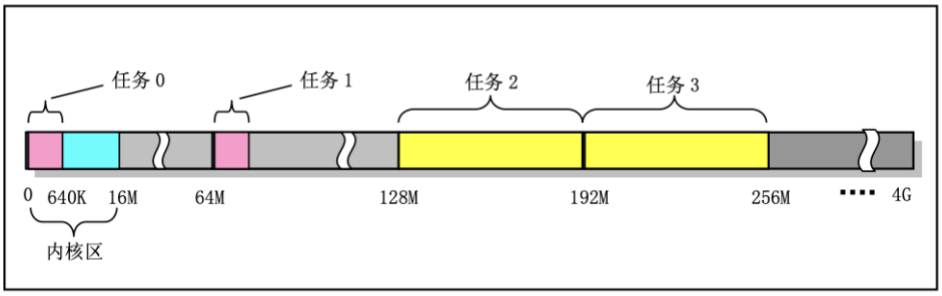


图 5-10 Linux 0.12 线性地址空间的使用示意图

如果也按照线性空间中任务的排列顺序排列虚拟空间中的任务，那么我们可以有图 5-11 所示的系统 同时可拥有所有任务在虚拟地址空间中的示意图，所占用虚拟空间范围也是 4GB。其中没有考虑内核代 码和数据在虚拟空间中所占用的范围。另外，在图中对于进程 2 和进程 3 还分别给出了各自逻辑空间中 代码段和数据段(包括数据和堆栈内容)的位置示意图。

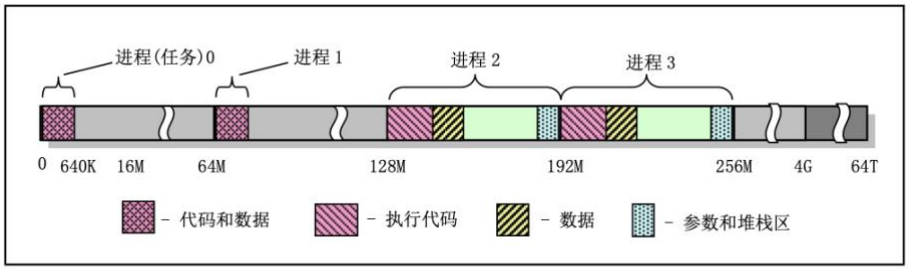
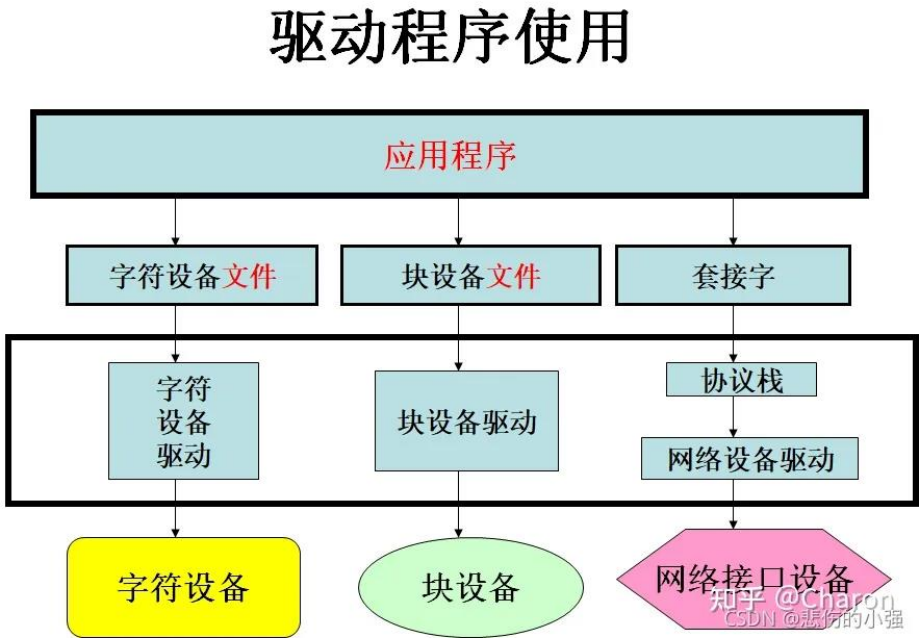


