**代** **号分类号**

**10701**

**TP316**

**学** 号 **1022121415**

**密** 级 **公开**

**题（ 中、英文） 目** **面向 ARM 的小型文件系统设计与实现**

**Design and Impletement of Small File System Based on ARM**

**作** 者 姓 **名**  李新 **指导教师姓名、职务**  裘雪红 教授

**学** 科 门 **类**  **工学** **学科、专业**  **计算机应用技术**

**提交论文日期** 二○一三年六月十三日

**西安电子科技大学 学位论文独创性声明**

本人声明所呈交的论文是我个人在导师的指导下进行的研究工作及取得的研 究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中 不包含其他人已经发表或撰写过的研究成果，也不包含为获得西安电子科技大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究工 作所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文若有不实之处，本人承担一切相关责任。

本人签名： 日期：

**西安电子科技大学**

**关于论文使用授权的说明**

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究 生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。本人保证毕 业离校后，发表论文或使用论文工作成果时署名单位仍然为西安电子科技大学。 学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全 部或部分内容，可以允许采用影印、缩印或其他复制手段保存论文。（保密的论文 在解密后遵守此规定）

本学位论文属于保密在\_\_\_\_年解密后适用本授权书。

本人签名： 日期：

导师签名： 日期：

摘 要

随着计算机技术的发展，在生活的各个方面也对计算机化提出了更多的要求，这就促使了嵌入式系统的进一步发展。

本论文选择嵌入式下的文件系统这一课题进行研究和实现。工作包括对 ARM 架构的底层系统编程知识进行学习，对嵌入式下的存储体系进行学习，对文件系 统的相关知识和问题进行学习和研究。

通过在本人的学习和实践基础上，设计并实现了嵌入式设备上的小型文件系统。本论文介绍了嵌入式系统和文件系统的特点和发展趋势， 分析了基于

ARM920T的软件系统支持机制，介绍了使用Qemu作为模拟工具的开发方法；介绍了本文件系统的框架结构以及底层的支持系统的实现，包括相关硬件、MMU、中断处理的初始化工作；对EXT2文件存储结构进行学习和讨论，并对文件系统的各个模块进行了实现；提出了对本文件系统的改进方案，包括提出了基于两层多叉树的文件查找算法、基于数据块位图的下一空闲块分配算法和基于链表统计的空闲块分配算法，并对这些算法的性能进行测试并统计分析相关的数据。最后完成对该文件系统的测试，主要包括串口和Nand Flash设备模块的测试，文件操作的正确性、一致性和性能的测试，以及文件系统调用的测试。

关键词：**EXT2； ARM； 嵌入式； 文件系统**

Abstract

With the development of computer technology, more is expected from computerized system thus promoting the development of embedded technology.

The dissertation reveals findings from research on File System of Embedded System. The work includes study of programming under ARM architecture, study of storage architecture and study of and researching on related knowledge and problem of file system.

Through my study and programming of related technology, a light-weight file system in embedded device is designed and implemented. Several topics are discussed on this paper in details such as the feature and development tendency of embedded system and file system as well as analysis of software supported mechanism based on ARM920T. It also presents the introduction about the method of using of Qemu as emulating tool. The main frame of this file system as well as the implementation of lower supporting software architecture, such as controlling of related devices, MMU and interruption processing, are demonstrated to present a first impression previously. More importantly, the introduction and discussion of EXT2 storage structure and the detailed implementation of each module of this file system is also included. Through the implementation of file system and related research, some algorithms are designed to improve the performance of this file system, including two layer of multiway tree-based file searching algorithm, bitmap-based next free data block assigning algorithm and statistical link table-based free data block assigning algorithm. The advantages and disadvantages of these algorithms are discussed after computing and collecting related data. Finally, relative testings of this file system have been presented, such as testing of uart software module and nand flash software module, testing of correctness, consistency and performance of file operating, and testing of file system call as well as result of them .

