南 京 工 程 学 院

毕业设计说明书(论文)

作 者： 朱宇 学 号： 202150340

院 系： 计算机工程学院

专 业： 计算机科学与技术（嵌入式方向）

题 目： 基于Cortex-A9多核处理器的

操作系统内核设计和实现

指导者：  **施展 讲师**

(姓 名) (专业技术职务)

评阅者：

(姓 名) (专业技术职务)

2019年6月 南 京

**毕 业 论 文 中 文 摘 要**

|  |
| --- |
| **题目：** 基于Cortex-A9多核处理器的  操作系统内核设计和实现  摘要：  在科技日益发展的今天，新兴的技术层出不穷，我国的计算机产业也以惊人的速度发展。然而，不可否认的是在基础软件领域仍然落后于发达国家。特别是像操作系统领域，尚缺少一个认可度较高，通用性强，稳定的操作系统内核。随着新型处理器架构和物联网的兴起，迫切需要一个新的操作系统内核适应变化，这是一个重要的发展机遇。因此操作系统内核目前具有重要的研究价值。  本课题设计并实现了一个支持多任务和文件系统的操作系统内核。该系统内核以Linux内核作为参考对象，能够运行在基于Cortex-A9多核处理器的开发板上。系统内核的具体功能模块包括内存管理、进程管理（添加，删除，切换）、中断管理、设备驱动、文件系统和系统调用等。内核的性能并没有专门优化，单纯展示内核是如何设计和实现。同时内核尽可能遵循POSIX标准，方便进行应用程序的移植。  关键词：Cortex-A9　ARM　操作系统　内核　POSIX |

**毕 业 论 文 外 文 摘 要**

|  |
| --- |
| **Title** Design and Implementation of Operating System Kernel Based on Cortex-A9 Multi-core Processor  **Abstract:**  Today, with the development of science and technology, emerging technologies are emerging one after another, and China's computer industry is also developing at an alarming rate. However, it is undeniable that it still lags behind developed countries in the field of basic software. Especially in the field of operating systems, there is still a lack of a highly recognized, versatile, and stable operating system kernel. With the rise of new processor architectures and the Internet of Things, there is an urgent need for a new operating system kernel to adapt to change, which is an important development opportunity. Therefore, the operating system kernel currently has important research value.  This topic designed and implemented an operating system kernel that supports multitasking and file systems. The system kernel is referenced to the Linux kernel and runs on a Cortex-A9 multicore processor-based development board. The specific functional modules of the system kernel include memory management, process management (add, delete, switch), interrupt management, device drivers, file systems, and system calls. The performance of the kernel is not specifically optimized, simply showing how the kernel is designed and implemented. At the same time, the kernel follows the POSIX standard as much as possible, facilitating the porting of applications.  **Keywords:** Cortex-A9, ARM, operating system, kernel, POSIX |

目　　录

[前言 1](#_Toc9702314)

[第一章 绪论 2](#_Toc9702315)

[1.1 研究背景和意义 2](#_Toc9702316)

[1.2 设计内容 3](#_Toc9702317)

[第二章 硬件与软件 5](#_Toc9702318)

[2.1 硬件平台 5](#_Toc9702319)

[2.2 软件框架 7](#_Toc9702320)

[第三章 开发框架搭建 9](#_Toc9702321)

[3.1 启动镜像制作 9](#_Toc9702322)

[3.2 USB下载驱动 10](#_Toc9702323)

[3.3 启动代码 10](#_Toc9702324)

[3.4 Makefile构建框架 13](#_Toc9702325)

[3.5 ld srcipt链接脚本 15](#_Toc9702326)

[第四章 功能模块设计 17](#_Toc9702327)

[4.1 内存管理 17](#_Toc9702328)

[4.2 设备管理 20](#_Toc9702329)

[4.3 中断管理 25](#_Toc9702330)

[4.4 文件系统 28](#_Toc9702331)

[4.5 系统调用 32](#_Toc9702332)

[4.6 进程管理 38](#_Toc9702333)

[第五章 总结与展望 48](#_Toc9702334)

[5.1 总结 48](#_Toc9702335)

[5.2 展望 48](#_Toc9702336)

[参考文献 50](#_Toc9702337)

# 前言

众所周知，操作系统内核作为最重要计算机基础软件之一，设计难度不是一般的大。因此现代操作系统往往非常复杂，而初学者一下子面对完整的操作系统，想要理清代码，更是难上加难。每个操作系统都是前辈们积累了几十年的智慧结晶，动辄就是几十上百万行代码，想要学习操作系统，无疑一个精简的操作系统会事半功倍。

基于上面一些考虑，本人很早就开始考虑自己实现一个操作系统内核。最开始在x86架构的CPU上编写，相关的资料比较全面。但很快发现x86由于多年的积累，其设计已经很复杂。作为CISC处理器的代表，不能满足编写内核的初衷，那为什么不选择RISC呢？于是很快投入ARM S3C2440处理器的内核研发中。从启动代码开始，一步一步，逐渐完善，经过不断调试，不断重构，精益求精，最终形成一个比较精简、具有比较完整功能的操作系统内核。

本文所述内核在此内核基础上重新设计并移植到ARM Cortex-A9多核处理器。如此短的时间，自然无法写出能完整支持SMP的内核，只能说勉强运行单个处理器，仅希望能以此抛砖引玉，期待完整支持SMP的作品。现在的内核具有以下功能：

1. 参考kbuild系统设计一套基于Makefile的跨平台构建框架。
2. 支持VFS，支持文件读写，目录创建，重命名等操作。
3. 比较完善的设备驱动框架，支持json格式的设备树
4. 支持部分POSIX接口，容易移植

# 第一章 绪论

## 1.1 研究背景和意义

这些电子产品的底层便是一个精简的操作系统。在以前，很大程度上，这类操作系统都是直接采用国外的技术，受制于人。在近几年我国科技日益强盛，自主研发的观念深入人心。操作系统内核作为最重要计算机基础软件之一，吸引了很多学者和工程师的注意。随之涌现了一批以RT-thread为首的国产操作系统内核[4] 。

在欣慰之余，又不免有些失落，因为通用操作系统内核方面的成果还少之又少。尽管有不少号称国产通用操作系统，但多以Linux为基础进行二次开发。虽然Linux是开源的，但也能看出研究力度和层次还有待加强。

当然另一方面现在开源的操作系统内核由于时间的推移，设计过于复杂，给初学者造成很大的学习障碍。很多人试图通过重写一个内核来探究其中的奥秘。

不管是出于那方面的考虑，确实需要一个足够精简只有几万行代码的通用操作系统内核。本文所述内核正是在这样的背景下被设计出来。一方面解决自己对操作系统设计的困惑，另一方面希望能抛砖引玉，吸引更多的人投入操作系统内核研究中。

## 1.2 设计内容

本人研究了一些市面上广为流传的内核代码，同时也实际参与了一些内核项目的开发。最终结合对ARM处理器裸机编程的经验，基于Cortex-A9处理器研发一个具有多任务和文件系统的操作系统内核。具体来讲，其设计主要内容如下：

1) 内存管理，能满足各层次的内存分配请求

2) 设备管理，有相对完善的驱动框架

3) 中断管理，能支持二级中断

4) 文件系统，支持多文件系统

5) 系统调用，尽量兼容POSIX标准

6) 进程管理，拥有用户态和内核态

7) 拥有独立的构建框架，支持增量编译，项目配置，跨平台以及多架构

考虑到硬件结构的复杂性和时间限制，内核暂不支持SMP。

# 第二章 硬件与软件

## 2.1 硬件平台

本文所述内核运行在ibox4418开发板上，此开发板采用S5P4418。该芯片性能可观，能够流畅运行安卓操作系统。同时接口也很丰富，能满足正常的开发需求。

### 2.1.1 ibox4418

随着4412 芯片的即将停产，九鼎创展科技率先推出 ibox4418 卡片电脑。之所以选择S5P4418作为开发板核心芯片基于以下几点考虑：

1. 从性能上，采用更先进的工艺，一点不弱于4412，；
2. 从价格上，比4412 更便宜；
3. 从兼容性上，无需修改硬件即可兼容 6818，5430；
4. 从供货周期上，能持续批量供货，即使 4418停产，也可以直接升级 6818，即使 6818 停产，也可以直接换成 5430，拥有10年生命期。

### 2.1.2 配置和接口

图2-1 开发板结构图

1. 内核：四个ARM Cortex-A9核心；
2. 主频：1.4GHz\*4；
3. 内存：1GB DDR3，可兼容2GB DDR3；
4. Flash：支持4GB~16GB emmc，标配8GB emmc；
5. 24位RGB接口；
6. 8位LVDS接口；
7. 两路USB HOST接口，支持更多的USB设备同时使用；
8. USB OTG接口；
9. 四路TTL电平UART接口；
10. 2路TF卡接口；
11. 两路LED指示；
12. 复位按钮；
13. 软件开关机按钮；
14. 支持外置扬声器；
15. 支持MIC输入；
16. 支持耳机输出接口；
17. 独家支持免启动配置开关设计；
18. 支持背光无级调节；
19. 支持HDMI接口；
20. 支持5点电容触摸；
21. 板载USB接口WIFI/BT二合一模块；
22. 支持多种SPI，I2C，UART，等外围器件扩展；
23. 支持MPEG4，H.263，H.264，MJPEG视频编码；
24. 支持几乎全格式视频解码；
25. 支持2D，3D高性能图形加速；
26. 支持RTC时钟实时保存；
27. 支持千兆有线以太网RTL8211E；
28. 支持BT656/BT601/MIPI摄相头接口；
29. 支持GPS接口；
30. 支持GPRS接口；
31. 支持外置USB 3G模块；
32. 支持USB鼠标，键盘；
33. 支持红外一体化接收头；