图 5-11 Linux 0.12 系统任务在虚拟空间中顺序排列所占的空间范围

4. linux 系统的设备驱动

4.1 设备驱动模型

Linux 0.12 版本的设备驱动模型可以总结如下：



字符设备驱动（Character Device Drivers）：字符设备驱动用于操作字符设备，如终端、串口等。在 Linux 0.12 中，字符设备驱动通过提供一组操作函数来对字符设备进行读写操作，这些函数由设备驱动编写者实现。

终端驱动程序用于控制终端设备，在终端设备和进程之间传输数据，并对所传输的数据进行一定的 处理。用户在键盘上键入的原始数据(Raw data)，在通过终端程序处理后，被传送给一个接收进程;而 进程向终端发送的数据，在终端程序处理后，会被显示在终端屏幕上或者通过串行线路被发送到远程终

端。根据终端程序对待输入或输出数据的方式，可以把终端工作模式分成两种。一种是规范(canonical) 模式，此时经过终端程序的数据将被进行变换处理，然后再送出。例如把 TAB 字符扩展为 8 个空格字符， 用键入的删除字符(backspace)控制删除前面键入的字符等。使用的处理函数一般称为行规则(line discipline)或行规程模块。另一种是非规范(non-canonical)模式或称原始(raw)模式。在这种模式下，行 规则程序仅在终端与进程之间传送数据，而不对数据进行规范模式的变换处理。

在终端驱动程序中，根据它们与设备的关系，以及在执行流程中的位置，可以把程序分为字符设备 的直接驱动程序和与上层直接联系的接口程序。我们可以用图 10-1 示意图来表示这种控制关系。

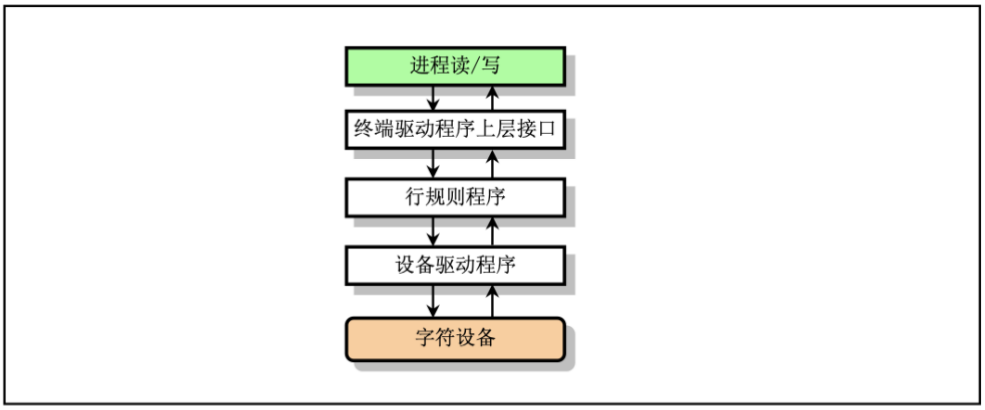


图 10-1 终端驱动程序控制流程

块设备驱动（Block Device Drivers）：块设备驱动用于操作块设备，如硬盘、闪存等。在 Linux 0.12 中，块设备驱动通常通过提供一组块操作函数来对块设备进行读写操作。

对硬盘和软盘块设备上数据的读写操作是通过中断处理程序进行的。内核每次读写的数据量以一个 逻辑块(1024 字节)为单位，而块设备控制器则是以扇区(512 字节)为单位访问块设备。在处理过程 中，内核使用了读写请求项等待队列来顺序地缓冲一次读写多个逻辑块的操作。

当一个程序需要读取硬盘上的一个逻辑块时，就会向缓冲区管理程序提出申请。而请求读写的程序 进程则进入睡眠等待状态。缓冲区管理程序首先在缓冲区中寻找以前是否已经读取过这块数据。如果缓 冲区中已经有了，就直接将对应的缓冲区块头指针返回给程序并唤醒等待的进程。若缓冲区中还不存在 所要求的数据块，则缓冲管理程序就会调用本章中的低级块读写函数 ll_rw_block()，向相应的块设备驱 动程序发出一个读数据块的操作请求。该函数会为此创建一个请求结构项，并插入请求队列中。为了提 高读写磁盘的效

率，减小磁头移动的距离，内核代码在把请求项插入请求队列时，会使用电梯算法将请 求项插入到磁头移动距离最小的请求队列位置处。

若此时对应块设备的请求项队列为空，则表明此刻该块设备不忙。于是内核就会立刻向该块设备的 控制器发出读数据命令。当块设备的控制器将数据读入到指定的缓冲块后，就会发出中断请求信号，并 调用相应的读命令后处理函数，处理继续读扇区操作或者结束本次请求项的过程。例如对相应块设备进 行关闭操作和设置该缓冲块数据已经更新标志，最后唤醒等待该块数据的进程。

网络设备驱动（Network Device Drivers）：网络设备驱动用于操作网络接口卡（NIC）和网络设备。在 Linux 0.12 中，网络设备驱动负责处理数据包的发送和接收，以及网络协议的处理。

4.2 设备驱动过程

（1）总线的启动与注册

总线是设备驱动过程中的核心架构。Linux 系统在启动阶段首先启动驱动程序，在这个过程中会调用 driver_init()方法，从而间接地调用 buses_init()方法初始化总线。由于 Linux 中对设备的管理等同于对文件的管理，所以总线初始化后会在 sysfs 目录下创建 bus 文件，bus 是一个 kset 类型的数据结构，其中包含了总线名称、总线所在的子系统、总线上所有设备集合、所有驱动集合等信息。

在总线初始化形成了一个文件后，系统使用 bus_register()函数来注册一个总线系统，注册成功后就可以在/sys/bus 目录下看到总线，之后就可以向总线中添加设备了。

（2）设备的注册

使用 device_register()用于注册设备，注册成功后可以在/sys/devices 目录下看到设备。设备添加可能失败，必须检查返回值，如若返回值是-1 则表示设备注册失败。设备文件存储了用来模拟系统的信息，device_attribute 结构体即表示了设备的属性。

（3）设备驱动的注册

设备驱动的注册过程与设备注册的过程几乎相同

（4）设备匹配

当设备注册和驱动注册都完成后，则意味着总线系统能够是被设备文件和驱动文件，这时总线便要建立设备与驱动之间的联系，因此需要一种机制实现设备与驱动程序之间的匹配。在总线的 platform_match()函数中，通过比较 pdev 指向的 platform_device 结构体的 name 属性，和 pdrv 指向的 platform_driver 结构体的 name 属性进行比较，即调用 strcmp(pdev->name, drv->name)函数来进行设备与驱动程序的识别控制。当两个 name 字段相同时，即可进行匹配操作。

5. Linux 系统的文件系统

5.1 文件系统结构

在开发 Linux 0.12 内核文件系统时，Linus 先生主要参考了当时的 MINIX 操作系统，使用了其中的 1.0 版 MINIX 文件系统。

目前 MINIX 操作系统的版本是 2.0，所使用的文件系统是 2.0 版，它与其 1.5 版系统之前的版本不同，对其容量已经作了扩展。但由于本书注释的 Linux 内核使用的是 MINIX 文件系统 1.0 版本，所以这里仅对其 1.0 版文件系统作简单介绍。