## 2.2 软件框架

### 2.2.1 xboot驱动框架

XBOOT是一个代码结构规整，代码抽象度很高的嵌入式裸机框架，支持各类主流的CPU架构，同时支持多款SoC。应用程序采用Lua虚拟机执行，功能强大，支持GUI开发。

本项目主要使用了XBOOT的驱动框架部分。原因有三点：

1. 支持s5p4418处理器，包含了常用驱动；
2. 代码抽象度高，接口简单，代码清晰，方便移植；
3. 支持设备树。通过修改json文件即可完成设备驱动的配置，方便快捷。

### 2.2.2 MINE内核

MINE内核由田宇编写，并写入《一个64位操作系统的设计与实现》一书中。本文所述内核早期源码便是从MINE内核移植过来。随着版本的变迁，MINE内核的源码已发生很多的改动，并且本课题也添加和删除了一些功能。不过不可否认，MINE内核给了本课题灵感和设计参考。MINE内核运行与x64架构CPU，并且支持SMP。其代码主要参考了Linux内核，总行数只有一万行，适合操作系统入门。

# 第三章 开发框架搭建

## 3.1 启动镜像制作

S5P4418处理器的启动过程有四步，如下图3-1：

系统上电

IROM

2ndboot

uboot.bin

图3-1 S5P4418处理器过程

在IROM代码中，会检测当前设备支持的启动方式如emmc、sdcard等。之后会按优先级从相应的设备运行2ndboot。在2ndboot部分，用户可以做更详细的启动配置。最后2ndboot启动用户定义的启动文件，也可以理解为3rdboot。整个过程就像火箭发射，一级推动一级。

下面是一个镜像在EMMC中分布图：

2nboot

uboot.bin

**0 1 2 ... 62 63 64 …**

图3-2 镜像结构图

绿块包含了sd/emmc卡的信息，紫块是Nsih.bin，包含了启动配置信息，如DDR的频率，CPU的频率，启动方式等。

本课题并不想依赖uboot启动，而是从2ndboot直接启动，即第四步换成自己的kernel.bin。因此本课题需要修改2ndboot源码。

2ndboot源码由三星官方提供，用户可更具需要修改。在下载代码后，配置为usb启动，并编译生成2ndboot文件。然后使用九鼎创展提供的工具生成启动镜像，命令如下：

|  |
| --- |
| mk4418 usbboot.bin nsih.txt 2ndboot 2ndboot |

工具名为mk4418，nsih.txt为系统配置文件。最后一个参数是3rdboot，因为本课题选择usb启动，暂时不指定3rdboot，随便写一个文件就行。

最后插上读卡器，假设/dev/sdb是对应的sdcard设备。使用dd命令将usbboot.bin从Block1开始烧写，命令如下：

|  |
| --- |
| dd if=usbboot.bin of=/dev/sdb bs=512 seek=1 |

这样就完成了启动镜像的制作。

## 3.2 USB下载驱动

上一节配置2ndboot为usb启动，因此需要通过usb将内核二进制文件传给开发板，这样就免不了要编写usb下载驱动。

方便起见，本课题使用libusb在用户态编写驱动。参考libusb的批量传输示例代码，定义厂家和设备编码如下：

|  |
| --- |
| #define X4418\_SECBULK\_IDVENDOR 0x04E8  #define X4418\_SECBULK\_IDPRODUCT 0x1234 |

这样就能识别开发板了，之后添加参数配置和文件读取的代码，即可完成驱动的编写。

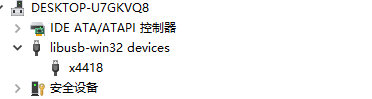
最后插上开发板usb线，按照libusb向导配置windows驱动并安装，最终效果如下：

图3-3 驱动加载图

## 3.3 启动代码

到这里，本课题已经能正常下载和启动自己的程序了，但是C语言代码不能直接运行，需要一小段汇编代码启动代码加载。按照道理应该把启动代码放在代码分析部分，但启动代码并不算具体哪个模块，因此独立出来，单独分析。主要包括异常向量表的设置，CPU初始化，各模式栈初始化等，具体代码如下：

异常向量表：

|  |
| --- |
| .text  .arm  .global \_start  \_start:  b reset  ldr pc, \_vector\_und  ldr pc, \_vector\_swi  ldr pc, \_vector\_pabt  ldr pc, \_vector\_dabt  ldr pc, \_vector\_addrexcptn  ldr pc, \_vector\_irq  ldr pc, \_vector\_fiq  \_vector\_und:  .word vector\_und  \_vector\_swi:  .word vector\_swi  \_vector\_pabt:  .word vector\_pabt  \_vector\_dabt:  .word vector\_dabt  \_vector\_addrexcptn:  .word vector\_addrexcptn  \_vector\_irq:  .word vector\_irq  \_vector\_fiq:  .word vector\_fiq |

系统重置代码：

|  |
| --- |
| reset: |

设置CPU为SVC32模式，并禁用中断：

|  |
| --- |
| mrs r0, cpsr  bic r0, r0, #0x1f  orr r0, r0, #0xd3  msr cpsr, r0 |

使能neon/vfp单元：

|  |
| --- |
| mrc p15, #0, r1, c1, c0, #2  orr r1, r1, #(0xf << 20)  mcr p15, #0, r1, c1, c0, #2  mov r1, #0  mcr p15, #0, r1, c7, c5, #4  mov r0, #0x40000000  fmxr fpexc, r0 |

初始化cache：

|  |
| --- |
| mrc p15, 0, r0, c0, c0, 0  and r1, r0, #0x00f00000  and r2, r0, #0x0000000f  orr r2, r2, r1, lsr #20-4  cmp r2, #0x30  mrceq p15, 0, r0, c1, c0, 1  orreq r0, r0, #0x6  mcreq p15, 0, r0, c1, c0, 1 |

失效数据cache和指令cache：

|  |
| --- |
| mov r0, #0  mcr p15, 0, r0, c8, c7, 0  mcr p15, 0, r0, c7, c5, 0 |

禁用MMU和cache：

|  |
| --- |
| mrc p15, 0, r0, c1, c0, 0  bic r0, r0, #0x00002000  bic r0, r0, #0x00000007  orr r0, r0, #0x00000002  orr r0, r0, #0x00000800  mcr p15, 0, r0, c1, c0, 0 |

设置异常向量表基地址寄存器：

|  |
| --- |
| ldr r0, =\_start  mcr p15, 0, r0, c12, c0, 0  mrc p15, 0, r0, c1, c0, 0  bic r0, #(1<<13)  mcr p15, 0, r0, c1, c0, 0 |

初始化各模式栈：

|  |
| --- |
| mrs r0, cpsr  bic r0, r0, #0x1f  orr r1, r0, #0x1b  msr cpsr\_cxsf, r1  ldr sp, \_stack\_und\_start  bic r0, r0, #0x1f  orr r1, r0, #0x17  msr cpsr\_cxsf, r1  ldr sp, \_stack\_abt\_start  bic r0, r0, #0x1f  orr r1, r0, #0x12  msr cpsr\_cxsf, r1  ldr sp, \_stack\_irq\_start  bic r0, r0, #0x1f  orr r1, r0, #0x11  msr cpsr\_cxsf, r1  ldr sp, \_stack\_fiq\_start  bic r0, r0, #0x1f  orr r1, r0, #0x13  msr cpsr\_cxsf, r1  ldr sp, \_stack\_srv\_end |

复制代码到链接地址：

|  |
| --- |
| adr r0, \_start  ldr r1, =\_start  cmp r0, r1  beq 1f  ldr r0, \_image\_start  adr r1, \_start  ldr r2, \_image\_end  sub r2, r2, r0  bl memcpy  1: nop |

清除bss段：

|  |
| --- |
| ldr r0, \_bss\_start  ldr r2, \_bss\_end  sub r2, r2, r0  mov r1, #0  bl memset |

调用main函数：

|  |
| --- |
| ldr r1, =\_main  mov pc, r1  \_main:  mov r0, #0  mov r1, #0  mov fp, #0  bl main  b \_main |

## 3.4 Makefile构建框架

此框架由本人参考Linux的KBuild系统简化而来，支持增量编译和项目配置。

### 3.4.1 名词约定

1. 用户Makefile:指的是一般用户就可以修改的Makefile,主要用来指定参与编译的文件。
2. 系统Makefile:指的是对体系架构和内核有深入了解的开发人员才可以修改的Makfile文件。
3. 核心Makefile:指的是对编译系统有深入了解的开发人员才可以修改的Makfile文件。
4. 源码目录：也可以叫输入目录。代表当前项目的源码根目录。
5. 输出目录：你可以将生成的文件导出到输出目录，来避免污染源码。默认和源码目录一致。
6. 编译目录：当前正在编译文件的目录。
7. 工作目录：工作目录始终在输出目录的根目录。如果你没有指定输出目录，则和源码目录一致。

### 3.4.2 总体框架

1. 顶层Makefile：它是整个工程的入口，从总体上控制着项目的编译和链接过程。
2. 项目配置文件：包括arch/$(ARCH)/$(MACH)/include/config.h和include/kconfigs.h，后者是所有平台通用的配置，前者与具体运行环境相关。编译系统会根据这两个文件自动在include/config生成autoconf.h和auto.conf文件。前者用于C语言的配置，后者用于Makefile的配置。
3. 用户Makefile：基本每个源码目录下都会存在，分散在项目中。
4. 核心Makefile：
5. scripts/Makefile.\* : Makefile共用的通用规则、脚本等
6. scripts/Makefile.build:用于递归的编译项目
7. scripts/Makefile.include:一些通用的变量和函数
8. scripts/fixdep.c:修复gcc生成的依赖文件
9. arch/$(ARCH)/$(MACH)/Makfile.mach：用于设置机器相关的工具链参数
10. arch/$(ARCH)/$(MACH)/Makfile.arch：用于设置架构相关的工具链参数