MINIX 文件系统与标准 UNIX 的文件系统基本相同，它由 6 个部分组成：①引导块；②超级块；③i 节点位图；④逻辑块位图；⑤i 节点；⑥数据块。对于一个普通的磁盘块设备来说，其各部分的分布见图 12-1 所示。

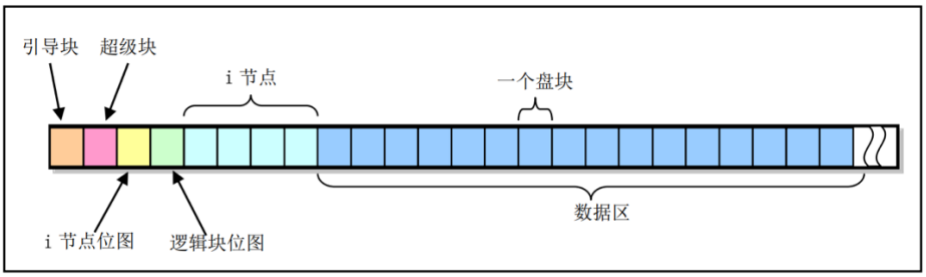


图 12-1 建有 MINIX 文件系统的块设备上各部分的布局示意图

图中，整个块设备被划分成以 1KB 为单位的磁盘块，因此对于一个 360KB 的磁盘，上图中共有 360 个磁盘块，每个方格表示一个磁盘块。在后面的说明我们会知道，在 MINIX 1.0 文件系统中，其磁盘块大小与逻辑块大小正好是一样的，也是 1KB 字节。因此 360KB 盘片也含有 360 个逻辑块。在后面的讨论中我们有时会混合使用这两个名称。

引导块是计算机加电启动时可由 ROM BIOS 自动读入的执行代码和数据盘块。但一个系统中并非所有盘设备都用于作为引导设备，所以对于不用于引导的盘片，这一盘块中可以不含代码。但任何盘块设备必须含有引导块空间，以保持 MINIX 文件系统格式的统一。即文件系统只是在块设备上空出一个存放引导块的空间。如果你把内核映像文件放在文件系统中，那么你就可以在文件系统所在设备的第 1 个块（即引导块空间）存放实际的引导程序，并由它来取得和加载文件系统中的内核映像文件。

对于容量巨大的硬盘块设备，通常会在其上会划分出几个分区，并且在每个分区中都可存放一个不同的完整文件系统，见图 12-2 所示。图中表示有 4 个分区，分别存放着 FAT32 文件系统、NTFS 文件系统、MINIX 文件系统和 EXT2 文件系统。硬盘的第一个扇区是主引导扇区，其中存放着硬盘引导程序和

分区表信息。分区表中的信息指明了硬盘上每个分区的类型、在硬盘中起始位置参数和结束位置参数以及占用的扇区总数，参见 kernel/blk_drv/hd.c 文件后的硬盘分区表结构。

图 12-2 硬盘设备上的分区和文件系统

Linux 0.12 系统采用的是 MINIX 文件系统 1.0 版。它的目录结构和目录项结构与传统 UNIX 文件系统的目录项结构相同，定义在 include/linux/fs.h 文件中。在文件系统的一个目录中，其中所有文件名的目录项存储在该目录文件的数据块中。例如，目录名 root/下的所有文件名的目录项就保存在 root/目录名文件的数据块中。而文件系统根目录下的所有文件名信息则保存在指定 i 节点（即 1 号 i 节点）的数据块中。文件名目录项结构见如下所示：

```
36  #define NAME_LEN 14
37  #define ROOT_INO 1
38


2
3   struct dir_entry {
4       unsigned short inode;
5       char name[NAME_LEN];
6   };
7
```

每个文件目录项只包括一个长度为 14 字节的文件名字符串和该文件名对应的 2 字节的 i 节点号。因 此一个逻辑磁盘块可以存放 1024/16=64 个目录项。有关文件的其它信息则被保存在该 i 节点号指定的 i 节点结构中，该结构中主要包括文件访问属性、宿主、长度、访问保存时间以及所在磁盘块等信息。每 个 i 节点号的 i 节点都位于磁盘上的固定位置处。

在打开一个文件时，文件系统会根据给定的文件名找到其 i 节点号，从而通过其对应 i 节点信息找 到文件所在的磁盘块位置，见图 12-8 所示。例如对于要查找文件名/usr/bin/vi 的 i 节点号，文件系统首先 会从具有固定 i 节点号(1)的根目录开始操作，即从 i 节点号 1 的数据块中查找到名称为 usr 的目录项， 从而得到文件/usr 的 i 节点号。根据该 i 节点号文件系统可以顺利地取得目录/usr，并在其中可以查找到 文件名 bin 的目录项。这样也就知道了/usr/bin 的 i 节点号，因而我们可以知道目录/usr/bin 的目录所在位置，并在该目录中查找到 vi 文件的目录项。最终我们可获得文件路径名

/usr/bin/vi 的 i 节点号，从而可以 从磁盘上得到该 i 节点号的 i 节点结构信息。

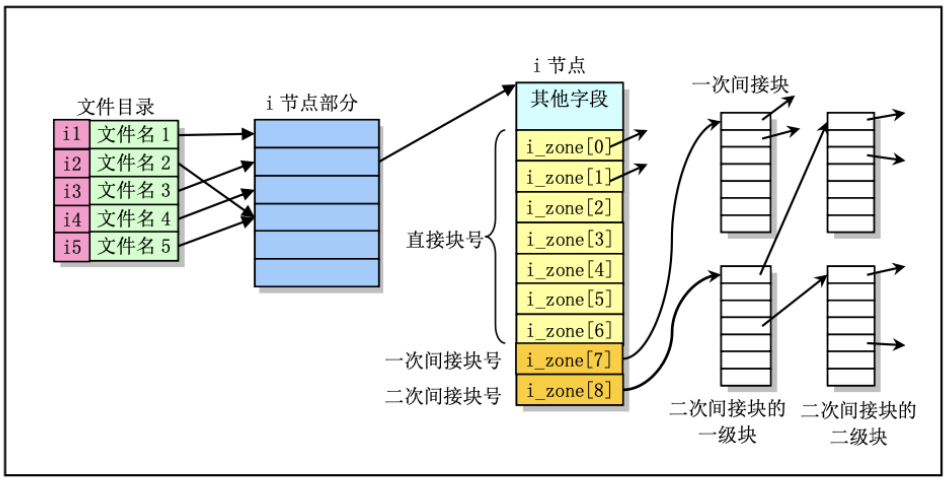


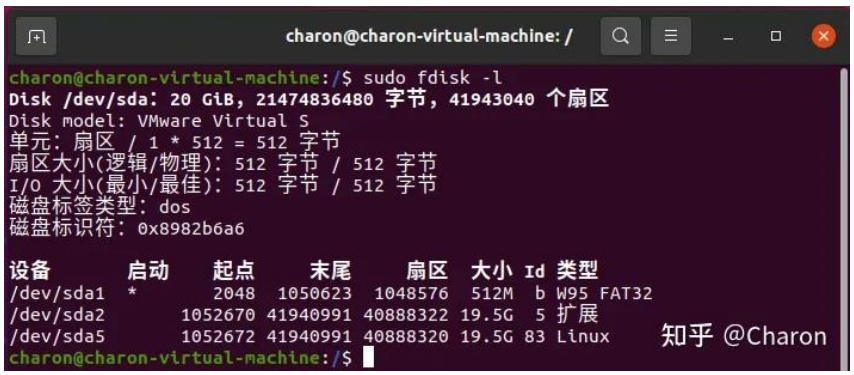
图 12-8 通过文件名最终找到对应文件磁盘块位置的示意图

5.2 磁盘空间的管理

磁盘分区管理主要包括格式化、分区、挂载三个部分的内容。

5.2.1 磁盘分区

Linux 中磁盘分区在目的上可以类比 Windows 的 C D E 等盘符的划分，不同的盼复对应的可能是一整块磁盘。磁盘分区一方面是为了易于管理和使用。我们将一块 SCSI 磁盘分为一二三四块，每块分别用作游戏、办公、软件等不同