### 3.4.3 使用方法

1. 编译

|  |
| --- |
| make |

1. 清除

|  |
| --- |
| make clean |

1. 导出生成目录

|  |
| --- |
| make O=output #编译  make clean O=output #清除 |

Makefile会自动创建输出目录，所以你应该保证目录创建成功。使用这个功能请确保指定了一个空目录，否则可能在清除时，删除一些重要文件。

1. 添加编译目录或文件。

假设添加init/yy目录，你应该找到init目录下的Makefile（如果没有请创建）添加如下内容：

|  |
| --- |
| obj-y += yy/ |

这里yy目录下的所有.S和.c文件都会参与编译。值得注意的是yy目录的子目录并不会参与编译，防止将一些错误的文件的编译进内核。

1. 添加编译文件。

这时假设添加init/xx.c文件，你应该在到init目录下的Makefile，添加如下内容：

|  |
| --- |
| obj-y += xx.c |

## 3.5 ld srcipt链接脚本

链接脚本主要有四个部分组成。通过一系列的定义描述，控制最终生成的可执行文件布局。

### 3.5.1 总体描述

1. 定义输出格式

|  |
| --- |
| OUTPUT\_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm") |

1. 定义体系架构

|  |
| --- |
| OUTPUT\_ARCH(arm) |

1. 设置入口函数

|  |
| --- |
| ENTRY(\_start) |

### 3.5.2 常量定义

主要是各个模式栈大小定义：

|  |
| --- |
| STACK\_UND\_SIZE = 0x12;  STACK\_ABT\_SIZE = 0x12;  STACK\_IRQ\_SIZE = 0x12;  STACK\_FIQ\_SIZE = 0x12;  STACK\_SRV\_SIZE = 0x100000; |

### 3.5.3 内存区域划分

主要分为RAM、DMA和HEAP区域，其位置和大小如下：

|  |
| --- |
| ram: org = 0x40100000, len = 128M - 1M  dma: org = 0x48000000, len = 128M  heap: org = 0x50000000, len = 256M |

### 3.5.4 段划分

这里定义了最终可执行文件有哪些段，这些段由什么组成。由于代码冗长，择要说明。

.text包含了所有的可执行代码，描述如下：

|  |
| --- |
| .text :  {  PROVIDE(\_\_image\_start = .);  PROVIDE(\_\_text\_start = .);  \_text = .;  arch/arm32/mach-x4418/head.o;  \*(.text\*);  \*(.init.text);  \*(.exit.text);  PROVIDE(\_\_text\_end = .);  \_etext = .;  } > ram |

.initcall包含了所有的初始化函数，系统关键的初始化程序都是通过这个段调用的，总共有10个等级，描述如下：

|  |
| --- |
| .initcall ALIGN(8) :  {  PROVIDE(\_\_initcall\_start = .);  KEEP(\*(.initcall\_0.text))  KEEP(\*(.initcall\_1.text))  KEEP(\*(.initcall\_2.text))  KEEP(\*(.initcall\_3.text))  KEEP(\*(.initcall\_4.text))  KEEP(\*(.initcall\_5.text))  KEEP(\*(.initcall\_6.text))  KEEP(\*(.initcall\_7.text))  KEEP(\*(.initcall\_8.text))  KEEP(\*(.initcall\_9.text))  PROVIDE(\_\_initcall\_end = .);  } > ram |

以上便是链接脚本的主要内容。

# 第四章 功能模块设计

## 4.1 内存管理

内存管理单元是操作系统内核最最基础的模块，一直以来是重要的研究对象。任何程序想必最重要的资源必定是内存，而系统关键的数据结构都依赖于内存分配。为此，科学家曾提出很多不尽相同的算法，这些算法优势也各不相同。

内存管理主要负责内存分配和MMU设置。内存分配能够让内核更好的管理内存空间。MMU则可以让不同的程序共享同一段内存地址空间，即管理地址空间。本节主要内容侧重内存分配。算法设计上，主要借鉴了Linux内核的SLAB分配器，能够有效减少内存的碎片化。当然，在API设计上也尽量和Linux保持一致。

### 4.1.1 SLAB内存池

虽然SLAB是建立在更基础的分配算法上，但由于其相对简单易懂，因此放在最前面讲解。正常编写程序申请内存时，其实会申请很多一样大小的数据，如果用通用的分配算法申请，难免会出浪费的情况。举个例子，当你申请30B的数据时，可能获得的空间是32B。并且，采用通用算法，算法会更复杂，操作更耗时。Linux为了处理这类问题，就采用了SLAB内存池的技术。通俗来讲，就是提前申请一大块空间，然后划分出许多大小一致的内存单元。当有程序申请内存，直接找一块空闲的单元就行了。

下面便看一下Slab\_cache的结构体定义：

|  |
| --- |
| struct Slab\_cache {  unsigned long size; //分配单元大小  unsigned long total\_using;//已分配个数  unsigned long total\_free;//未分配个数  struct Slab \* cache\_pool;//slab链表  struct Slab \* cache\_dma\_pool;  void \*(\* constructor)(void \* Vaddress, unsigned long arg);//构造函数  void \*(\* destructor)(void \* Vaddress, unsigned long arg);//析构函数  }; |

这里反映了一个slab总体信息，用户可以自定义构造函数和析构函数。cache\_pool指向具体的内存池链表，如果内存池用光，就需要将申请新的内存池加入链表。具体定义如下：

|  |
| --- |
| struct Slab {  struct list\_head list; //链表节点  struct Page \* page; //内存池所在Page  unsigned long using\_count; //使用数  unsigned long free\_count; //空闲数  void \* Vaddress; //Page对应的地址  unsigned long color\_length;// color\_map的bit数  unsigned long color\_count; // color\_map的字节数  unsigned long \* color\_map; // color\_map的地址  }; |

color\_map是一个动态分配的数组，包含了内存池的分配情况。其中的每一个bit代表一个内存单元。

下面介绍一下slab的几个API。

1. slab创建：

|  |
| --- |
| struct Slab\_cache \* slab\_create(  unsigned long size, //分配单元大小  void \* (\* constructor)(void \* Vaddress, unsigned long arg),  void \* (\* destructor)(void \* Vaddress, unsigned long arg),  unsigned long arg  ); |

1. slab销毁：

|  |
| --- |
| unsigned long slab\_destroy(struct Slab\_cache \* slab\_cache); |

1. slab分配对象：

|  |
| --- |
| void \* slab\_malloc(  struct Slab\_cache \* slab\_cache,  unsigned long arg  ); |

1. slab释放对象：

|  |
| --- |
| unsigned long slab\_free(  struct Slab\_cache \* slab\_cache,  void \* address,  unsigned long arg  ); |

### 4.1.2 页管理单元

这一小节介绍内核最底层的分配算法，不管是SLAB内存池还是通用内存分配模块，其根本都是调用页管理单元。

本文内核的页都是2MB大小，Linux的页分配最小则是4KB。在粒度上相对比较大，但也因此碎片化的问题会少很多。Linux的采用了伙伴系统算法，能有效抑制碎片化，而本课题为了简单起见，单纯采用最先适配算法。下面介绍内存大致的区域划分。

图4-1 内存区域划分图

从左往右依次是.text段、.data段、page bit map、page stuct、zone struct、page和异常向量表。

内存首先会被划分为不同的区域即zone，内核定义了ZONE\_DMA、ZONE\_NORMAL和ZONE\_UNMAPED。每个zone包含若干page struct，每个page struct对应一个page。

主要的API如下：

1. 初始化内存

|  |
| --- |
| void init\_memory(); |

1. 页分配

|  |
| --- |
| struct Page \* alloc\_pages(  int zone\_select,  int number,  unsigned long page\_flags  ); |

zone\_select用于选定在哪个zone分配页，像LCD控制器申请framebuffer时就要指定ZONE\_DMA。Number代表要申请的page数，page\_flags暂时没有用到，写零即可。

1. 页释放

|  |
| --- |
| void free\_pages(struct Page \* page, int number); |

### 4.1.3 通用内存管理单元

为了处理程序任意的内存申请请求，必然需要一个比较普适的分配算法。这里的算法其实和slab一致，就连结构体用的都是同一个。但是这里的slab是预先定义好的，并且和slab的具体实现略有不同。Slab事实上是依赖通用内存管理单元的，因为color\_map大小肯定是不固定的。总之，它们都是使用内存池技术，却是两套不同的实现，通用内存管理单元的实现更偏底层一点罢了。

既然是通过slab实现的，那先看看是怎么定义slab\_cache的。代码如下：

|  |
| --- |
| struct Slab\_cache kmalloc\_cache\_size[16] = {  {32 , 0 , 0 , NULL , NULL , NULL , NULL},  {64 , 0 , 0 , NULL , NULL , NULL , NULL},  {128 , 0 , 0 , NULL , NULL , NULL , NULL},  {256 , 0 , 0 , NULL , NULL , NULL , NULL},  {512 , 0 , 0 , NULL , NULL , NULL , NULL},  {1024 , 0 , 0 , NULL , NULL , NULL , NULL}, //1KB  {2048 , 0 , 0 , NULL , NULL , NULL , NULL},  {4096 , 0 , 0 , NULL , NULL , NULL , NULL}, //4KB  {8192 , 0 , 0 , NULL , NULL , NULL , NULL},  {16384 , 0 , 0 , NULL , NULL , NULL , NULL},  {32768 , 0 , 0 , NULL , NULL , NULL , NULL},  {65536 , 0 , 0 , NULL , NULL , NULL , NULL}, //64KB  {131072 , 0 , 0 , NULL , NULL , NULL , NULL}, //128KB  {262144 , 0 , 0 , NULL , NULL , NULL , NULL},  {524288 , 0 , 0 , NULL , NULL , NULL , NULL},  {1048576, 0 , 0 , NULL , NULL , NULL , NULL}, //1MB  }; |

可以发现一共定义了16的内存池，从32B到1MB大小不等。当程序申请一个500B的内存空间，它会匹配到最接近的512B的内存池，以此类推，也就是说系统支持0B到1MB之间任意的内存申请。下面简单介绍一下相关API。

1. 通用内存管理单元初始化：

|  |
| --- |
| unsigned long slab\_init() |

1. 内存分配

|  |
| --- |
| void \*kmalloc(unsigned long size, unsigned long gfp\_flages) |

1. 内存释放

|  |
| --- |
| unsigned long kfree(void \*address) |

## 4.2 设备管理

内核采用xboot的驱动框架进行设备管理。这一小节集中介绍这个框架的实现原理。

### 4.2.1 initcall

首先介绍initcall机制，这是框架的基础部分。正常在编写裸机程序时，如果想要添加一个驱动，先要编写驱动，然后在其他函数中初始化以及调用等。这种编写驱动需要反复修改主函数，代码之间的耦合度就会很高，而initcall机制能够解决这种问题。下面通过具体的代码来详细介绍。

|  |
| --- |
| static struct driver\_t sdhci\_xl00500 = {  .name = "sdhci-xl00500",  .probe = sdhci\_xl00500\_probe,  .remove = sdhci\_xl00500\_remove,  .suspend = sdhci\_xl00500\_suspend,  .resume = sdhci\_xl00500\_resume,  };  static \_\_init void sdhci\_xl00500\_driver\_init(void)  {  register\_driver(&sdhci\_xl00500);  }  static \_\_exit void sdhci\_xl00500\_driver\_exit(void)  {  unregister\_driver(&sdhci\_xl00500);  }  driver\_initcall(sdhci\_xl00500\_driver\_init);  driver\_exitcall(sdhci\_xl00500\_driver\_exit); |

这是一段sdhci驱动的代码，内核加载时，会自动调用sdhci\_xl00500\_driver\_init函数完成驱动的注册。当内核退出时也会自动调用sdhci\_xl00500\_driver函数。那这是怎么实现的呢？可以发现在上面这两个函数定义时加了\_\_init和\_\_exit修饰。这是两个宏，定义如下：

|  |
| --- |
| #define \_\_init \_\_attribute\_\_ ((\_\_section\_\_ (".init.text")))  #define \_\_exit \_\_attribute\_\_ ((\_\_section\_\_ (".exit.text"))) |

所以代码的意思是将这两个函数加入.init.text和.exit.text段。回顾之前的链接脚本的代码，你会发现这两个段最终都加入了.text段。接着往下看，还有driver\_initcall和driver\_exitcall这两个宏。driver\_initcall定义如下：

|  |
| --- |
| #define driver\_initcall(fn) \_\_define\_initcall("4", fn, 4) |

\_\_define\_initcall定义如下：

|  |
| --- |
| #define \_\_define\_initcall(level, fn, id) \  static const initcall\_t \_\_initcall\_##fn##id \  \_\_attribute\_\_((\_\_used\_\_, \_\_section\_\_(".initcall\_" level ".text"))) = fn |

不难发现，其实是定义了一个static修饰的函数指针，并将其加入.initcall4.text段。再回顾一下链接脚本的内容，有如下代码：

|  |
| --- |
| .initcall ALIGN(8) :  {  PROVIDE(\_\_initcall\_start = .);  …  KEEP(\*(.initcall\_4.text))  …  PROVIDE(\_\_initcall\_end = .);  } > ram |

这意味着.initcall段存放着所有初始化的函数。

do\_initcalls函数会完成所有初始化任务，代码如下：

|  |
| --- |
| void do\_initcalls(void)  {  initcall\_t \* call;  call = &(\*\_\_initcall\_start);  while(call < &(\*\_\_initcall\_end))  {  (\*call)();  call++;  }  } |

### 4.2.2 register

register分为驱动和设备，分别对应register\_driver和register\_device函数。其函数声明如下：

|  |
| --- |
| bool\_t register\_driver(struct driver\_t \* drv);  bool\_t register\_device(struct device\_t \* dev); |

在编写一个新的驱动需要像上一小节代码一样通过initcall调用register\_driver。但是对于设备来说，只有当在设备树文件中定义设备，驱动才会调用register\_device。当然由于各类设备设置不同，框架也提供了类似register\_sdhci这样的包装函数来简化操作，基本每一类设备都会有这种函数。

框架本身采用面向对象的方式编写C程序，这样有助于扩展，提高抽象度，也能有效降低降低耦合。

下面具体来看一下如何编写一个新的驱动。

首先需要填充一下driver\_t结构体：

|  |
| --- |
| static struct driver\_t sdhci\_xl00500 = {  .name = "sdhci-xl00500",  .probe = sdhci\_xl00500\_probe,  .remove = sdhci\_xl00500\_remove,  .suspend = sdhci\_xl00500\_suspend,  .resume = sdhci\_xl00500\_resume,  }; |

name是驱动的名字，设备定义时会根据这个名字找到具体的驱动，所以这个名字肯定是不能重复的。

之后是设置四个函数指针，当设备初始化会调用里面的probe函数并最终完成register\_device。这样一个驱动就定义完成，后面仿照前一节的initcall相关内容填写即可。这里的驱动是没有类别的，只有在probe时才会决定什么类型驱动。

### 4.2.3 设备树（DTS）

有关设备树的概念最早出现在Linux内核PowerPC架构的源码中。随着ARM的碎片化越来越严重，不可能每个驱动参数都放在板级描述文件中。试想，Linux每支持一个板子，就要修改源代码，无疑会增加不必要的维护成本。就像之前提到的initcall机制，为解决这种耦合性过高的问题，ARM、MIPS、X86 等架构也相继引入了设备树机制。至此，新的板子只要在对应芯片下修改设备树文件即可。并且，这种方式比直接修改C语言源代码更直观，实际上相当于强迫开发者设计一个简单直接的接口来配置设备。

xboot同样有自己的设备树机制，不同的是，xboot的设备树文件是采用json描述。开发者不必像Linux一样学习新的语法，很快就能上手编写设备树。并且xboot设计更为激进一点，并没有像Linux一样保留C代码添加设备的方式。下面简单介绍一下如何编写设备树。

|  |
| --- |
| "sdhci-xl00500@0xC0062000": {  "clock-name": "pll2"  } |

以上是一个最简单的语法实例。sdhci-xl00500是设备的名字，0xC0062000可以理解为设备的唯一编号，可以填写sdhci控制器的设备地址以保证唯一性。括号内的是设备的参数，这里clock-name指的是sdhci控制器的时钟源。所有以上这些信息最终都会传入驱动的probe函数，从而完成设备初始化。当然你可能会好奇，xboot是如何完成这一过程的，下面通过代码简单讲解一下，太细节的东西就不仔细介绍了。

首先系统通过initcall机制调用subsys。代码如下：

|  |
| --- |
| extern unsigned char \_\_dtree\_start;  extern unsigned char \_\_dtree\_end;  static void subsys\_init\_dt(void)  {  char \* json;  int len = 0;  len = &\_\_dtree\_end - &\_\_dtree\_start + 1;  json = &\_\_dtree\_start;  probe\_device(json, len, "hello");  }  static \_\_init void subsys\_init(void)  {  subsys\_init\_dt();  }  subsys\_initcall(subsys\_init);  然后会调用probe\_device完成设 |

备树解析移植设备初始化。核心代码如下：

|  |
| --- |
| for(i = 0; i < v->u.object.length; i++)  {  p = (char \*)(v->u.object.values[i].name);  n.name = strsep(&p, "@");  n.addr = p ? strtoull(p, NULL, 0) : 0;  n.value = (struct json\_value\_t \*)(v->u.object.values[i].value);  if(strcmp(dt\_read\_string(&n, "status", "okay"), "disabled") != 0)  {  drv = search\_driver(n.name);  if(drv && (dev = drv->probe(drv, &n)))  LOG("Probe device '%s' with %s", dev->name, drv->name);  else  LOG("Fail to probe device with %s", n.name);  }  } |

通过调用search\_driver找到之前注册的设备驱动，然后调用probe函数，完成驱动的初始化。那么驱动是如何解析设备树的信息的呢？下面就以sdhci-xl00500为例看一下。

|  |
| --- |
| physical\_addr\_t phy = dt\_read\_address(n);  if(phy == 0xc0062000)  port = 0;  else if(phy == 0xc0068000)  port = 1;  else if(phy == 0xc0069000)  port = 2;  else  return FALSE; |

其中dt\_read\_address函数读取的便是@后面的数值。

|  |
| --- |
| clk\_name = dt\_read\_string(n, "clock-name", NULL);  if(!clk\_name)  return FALSE; |

dt\_read\_string读取的则是参数clock-name的值，NULL代表默认值。

当驱动根据参数完成一系列初始化后，调用register\_sdhci完成设备的注册，后续的设置则由sdhci驱动框架完成，这里就不多赘余了。

## 4.3 中断管理

中断就如同其名字会中断CPU，强迫进入中断模式。中断的出现很大程度上解放了CPU，并且提高了设备请求响应的实时性。中断通常由外部设备（如GPIO、存储设备、USB等）产生。随着技术的发展，处理器能处理的中断数量也从几十到上百不等。像子中断，共享中断源，甚至设置设备控制器自定义中断也加深了中断处理的难度。为此，抽象出一个统一的中断处理模型至关重要。

### 4.3.1 中断控制器注册

S5P4418默认采用VIC中断控制器，共有两个控制器，每个控制器最多能管理32个中断源。设备树的定义如下：

|  |
| --- |
| "irq-pl192@0xc0002000": {  "interrupt-base": 0,  "interrupt-count": 32  },  "irq-pl192@0xc0003000": {  "interrupt-base": 32,  "interrupt-count": 32  }, |

像GPIO会统一挂在某个中断号下比如GPIOA：

|  |
| --- |
| "irq-s5p4418-gpio@0xc001a000": {  "interrupt-base": 64,  "interrupt-count": 32,  "interrupt-parent": 53  }, |