功能，这样管理起来既方便又易于寻找。另一方面是为了数据的安全，通过分区能够降低磁盘坏道、操作错误等造成的数据损失的规模。

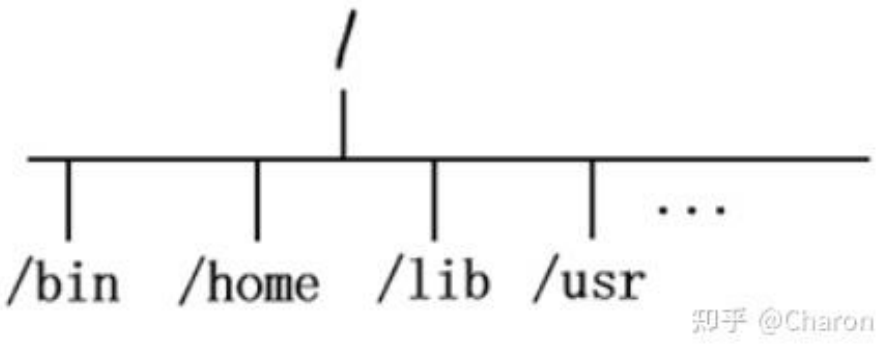
上图展示的是本人设备上 20GB 磁盘空间下 Linux 操作系统的分区情况，我们可以看到，本系统扇区大小为 512KB，其中有 64KB 存储的是分区表，一个分区占据 16KB 的存储空间。图片下方的表格描述了这块磁盘上分区的相关信息，设备号为 sda1 的分区大小为 512M，是用于引导系统启动的磁盘。Sda5 分区大小为 19.5GB，是磁盘主要的分区。Sda2 分区大小为 19.5GB，不过它是拓展分区。用户也可以使用 fdisk /dev/sda 命令管理分区，如果磁盘仍然有空余的空间，则可以继续添加分区。

5.2.2 磁盘格式化

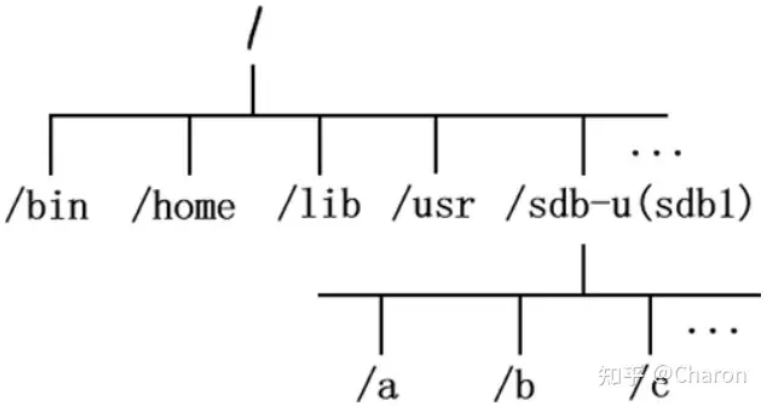
磁盘格式化指将分区格式化为不同的文件系统。我们在本节的一开始了解到了 Linux 支持多种文件系统，不同的文件系统存储组织数据存储的方式不同，作用也有差异，所以我们需要确定不同分区的文件系统类型。