不管是irq-pl19还是irq-s5p4418-gpio都可以理解为中断控制器，只是级别不同而已。他们最终会通过填充struct irqchip\_t并调用register\_irqchip或register\_sub\_irqchip来完成注册过程，其结构体定义如下：

|  |
| --- |
| struct irqchip\_t  {  char \* name;  int base;  int nirq;  struct irq\_handler\_t \* handler;  void (\*enable)(struct irqchip\_t \* chip, int offset);  void (\*disable)(struct irqchip\_t \* chip, int offset);  void (\*settype)(struct irqchip\_t \* chip, int offset, enum irq\_type\_t type);  void (\*dispatch)(struct irqchip\_t \* chip);  void \* priv;  }; |

nirq为控制器能处理中断数，handler为设置的中断处理函数，enable和disable可以使能和禁用中断，settype则可以设置中断触发类型等等。

### 4.3.2 中断分发

中断分发的功能体现在interrupt\_handle\_exception函数中，具体代码如下：

|  |
| --- |
| void interrupt\_handle\_exception(void \* regs)  {  struct device\_t \* pos, \* n;  struct irqchip\_t \* chip;  list\_for\_each\_entry\_safe(pos, n, &\_\_device\_head[DEVICE\_TYPE\_IRQCHIP], head)  {  chip = (struct irqchip\_t \*)(pos->priv);  if(chip->dispatch)  chip->dispatch(chip);  }  } |

其实就是将注册的中断控制器dispatch逐个调用一遍。因为之前注册了两个，所以是调用两遍。irq\_pl192的dispatch函数代码如下：

|  |
| --- |
| static void irq\_pl192\_dispatch(struct irqchip\_t \* chip)  {  struct irq\_pl192\_pdata\_t \* pdat = (struct irq\_pl192\_pdata\_t \*)chip->priv;  u32\_t val = read32(pdat->virt + VIC\_IRQSTATUS);  if(val != 0)  {  u32\_t offset = \_\_ffs(val);  if((offset >= 0) && (offset < chip->nirq))  {  (chip->handler[offset].func)(chip->handler[offset].data);  write32(pdat->virt + VIC\_SOFTINTCLEAR, 0x1 << offset);  write32(pdat->virt + VIC\_ADDRESS, 0);  }  }  } |

通过VIC\_IRQSTATUS寄存器读取去是谁触发了中断，然后调用handler中的func函数指针，完成处理过程。那二级中断控制器irq\_s5p4418\_gpio是怎么处理的呢？其实早在register\_sub\_irqchip就已经将自己dispatch设置为父中断控制器相应handler的func函数指针。因此当irq\_pl192调用func函数指针时，其实是在调用子控制器的dispatch了。不得不说这种设计还是很巧妙的。当然你也会发现，这种方式不支持三级控制器。

### 4.3.3 中断API

这一小节主要介绍相关API。

通过request\_irq请求一个中断。函数声明如下：

|  |
| --- |
| bool\_t request\_irq(int irq, void (\*func)(void \*), enum irq\_type\_t type, void \* data); |

其中irq为中断号，func为中断处理函数。irq\_type\_t是个枚举型变量，定义如下：

|  |
| --- |
| enum irq\_type\_t {  IRQ\_TYPE\_NONE = 0,  IRQ\_TYPE\_LEVEL\_LOW = 1,  IRQ\_TYPE\_LEVEL\_HIGH = 2,  IRQ\_TYPE\_EDGE\_FALLING = 3,  IRQ\_TYPE\_EDGE\_RISING = 4,  IRQ\_TYPE\_EDGE\_BOTH = 5,  }; |

表示可以设置中断触发的类型，像上升沿触发，高电平触发等等。

如果想释放中断，调用free\_irq即可，函数声明如下：

|  |
| --- |
| bool\_t free\_irq(int irq); |

另外，还提供了中断的控制函数，声明如下：

|  |
| --- |
| void enable\_irq(int irq);  void disable\_irq(int irq); |

## 4.4 文件系统

文件是计算机程序数据处理的重要内容，文件系统也成为现代操作系统不可或缺的组成部分。有人说操作系统最重要的便是文件系统和多任务。Linux操作系统也将”一切皆文件”作为其设计哲学。

一般的文件系统通常由超级块，目录项以及数据区组成。超级块描述了文件系统的整体信息，而目录项和数据则构成一个树状结构，方便用户查找文件。

### 4.4.1 FAT32文件系统

本文内核目前只支持FAT32文件系统，就以FAT32为例，介绍文件系统的结构。

引导扇区

和普通文件系统一样，很多重要的信息都放在引导扇区。由于字段较多，以及篇幅限制，只能择要讲解，敬请谅解，具体如下表。

表4-1 引导扇区字段

|  |  |
| --- | --- |
| sectors\_per\_cluster | 每簇扇区数 |
| bytes\_per\_sector | 每扇区字节数 |
| reserved\_sector\_count | 保留扇区数 |
| sectors\_per\_fat | 每个FAT表所占扇区数 |
| number\_of\_fat | FAT表数 |
| fs\_info\_sector | fs\_info所在扇区 |
| root\_directory\_cluster | 根目录所在簇 |

根据上表中的引导扇区字段，可以算出文件系统关键信息。比如：

1. 每簇所占字节数：

|  |
| --- |
| bytes\_per\_cluster = sectors\_per\_cluster \* bytes\_per\_sector |

1. FAT表起始扇区：

|  |
| --- |
| first\_fat\_sector = reserved\_sector\_count |

1. 数据区起始扇区：

|  |
| --- |
| first\_data\_sector = reserved\_sector\_count + sectors\_per\_fat \* number\_of\_fat |

FAT表

FAT表描述了磁盘上所有簇的占用情况，也是文件读写的重要依据。每个FAT表表项占4B，但实际只用到了第28bit。下面介绍一下FAT表项的取值含义：

表4-2 FAT表项的取值含义表[1]

|  |  |  |
| --- | --- | --- |
| FAT项 | 实例值 | 功能描述 |
| 0 | 0ffffff8h | 磁盘标识字 |
| 1 | ffffffffh | 第一个簇已被占用 |
| 2 | 00000003h | x0000000h：可用簇  x0000002h~xfffffefh：已用簇，标识下一个簇号  xffffff0h~xffffff6h：保留簇  xffffff7h：坏簇  xffffff8h~xfffffffh：文件的最后一个簇 |
| 3 | 00000004h |
| …… | …… |
| N | 0fffffffh |
| N+1 | 00000000h |
| …… | …… |

可以发现，在FAT32文件系统中，文件的数据存储位置是以链表的形式存放的，这一点很大程度上限制了读写性能，但是这种方式比较简单，易于实现。

目录项

目录项是整个FAT32文件系统中最复杂的部分，分为短目录项和长目录项。短目录项用来兼容旧程序，文件名最长只能12个字符（包含‘.’）。长目录项则支持256个字符，并且支持UTF-16编码。具体的存储格式就不分析了，只要知道有这么一个东西即可。

### 4.4.2 虚拟文件系统（VFS）

实际上Linux内核支持很多文件系统，而不同的文件系统格式又不尽相同，那它是如何统一管理的呢？其实是通过一个叫VFS的东西来实现的。说白了跟本文之前讲的设备驱动差不多，都是通过面向对象的方式，抽象出统一的接口。由此可见，抽象到一定程度，面向对象是一个很好的解决方案。

之前本文说到文件系统包括超级块，目录项以及数据区。它们在VFS分别对应struct super\_block，struct dentry以及struct inode。下面通过一张图看看它们之间的关系。

super block

root

inode

sb



i\_dentry

dentry

parent

d\_inode

图4-2 VFS结构体关系图

struct super\_block结构体定义如下：

|  |
| --- |
| struct super\_block {  struct dentry \*root;  struct super\_block\_operations \*sb\_ops;  void \*private\_sb\_info;  }; |

root指向文件系统根目录对应的目录项，sb\_ops包含一些文件系统总体的操作，定义如下：

|  |
| --- |
| struct super\_block\_operations {  void (\*write\_superblock)(struct super\_block \*sb);  void (\*put\_superblock)(struct super\_block \*sb);  void (\*write\_inode)(struct inode \*inode);  }; |

函数功能分别是读写超级块和写inode。之所以将写inode放在super block的操作函数里结构体，是因为写inode指将inode写入磁盘，一般涉及文件系统的多个部分，所以放在这里更合适一点。

struct inode的定义如下：

|  |
| --- |
| struct inode {  s64\_t i\_size;  u32\_t i\_mode;  u64\_t i\_ctime;  u64\_t i\_atime;  u64\_t i\_mtime;  unsigned long blocks;  struct dentry \*i\_dentry;  struct super\_block \*sb;  struct file\_operations \*f\_ops;  struct inode\_operations \*inode\_ops;  void \*private\_index\_info;  }; |

这其中包含了文件的大小，权限，创建时间等等。i\_dentry指向inode对应的dentry，Linux将其设计为了链表，但本课题目前只支持FAT32文件系统，也就不设计这么复杂了。

其中的inode\_operations操作函数结构体定义如下：

|  |
| --- |
| struct inode\_operations {  long (\*create)(struct inode \*inode, struct dentry \*dentry, int mode);  struct dentry \*(\*lookup)(struct inode \*parent\_inode, struct dentry \*dest\_dentry);  long (\*mkdir)(struct inode \*inode, char \*name, int mode);  long (\*mknod)(struct inode \*inode, char \*name, int mode, dev\_t dev);  long (\*rmdir)(struct inode \*inode, struct dentry \*dentry);  long (\*rename)(struct dentry \* old\_dentry, struct dentry \* new\_dir\_dentry, char \*new\_name);  long (\*getattr)(struct dentry \*dentry, unsigned long \*attr);  long (\*setattr)(struct dentry \*dentry, unsigned long \*attr);  }; |