在 Linux 系统中我们可以使用 mkfs 命令查看系统中的文件类型，也可以指定某个分区和文件系统类型对其进行格式化，如 mkfs.ext3 /dev/sda5 意思就是将 sda5 分区格式化为 ext3 文件系统。在完成格式化后的磁盘分区还不能立刻使用，我们还要对其实施挂载操作。

5.2.3 磁盘挂载



Linux 操作系统“一切皆文件”，所有文件都放置在以根目录为 root 的树状结构中。如图所示，任何的文件都要直接或间接连接到根目录才能拥有自己的地址，从而被操作系统找到，如果我们有一个新的设备，他有自己的文件系统，这时通过根目录是无法找到它的，如果想要通过根目录找寻这个设备下的文件，就需要进行挂载操作。



首先我们在根目录下为设备创建一个新的目录，然后通过挂载命令将设备的文件系统挂载到此目录，就得到上图所示的文件结构。

在 Linux 中我们可以使用 ll -d m*命令查看系统挂载点的情况，如上图所示。

5.3 文件操作有关的系统调用

open.c 文件用于实现与文件操作相关的系统调用。主要有文件的创建、打开和关闭，文件宿主和属 性的修改、文件访问权限的修改、文件操作时间的修改和系统文件系统 root 的变动等。

exec.c 程序实现对二进制可执行文件和 shell 脚本文件的加载与执行。其中主要的是函数 do_execve()， 它是系统中断调用(int 0x80)功能号 __NR_execve()调用的 C 处理函数，是 exec()函数簇的主要实现函数。fcntl.c 实现了文件控制操作的系统调用 fcntl()和两个文件句柄(描述符)复

制系统调用 dup()和 dup2()。

dup2()指定了新句柄的数值，而 dup()则返回当前值最小的未用句柄。句柄复制操作主要用在文件的标准 输入/输出重定向和管道操作方面。

ioctl.c 文件实现了输入/输出控制系统调用 ioctl()。主要调用 tty_ioctl()函数，对终端的 I/O 进行控制。

stat.c 文件用于实现取得文件状态信息的系统调用 stat()和 fstat()。stat()是利用文件名取信息，而 fstat() 是使用文件句柄(描述符)来取信息。

5.4 文件系统安全

DAC 自主访问控制

DAC 的核心内容是：在 Linux 中，进程理论上所拥有的权限与执行它的用户的权限相同。DAC 主要的内容包括以下几个概念：主体、客体、权限（rwx）、所有权（ugo）。

主体是用户的身份，客体是资源或者说是文件（切记：一切皆文件）。由客体的属主对自己的客体进行管理，由主体自己决定是否将自己的客体访问权限或部分访问权限授予其他主体，这种控制方式是自主的。也就是说，在自主访问控制下，用户可以按自己的意愿，有选择地与其他用户共享他的文件。

DAC 系统有两个至关重要的标准：1、文件的所有权的优先级高于访问权限，文件的所有者即便没有任何权限，也可以在为自己分配权限之后获得访问文件的能力。非文件的所有者即便已经获得了访问权限，也可能会被所有者随时收回，从而导致无权访问该文件。2、权限是文件访问的关键。无论是不是文件的所有者，关系到使用者能否访问文件的最直接的因素是其所对应的用户是否获得了可访问该文件的权限。对文件系统来说，权限包括读权限(r)，写权限(w)，执行权限(x)

MAC 强制访问权限

DAC 的机制中文件的安全很大程度上取决于使用者的个人意志，尤其对于 root 权限用户，几乎能完成任何事情，很容易出现误操作威胁到文件安全。

MAC 是利用策略将访问控制规则“强加”给访问主体的，即系统强制主体服从访问控制策略。MAC 的主要作用对象是所有主体及其所操作的客体（如：进程、文件等）。MAC 为这些主体及其所操作的客体提供安全标记，这些标记是实施强制访问控制的依据。MAC 一般与 DAC 共同使用，两种访问控制机制的过滤结果将累积，以此来达到更佳的访问控制效果。也就是说，一个主体只有通过了 DAC 限制检查与 MAC 限制检查的双重过滤装置之后，才能真正访问某个客体。