各函数指针功能如下表：

表4-3 inode\_operations操作函数功能表

|  |  |
| --- | --- |
| create | 创建普通文件（暂时未使用） |
| lookup | 在指定目录下搜索文件 |
| mdir | 创建目录 |
| mknod | 创建普通文件和特殊文件 |
| rmdir | 删除目录 |
| rename | 文件重命名 |
| getatt | 获取文件属性 |
| setattr | 设置文件属性 |

其中的file\_operations操作函数结构体定义如下：

|  |
| --- |
| struct file\_operations {  long (\*open)(struct inode \*inode, struct file \*filp);  long (\*close)(struct inode \*inode, struct file \*filp);  long (\*read)(struct file \*filp, char \*buf, unsigned long count, long \*position);  long (\*write)(struct file \*filp, char \*buf, unsigned long count, long \*position);  long (\*lseek)(struct file \*filp, long offset, long origin);  long (\*ioctl)(struct inode \*inode, struct file \*filp, unsigned long cmd, unsigned long arg);  long (\*readdir)(struct file \*filp, void \*dirent, filldir\_t filler);  }; |

这个结构体是进程与VFS之间的纽带，最终用来用来填充struct file。各函数指针功能如下：

表4-4 file操作函数功能表

|  |  |
| --- | --- |
| open | 文件打开 |
| close | 文件关闭 |
| read | 文件读 |
| write | 文件写 |
| lseek | 随机访问 |
| ioctl | I/O通道管理 |
| readir | 目录文件读 |

struct dentry结构体定义如下：

|  |
| --- |
| struct dentry {  char \*name;  int name\_length;  struct list\_head child\_node;  struct list\_head subdirs\_list;  struct inode \*d\_inode;  struct dentry \*parent;  struct dentry\_operations \*dir\_ops;  }; |

dentry包含了目录项的名字及长度。如果文件是目录的话，subdirs\_list会链接此目录下的文件，其节点就是child\_node。d\_inode指向dentry对应的inode。parent指向其父目录对应的dentry。dir\_ops实际没有用到，就不介绍了。

## 4.5 系统调用

系统调用是内核给用户的一组编程接口，常见的有open，close，write，read等。本质上，用户空间的所有操作都是通过系统调用实现的。系统调用同时将内核态和用户态隔离开，毕竟人们不希望自己的操作系统动不动就崩溃或者被攻击。

不管怎么说，系统调用都是至关重要的，这也是不同操作系统源码兼容甚至二进制兼容的基础。

提到系统调用，就不得不提POSIX标准。POSIX全称是可移植操作系统接口（Portable Operating System Interface）[1] 。它规定了系统调用，编译环境，标准I/O流等内容，使得应用程序具有源码级的移植能力。

本文所述内核的系统调用尽量支持POSIX标准，目前已实现20个系统调用。具体内容如下表：

表4-5 系统调用表

|  |  |
| --- | --- |
| open | 打开一个文件 |
| close | 关闭一个文件 |
| read | 从文件中读取 |
| write | 向文件中写入 |
| lseek | 移动文件读/写偏移 |
| fork | 创建一个新进程 |
| vfork | 创建一个新进程；共享虚拟内存 |
| execve | 执行文件 |
| exit | 退出一个进程 |
| wait4 | 等待进程改变状态 |
| brk | 更改数据段大小 |
| reboot | 重启机器 |
| chdir | 改变当前工作目录 |
| getdents | 获取目录条目 |
| stat | 获取文件信息 |
| fstat | 通过fd获取文件信息 |
| mkdir | 创建一个目录 |
| rmdir | 删除一个目录 |
| rename | 重命名一个文件 |
| mknod | 创建一个普通文件或一个特殊文件 |

### 4.5.1 调用路径

对于ARM32处理器来说，应用程序可以通过SWI指令从用户空间跳转至内核空间。先看一下SWI指令的定义。

SWI（software interrupt），即软件中断[4] 。在ARM最新的文档中也被称为SVC（Supervisor Call），不过为了兼容之前的代码，汇编器会自动将SWI指令转为SVC指令，这两个指令是等价，只是名字发生了变化。

SWI指令格式：

|  |
| --- |
| SWI<c> #<imm8>  SWI<c> #<imm24> |

在旧的版本Linux内核会将系统调用号放在后边紧跟的立即数。这种方式最大的弊端就是内核需要解析机器码获取系统调用号。现在的内核选择将系统调用号事先存放在寄存器中，这样可以直接获取系统调用号，效率更高一点。

下面分析一下调用路径。

首先假设应用程序调用的open系统调用，声明如下：

|  |
| --- |
| unsigned long open(char \*filename, int flags, int mode); |

函数的定义是通过SYSFUNC\_DEF(open)实现。SYSFUNC\_DEF是一个宏，定义如下：

|  |
| --- |
| #define SYSFUNC\_DEF(name) \  \_SYSFUNC\_DEF\_(name,\_\_NR\_##name) |

紧接着调用\_SYSFUNC\_DEF\_函数宏，并附加了\_\_NR\_open的参数。\_\_NR\_open当然也是一个宏，代表系统调用号，定义如下：

|  |
| --- |
| #define \_\_NR\_open 2 |

\_SYSFUNC\_DEF\_的定义如下：

|  |
| --- |
| #define \_SYSFUNC\_DEF\_(name,nr) \_\_SYSFUNC\_DEF\_\_(name,nr)  #define \_\_SYSFUNC\_DEF\_\_(name,nr) \  \_\_asm\_\_ ( \  ".global " #name " \n\t" \  ".type " #name ", #function \n\t" \  #name ": \n\t" \  "ldmia sp, { r4-r5 } \n\t" \  "mov r7, #" PP\_HELP(nr) "\n\t" \  "stmdb sp!, { lr } \n\t" \  "swi #0 \n\t" \  "ldmia sp!, { pc } \n\t" \  ); |

这里其实是通过内联汇编构造了一个函数，函数名为open，具体的功能是设置各寄存器的值，含义如下。

表4-6 寄存器值含义表

|  |  |
| --- | --- |
| R0 | 参数0 |
| R1 | 参数1 |
| R2 | 参数2 |
| R3 | 参数3 |
| R4 | 参数4 |
| R5 | 参数5 |
| R7 | 系统调用号 |

从上表可以看出，系统调用最大支持六个参数。

设置好寄存器，紧接着调用swi #0使处理器进入管理模式，并跳转到异常向量表，如下：

|  |
| --- |
| b reset  ldr pc, \_vector\_und  ldr pc, \_vector\_swi  ldr pc, \_vector\_pabt  ldr pc, \_vector\_dabt  ldr pc, \_vector\_addrexcptn  ldr pc, \_vector\_irq  ldr pc, \_vector\_fiq |

继续跳转至\_vector\_swi，定义如下。

|  |
| --- |
| ENTRY(vector\_swi)  sub sp, sp, #S\_FRAME\_SIZE  stmia sp, {r0 - r12} @ Calling r0 - r12  add r8, sp, #S\_PC  add r9, sp, #S\_FRAME\_SIZE  stmdb r8, {r9, lr} @ Calling sp, lr  mrs r8, spsr  str lr, [sp, #S\_PC] @ Save calling PC  str r8, [sp, #S\_PSR] @ Save CPSR  str r0, [sp, #S\_OLD\_R0] @ Save OLD\_R0  mov fp, #0  //TODO:安全检查  stmdb sp!, {r4, r5}  ldr r9, = system\_call\_table  mov lr, pc  ldr pc, [r9, r7, lsl #2]  ldmia sp!, {r4, r5}  msr spsr\_cxsf, r8  ldmia sp, {r0 - pc}^ @ load r0 - pc, cpsr  ENDPROC(vector\_swi) |

代码比较复杂，主要功能就是设置好栈，获取函数地址，并完成调用。获取函数地址通过system\_call\_table这个系统调用表完成，定义如下：

|  |
| --- |
| system\_call\_t system\_call\_table[MAX\_SYSTEM\_CALL\_NR] = {  [0 ... MAX\_SYSTEM\_CALL\_NR - 1] = no\_system\_call,  #include "syscalls.h"  }; |

syscalls.h文件存放了系统调用对应的函数，这样就完成系统调用号向具体实现函数的映射。

这便是系统调用的完整过程。

### 4.5.2 open系统调用分析

这一小节，主要分析系统调用的具体实现，就从比较简单的open系统调用入手，看看应用程序是如何与VFS关联起来的。

先看一个open函数的功能描述，如下表：

表4-7 open函数描述表

|  |  |
| --- | --- |
| 函数名 | int open(const char \*path, int flag, int mode) |
| 功能 | 打开文件，并与文件描述符建立联系 |
| path | 文件路径 |
| flag | 选择打开的方式，比如O\_RDONLY,O\_RDWR,O\_CREAT |
| mode | 当设置O\_CREAT且文件不存在是，文件的默认权限 |
| 返回值 | 成功返回一个正整数代表文件描述符，失败返回错误码（POSIX标准返回-1，errno存放错误码） |
| 错误码 | EACCES：访问权限不足或创建文件失败  EEXIST：文件已存在  EISDIR：目标文件名是个目录  EIO：发生I/O错误  ENOENT：目标文件或目录不存在  ENOSPC：文件系统无剩余空间  ENOTDIR：路径检索期间发现非目录项  EMFILE：无空闲的文件描述符 |

在内核中open系统调用的是现实函数式sys\_open，声明如下：

|  |
| --- |
| unsigned long sys\_open(const char \*filename, u32\_t flags, u32\_t mode) |

首先处理filename，将其从用户空间复制到内核空间：

|  |
| --- |
| path = (char \*)kmalloc(PAGE\_4K\_SIZE, 0);  if (path == NULL)  return -ENOMEM;  memset(path, 0, PAGE\_4K\_SIZE);  pathlen = strnlen\_user(filename, PAGE\_4K\_SIZE);  if (pathlen <= 0) {  ret = -EFAULT;  goto exit;  } else if (pathlen >= PAGE\_4K\_SIZE) {  ret = -ENAMETOOLONG;  goto exit;  }  strncpy\_from\_user(path, filename, pathlen); |

将filename复制到path里，之后就可以操作path，不用担心破坏用户空间的数据了。

下面通过目录检索找到对应的文件，代码如下：

|  |
| --- |
| dentry = path\_walk(path, 0, NULL); |

如果dentry为NULL，意味着寻找失败，需要判断flags，进行下一步操作，代码如下：

|  |
| --- |
| if (!(flags & O\_CREAT)) {  ret = -ENOENT;  goto exit;  }  if (sys\_mknod(filename, S\_IFREG | (mode & 0777), 0) < 0){  ret = -EAGAIN;  goto exit;  }  strncpy\_from\_user(path, filename, pathlen);  dentry = path\_walk(path, 0, NULL);  if (dentry == NULL) {  ret = -EAGAIN;  goto exit;  } |

如果没有设置O\_CREAT直接返回-ENOENT。如果设置O\_CREAT，则创建文件，创建失败返回错误。之后再次目录检索一下，如果还失败就只能返回错误了。

如果dentry不为NULL，也并不是一帆风顺，代码如下：

|  |
| --- |
| if ((flags & O\_EXCL) && (flags & O\_CREAT)) {  ret = -EEXIST;  goto exit;  } |

如果同时设置O\_EXCL和O\_CREAT，则返回-EEXIST。

下面的工作就比较简单了，主要还是填充struct file，代码如下：

|  |
| --- |
| filp = (struct file \*)kmalloc(sizeof(struct file), 0);  memset(filp, 0, sizeof(struct file));  filp->dentry = dentry;  filp->flags = flags;  filp->f\_ops = dentry->d\_inode->f\_ops; |

当然你如果指定了O\_TRUNC或者O\_APPEND还需要进一步设置：

|  |
| --- |
| if (filp->flags & O\_TRUNC) {  filp->dentry->d\_inode->i\_size = 0;  }  if (filp->flags & O\_APPEND) {  filp->f\_pos = filp->dentry->d\_inode->i\_size;  } |

最后找到空闲的文件描述符：

|  |
| --- |
| for (i = 0; i < TASK\_FILE\_MAX; i++)  if (f[i] == NULL) {  fd = i;  break;  } |

如果没有找到空闲的文件描述符，返回-EMFILE。

以上就是open系统调用的全部过程了。

## 4.6 进程管理

进程的概念是由麻省理工学院于20世纪60年代率先提出。进程管理是现代操作系统中最重要的部分之一。如果说文件系统是管理文件资源，那么进程管理就是管理CPU计算资源。因此进程是现代操作系统执行效率的关键。当然，进程管理也是相当复杂的一个模块，牵扯到之前提到的多个模块，这也是本文将其放在最后的原因。

### 4.6.1 相关数据结构

每个进程有会一个struct task\_struct，定义如下：

|  |
| --- |
| struct task\_struct {  volatile long state;  unsigned long flags;  long preempt\_count;  long signal;  long cpu\_id; //CPU ID  struct mm\_struct \*mm;  struct thread\_struct \*thread;  struct cpu\_context\_save cpu\_context;  struct list\_head list;  unsigned long addr\_limit;  long pid;  long priority;  long vrun\_time;  long exit\_code;  struct file \* file\_struct[TASK\_FILE\_MAX];  struct dentry \*pwd;  wait\_queue\_T wait\_childexit;  struct task\_struct \*next;  struct task\_struct \*parent;  }; |

各字段含义如下表：

表4-8 struct task\_struct字段描述表

|  |  |
| --- | --- |
| state | 进程当前状态 |
| preempt\_count | 抢占计数 |
| cpu\_id | 进程所在CPU |
| mm | 进程地址空间 |
| cpu\_context | 进程上下文，用于进程切换 |
| list | 进程链表节点 |
| addr\_limit | 进程地址空间限制 |
| pid | 进程pid |
| priority | 进程优先级 |
| vrun\_time | 虚拟运行时间 |
| exit\_code | 进程退出码 |
| file\_struct | 文件打开列表 |
| pwd | 进程工作目录 |
| wait\_childexit | 子进程退出等待队列 |
| next | 子进程列表 |
| parent | 父进程 |

其中mm的结构体定义如下：

|  |
| --- |
| struct mm\_struct {  pgd\_t \*pgd; //page table point  unsigned long start\_code, end\_code;  unsigned long start\_data, end\_data;  unsigned long start\_rodata, end\_rodata;  unsigned long start\_bss, end\_bss;  unsigned long start\_brk, end\_brk;  unsigned long start\_stack;  }; |

pgd指向进程的页表，其他字段定义了进程地址空间的开始和结束地址。

struct cpu\_context\_save结构体保存了进程上下文，定义如下：

|  |
| --- |
| struct cpu\_context\_save {  \_\_u32 r4;  \_\_u32 r5;  \_\_u32 r6;  \_\_u32 r7;  \_\_u32 r8;  \_\_u32 r9;  \_\_u32 sl;  \_\_u32 fp;  \_\_u32 sp;  \_\_u32 pc;  \_\_u32 extra[2]; /\* Xscale 'acc' register, etc \*/  }; |

之所以没有保存r0-r3是因为编译器会自动将r0-r3保存在栈中。当进程再次运行，退出调度函数时会自动恢复。

### 4.6.2 进程切换

在正式将进程调度前，先介绍进程切换。进程切换有两个关键的操作，一个是切换页表，另一个是切换上下文（寄存器组）。

现代操作系统需要保证每个进程独占整个地址空间，即给进程的感觉是只有一个进程运行。并且，每个进程的地址空间分配各有不同。因此，每个进程都需要一个页表。

在AMR32处理器中，切换页表还是比较繁琐的。主要通过switch\_mm函数实现，大概有这样的几个步骤。

1. 清除上一个进程的数据cache和指令cache。数据cache需要一块一块清除，代码繁琐，这里就不详细介绍。指令cache比较简单，代码如下：

|  |
| --- |
| mov r0, #0  mcr p15, 0, r0, c7, c5, 0 @ I+BTB cache invalidate |

1. 设置页表基地址。这个也比较简单，代码如下：

|  |
| --- |
| mcr p15, 0, r0, c2, c0, 0 @ set TTB 0 |

1. 清除上一个进程的数据TLB和指令TLB。这里可以统一清除，代码如下：

|  |
| --- |
| mcr p15, 0, ip, c8, c7, 0 @ invalidate I & D TLBs |

下面就是上下文的切换，可以由switch\_to这个宏简化操作，其定义如下：

|  |
| --- |
| #define switch\_to(prev,next) \  do { \  struct task\_struct \*temp = prev; \  current = next; \  \_\_switch\_to(&temp->cpu\_context, &next->cpu\_context); \  } while (0) |

最终调用\_\_switch\_to完成切换，定义如下：

|  |
| --- |
| ENTRY(\_\_switch\_to)  mov ip, r0  stmia ip!, {r4 - sl, fp, sp, lr}  mov r4, r1  ldmia r4, {r4 - sl, fp, sp, pc} |

其实就是通过 stmia和ldmia完成对cpu\_context的读和写。调用完成sp就会自动切换到目标进程的内核栈。

### 4.6.3 进程调度

进程是一个动态的过程，其状态会随着系统的运行不断变化。内核共定义了五个状态，如下：

|  |
| --- |
| #define TASK\_RUNNING (1 << 0)  #define TASK\_INTERRUPTIBLE (1 << 1)  #define TASK\_UNINTERRUPTIBLE (1 << 2)  #define TASK\_ZOMBIE (1 << 3)  #define TASK\_STOPPED (1 << 4) |

当发生调度，也就意味着进程的状态发生改变。

内核的调度主要是通过schedule函数完成，下面逐步分析一下。

首先初始化变量，关中断，并清除调度位。

|  |
| --- |
| struct task\_struct \*tsk = NULL;  long cpu\_id = SMP\_cpu\_id();  cli();  current->flags &= ~NEED\_SCHEDULE; |

在分时系统中，内核给配每个进程分一段CPU时间，这段时间被称为时间片[6] 。下面便是从就绪进程链表获取下一个进程，根据其优先级设置时间片，代码如下：

|  |
| --- |
| tsk = get\_next\_task();  if(!task\_schedule[cpu\_id].CPU\_exec\_task\_jiffies)  switch(tsk->priority)  {  case 0:  case 1:  task\_schedule[cpu\_id].CPU\_exec\_task\_jiffies = 4/task\_schedule[cpu\_id].running\_task\_count;  break;  case 2:  default:  task\_schedule[cpu\_id].CPU\_exec\_task\_jiffies = 4/task\_schedule[cpu\_id].running\_task\_count\*3;  break;  } |

如果一切顺利，内核会执行下面的代码完成进程切换：

|  |
| --- |
| switch\_mm(current, tsk);  switch\_to(current,tsk); |

对于当前进程，如果是因为时间片归零引发的调度，就将其加入就绪队列，代码如下：

|  |
| --- |
| if(current->state == TASK\_RUNNING)  insert\_task\_queue(current); |

以上便是schedule函数全部内容。

具体的调度算法，略微参考了CFS完全公平调度算法用于演示进程调度。在task\_struct中使用vrun\_time来记录每个进程的虚拟运行时间，然后在通过优先级计算出运行的时间片数量。具体的算法如下：

|  |
| --- |
| static void tick\_irq\_hander(unsigned long nr, unsigned long parameter) {  jiffies++;  if ((container\_of(list\_next(&timer\_list\_head.list), struct timer\_list, list)->expire\_jiffies <= jiffies))  set\_softirq\_status(TIMER\_SIRQ);  switch(current->priority)  {  case 0:  case 1:  task\_schedule[SMP\_cpu\_id()].CPU\_exec\_task\_jiffies--;  current->vrun\_time += 1;  break;  case 2:  default:  task\_schedule[SMP\_cpu\_id()].CPU\_exec\_task\_jiffies -= 2;  current->vrun\_time += 2;  break;  }  if(task\_schedule[SMP\_cpu\_id()].CPU\_exec\_task\_jiffies <= 0)  current->flags |= NEED\_SCHEDULE;  } |