MAC 的强制访问策略为每个用户及文件分配一个安全访问等级，包括 T（最高秘密级），S（秘密级），C（机密级）和 U（无级别级），并根据主体与课题的敏感程度标记来决定访问模式，包括下读：用户级别>文件级别的读；上写：用户级别<文件级别的写；下写：用户级别>文件级别的写；上读：用户级别<文件级别的读。

以下是 Linux0.12 安全措施的主要特点：

- 文件权限：Linux 0.12 中的文件系统使用基于权限的访问控制模型。每个文件和目录都有一个所有者（owner）和一组权限位（permissions），用于控制谁可以读取、写入或执行该文件。权限位包括读取（r）、写入（w）和执行（x）的权限，分别用于文件的不同操作。
- 所有者和组：每个文件和目录都有一个所有者和一个关联的用户组。所有者是创建文件的用户，用户组则是一组相关用户的集合。文件的权限

可以独立设置给所有者、用户组和其他用户，从而实现细粒度的访问控制。

- umask：umask 是一个掩码，用于设置新创建文件和目录的默认权限。在 Linux 0.12 中，umask 的值可以通过 umask 命令进行设置。通过适当地配置 umask 值，可以限制新创建文件的默认权限，提高文件系统的安全性。
- suid、sgid 和 sticky 位：Linux 0.12 中的文件系统支持 suid、sgid 和 sticky 位。suid（Set User ID）和 sgid（Set Group ID）位可以设置在可执行文件上，以在执行时暂时改变用户或组的权限。sticky 位可以设置在目录上，防止其他用户删除其他用户的文件。
- 文件系统层面的限制：Linux 0.12 版本的文件系统没有引入一些高级的安全功能，如文件加密、访问控制列表（ACL）、强制访问控制（MAC）等。因此，在文件系统层面上，安全措施相对较为有限。

6. 收获与体会

在刚开始学习 Linux 0.12 内核源码时，我下载内核源码并且尝试直接阅读，但是发现 Linux 内核代码结构比较复杂，而个人的对于阅读大型工程项目的代码经验很少，开始阅读时寸步难行，且网上相关的资料良莠不齐，对于源码学习造成了一定的阻碍。在 oldLinux 网站上，我偶然发现了赵炯博士的《Linux 内核完全注释》，并且阅读了其中部分章节，该书对我的帮助很大。

以下是我的收获：

深入理解了操作系统：Linux 内核是操作系统的核心，研究内核代码帮助我深入了解操作系统的运作原理、内核调度、内存管理、设备驱动程序等关键概念。我了解了操作系统是如何与硬件交互、管理进程和资源，并提供各种服务和功能。

学习高级编程技术：Linux 内核代码采用了高级的编程技术和设计模式。通过研究内核代码，我可以学习到许多高级编程技巧，如内存管理、并发编程、事件驱动编程等。这些技能对于开发高性能和可靠的软件系统都非常有价值。

了解开源社区：Linux 内核是一个典型的开源项目，研究内核代码可以帮助我了解开源社区的工作方式、代码管理和协作模式。比如我发现了蜂窝科技这个开源社区，其中分享了很多 Linux 相关技术。

提高编程能力和代码质量：研究 Linux 内核代码提高了我的编程能力和代码质量。Linux 内核是一个庞大而复杂的代码库，具有高度的可维护性和可扩展性要求。通过学习内核代码，我了解到如何编写高质量的、可读性强的代码，并学会设计和实现模块化的系统。

7 参考文献

[1] 赵炯.《Linux 内核完全注释 修正版 V5.0》

[2] 知乎鱼肠. 操作系统案例分析—Ubuntu 操作系统(知乎文章)

[3] 蜂窝科技. OPPO 内核团队. CFS 任务的负载均衡(博客文章)