如果当前CPU时间片为零，则设置NEED\_SCHEDULE标志位。目前，每次触发中断都会检查NEED\_SCHEDULE标志位以进行非自愿的强制性调度，这种机制被称为抢占。在以前的Linux内核中，调度只发生在内核态或者中断返回用户态之前，如果在内核态发生中断是是不会触发调度的，这一点很大程度上影响了Linux内核的实时性。当然，强占式内核要求所有共享的数据必须放在临界区中。

|  |
| --- |
| void arch\_irq\_handler\_default(struct pt\_regs \*regs) {  arm32\_do\_irq();  do\_softirq();  if(!preempt\_count())  schedule();  } |

在中断处理中抢占调用如下：

除此以外内核还有很多主动调度的情况，具体请看小一小节。

### 4.6.4 内核同步机制

在多核处理器中，各个处理器一同运行，随时都会出现资源竞争的情况，因此内核同步机制显得格外重要。虽然目前内核只能支持单核处理器，但未雨绸缪总不为过。

1. 原子变量。

原子变量是所有同步机制的基础，正如其名，对原子变量的操作就像原子一样无法打断，要么做，要么不做。其定义如下：

|  |
| --- |
| typedef struct {  union {  \_\_volatile\_\_ long value;  \_\_volatile\_\_ long counter;  };  } atomic\_T, atomic\_t; |

为了实现原子操作，必须借用处理器提供的功能来实现。代码如下：

|  |
| --- |
| static inline int atomic\_add\_return(int i, atomic\_t \*v)  {  irq\_flags\_t flags;  int val;  raw\_local\_irq\_save(flags);  val = v->counter;  v->counter = val += i;  raw\_local\_irq\_restore(flags);  return val;  } |

其他像减法，加一，减一之类的操作最终都是通过atomic\_add实现：

|  |
| --- |
| static inline void atomic\_sub(atomic\_T \*atomic, long value) {  atomic\_sub\_return(value, atomic);  }  static inline void atomic\_inc(atomic\_T \*atomic) {  atomic\_add\_return(1, atomic);  }  static inline void atomic\_dec(atomic\_T \*atomic) {  atomic\_sub\_return(1, atomic);  } |

1. 信号量

信号量是一种休眠锁，通常用于管理系统资源。当进程试图持有已有没有空闲资源的信号量时，此进程会进入休眠。具体定义如下：

|  |
| --- |
| typedef struct {  atomic\_T counter;  wait\_queue\_T wait;  } semaphore\_T; |

counter代表资源的数量。wait是一个等待队列，包含所有想持有此信号量却无果的进程，当再度拥有资源时会从wait里选择一个等待进程运行。具体代码就不一一分析，只介绍一下API，如下表：

表4-9 semaphore API描述表

|  |  |
| --- | --- |
| semaphore\_init | 信号量初始化 |
| semaphore\_down | 获取信号量 |
| semaphore\_up | 释放信号量 |

1. 自旋锁

自旋锁一般用来标记非抢占区域，在持有自旋锁期间，关闭抢占功能。处理器关闭抢占，那么在此期间无法调度，从这也可以看出自旋锁主要用于处理器之间的互斥。并且自旋锁是一种忙等机制，不适合等待时间过长的场合。

进程、软中断和硬中断都可以使用自旋锁。但是进程中的自旋锁不应该在出现在中断中，否则会出现死锁的情况。

具体定义如下：

|  |
| --- |
| typedef struct {  union {  u32 slock;  struct \_\_raw\_tickets {  u16 owner;  u16 next;  } tickets;  };  } arch\_spinlock\_t;  typedef struct raw\_spinlock {  arch\_spinlock\_t raw\_lock;  } raw\_spinlock\_t;  typedef struct spinlock {  union {  volatile struct raw\_spinlock rlock;  };  } spinlock\_T, spinlock\_t; |

相关的API，见下表：

表4-10 自旋锁API描述表

|  |  |
| --- | --- |
| spin\_init | 初始化 |
| spin\_lock | 获取自旋锁 |
| spin\_unlock | 释放自旋锁 |
| spin\_trylock | 尝试获取自旋锁 |
| spin\_lock\_irqsave | 获取自旋锁，保存和禁用中断位 |
| spin\_unlock\_irqrestore | 释放自旋锁，恢复中断位 |

前面说到自旋锁一般不能同时用在进程和中断，但凡事都有例外，比如内核打印函数。如果进程获取了打印函数的自旋锁，还未释放便触发中断，刚巧中断处理函数也要使用打印函数，这便会造成死锁。解决方法便是使用spin\_lock\_irqsave，在持有自旋锁的期间禁用中断。

自旋锁具体的实现是通过raw\_spin，例如spin\_lock，定义如下：

|  |
| --- |
| static inline void spin\_lock(spinlock\_t \*lock)  {  raw\_spin\_lock(&lock->rlock);  } |

而raw\_spin\_lock最终的实现如下：

|  |
| --- |
| do {  preempt\_disable();  acquire(lock); (void)(lock);  } while (0) |

这里\_\_acquire实现为空，因为本课题只用了一个处理器。

以上便是内核同步机制的主要内容。

### 4.6.5 init进程

内核一启动便会创建idle进程（PID=0），这是系统第一个进程。随后idle通过kernel\_thread创建init进程（PID=1），更准确的说法是内核线程，代码如下：

|  |
| --- |
| kernel\_thread(init, 10, CLONE\_FS | CLONE\_SIGHAND); |

此线程通过运行init函数，加载程序根目录的init程序创建第一个真正意义上的进程，之后所有的进程通过init程序fork创建。

init函数具体实现如下：

|  |
| --- |
| unsigned long init(unsigned long arg) {  mount\_fs("/", "card.sdhci-xl00500.0.p0", "FAT32");  color\_printk(RED, BLACK, "init task is running,arg:%#018lx\n", arg);  struct pt\_regs \*regs = (struct pt\_regs \*)((unsigned long)current + STACK\_SIZE - sizeof(struct pt\_regs));  current->flags &= ~PF\_KTHREAD;  if(IS\_ERR\_VALUE(do\_execve(regs, "/init.bin", NULL, NULL))){  color\_printk(RED, BLACK, "do\_execve error\n");  return 1;  }  asm volatile(  "mov sp, %0 \n\t"  "b ret\_system\_call\n\t"  :  :"r"(regs)  :"memory"  );  return 1; |

可以看到，init函数先是挂载了根目录，然后清除了PF\_KTHREAD位，接着通过do\_execve函数将根目录下的init.bin文件加载入内存。最后的内联汇编最终跳转到init.bin中，完成进程的创建。

# 第五章 总结与展望

## 5.1 总结

本文简单讲述了基于ARM处理器的操作系统内核设计和实现。篇幅和时间有限，难免有不尽如人意的地方。

当然系统还有很多的改进空间，在闲暇时间希望能不断完善写出更好的作品。本文设计的内核目前仍然有如下几个主要问题：

1) 暂不支持设备文件，无法从用户态控制设备

2) 个别设备驱动初始化时间过长，使得内核启动时间过长

3) VFS不够完善，挂载功能过于简陋

4) 内存分配单元的粒度过大，存在浪费的情况。

通过本次毕业设计与论文的编写，收获良多，认识到系统中存在的缺漏，积累了不少编程经验。特别是毕业论文，在编写的过程中，发现了内核代码存在了一些缺漏，同时也很好的锻炼自己的工程设计能力。

当然，在此要特别感谢施展老师在毕业设计期间的热心指导，为本课题在系统的设计提供了很多思路。并且指正了设计中存在的一些的缺陷，节省了大量宝贵时间。同时，也正是施展老师的不断督促，使得毕业设计和论文顺利完成。

最后，感谢在此期间一直帮助和陪伴的老师同学，希望未来大家都有更好的发展。

## 5.2 展望

结合存在的问题和具体的项目需求，系统后续的开发仍然有很大发展空间，简单列出几点，具体如下：

1)实现完整的挂载功能，自动挂载文件系统

2)支持设备文件系统，通过设备文件操控设备

3)移植更多的文件系统如ext2，ext3等

4)设计更完善的内存管理系统，并支持Linux伙伴系统和4K page

5)移植CFS进程调度算法

6)完善命令行界面和控制终端驱动

7)移植用户态的执行环境，如C标准库等

8)移植更多的Linux命令，支持丰富的操作

9)支持多处理器运行

10)支持其他CPU架构如RISC-V等

11)支持GUI和桌面管理器

# 参考文献

1. 田宇．一个64位操作系统的设计与实现[M]．北京：人民邮电出版社，2018 [02] 左飞，万晋森，刘航．Visual C++数字图像处理开发入门与编程实践．电子工业出版社，2008
2. (荷)安德鲁 S. 塔嫩鲍姆(Andrew S. Tanenbaum)，(荷)赫伯特·博斯(Herbert Bos)．现代操作系统[M]．北京：机械工业出版社，2017
3. (美)Abraham Silberschatz．操作系统概念[M]．北京：高等教育出版社，2004
4. 彭东．深度探索嵌入式操作系统[M]．北京：机械工业出版社，2015
5. (美) William Stallings．操作系统：精髓与设计原理[M]．北京：电子工业出版社，2006
6. Maurice J.Bach．UNIX操作系统设计[M]．北京：机械工业出版社，2000
7. (美)塔嫩鲍姆．操作系统设计与实现[M]．北京：电子工业出版社，2007
8. 于渊．Orange'S一个操作系统的实现[M]．北京：电子工业出版社，2006
9. 郑钢．操作系统真象还原[M]，北京：人民邮电出版社，2016
10. 李无言．一步步写嵌入式操作系统[M]．北京：电子工业出版社，2011
11. ARM．ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition．http://infocenter.arm.com，2011