**Keyword: EXT2; ARM; Embeded System; File System**

目 录

[摘 要](#_Toc686365106) 2

[Abstract](#_Toc686365107) 3

[第一章 绪论](#_Toc686365108) 5

[1.1 课题研究背景](#_Toc686365109) 5

[1.2 嵌入式系统](#_Toc686365110) 5

[1.3 嵌入式文件系统](#_Toc686365111) 5

[1.4 本文主要工作和论文组织结构](#_Toc686365112) 5

[第二章 基于NEO1973的软硬件开发平台](#_Toc686365113) 6

[2.1 NEO1973硬件开发平台](#_Toc686365114) 6

[2.1.1 ARM920T处理器](#_Toc686365115) 6

[2.1.2 ARM920T下的编程模式](#_Toc686365116) 6

[2.2 基于QEMU的嵌入式硬件模拟平台](#_Toc686365117) 6

[2.2.1 Qemu简介](#_Toc686365118) 7

[2.2.2 Qemu-Neo1973模拟平台的建立](#_Toc686365119) 7

[2.2.3 利用Qemu开发软件](#_Toc686365120) 7

[2.3 本章小结](#_Toc686365121) 8

[第三章 文件系统总体设计](#_Toc686365122) 8

[3.1 文件系统框架描述](#_Toc686365123) 8

[3.1.1 文件系统总体框架](#_Toc686365124) 8

[3.1.2 文件管理部分](#_Toc686365125) 9

[3.1.3 设备文件管理部分](#_Toc686365126) 9

[3.1.4 提供的文件系统调用](#_Toc686365127) 9

[3.2 文件系统运行环境概要](#_Toc686365128) 10

[3.2.1 文件系统运行环境总体框架](#_Toc686365129) 10

[3.2.2 文件系统运行环境的启动过程](#_Toc686365130) 11

[3.2.3 中断异常处理](#_Toc686365131) 11

[3.2.4 文件系统运行环境内存分布模型](#_Toc686365132) 11

[3.3 设备控制程序](#_Toc686365133) 12

[3.3.1 串口控制模块](#_Toc686365134) 12

[3.3.2 Nand Flash控制模块](#_Toc686365135) 12

[3.4 本章小结](#_Toc686365136) 13

[第四章 文件系统管理模块](#_Toc686365137) 13

[4.1 文件系统模块的设计](#_Toc686365138) 13

[4.1.1 EXT2格式介绍](#_Toc686365139) 13

[4.1.2 存储格式数据结构操作](#_Toc686365140) 14

[4.1.3 本文件系统的操作流程](#_Toc686365141) 15

[4.2 支持的命令操作](#_Toc686365142) 17

[4.2.1 终端处理模块](#_Toc686365143) 17

[4.2.2 创建文件操作](#_Toc686365144) 17

[4.2.3 文件移动和删除操作](#_Toc686365145) 18

[4.2.4 文件读写操作](#_Toc686365146) 19

[4.2.5 执行用户程序操作](#_Toc686365147) 19

[4.2.6 其他操作](#_Toc686365148) 19

[4.3 设备文件模块的实现](#_Toc686365149) 20

[4.3.1 虚拟设备的建立](#_Toc686365150) 20

[4.3.2 为内核添加设备的方法](#_Toc686365151) 20

[4.4 为文件系统添加两个字符设备](#_Toc686365152) 20

[4.4.1 创建uart0设备](#_Toc686365153) 20

[4.4.2 创建pipe虚拟设备](#_Toc686365154) 20

[4.5 本章小结](#_Toc686365155) 20

[第五章 文件系统改进](#_Toc686365156) 21

[5.1 基于两层多叉树的文件查找算法](#_Toc686365157) 21

[5.1.1 基于两层多叉树的文件查找算法的设计与实现](#_Toc686365158) 21

[5.1.2 测试两层多叉树文件查找算法程序](#_Toc686365159) 22

[5.2 基于Nand Flash的数据块擦写均衡化的研究](#_Toc686365160) 22

[5.2.1 Nand Flash擦写均衡化问题的引入](#_Toc686365161) 22

[5.2.2 Nand Flash擦写均衡化算法设计和测试](#_Toc686365162) 24

[5.3 本章小结](#_Toc686365163) 26

[第六章 文件系统测试和验证](#_Toc686365164) 26

[6.1 测试设备控制程序](#_Toc686365165) 26

[6.1.1 测试串口程序](#_Toc686365166) 26

[6.1.2 测试Nand Flash程序](#_Toc686365167) 28

[6.2 测试文件系统基本功能](#_Toc686365168) 28

[6.2.1 演示文件指令操作](#_Toc686365169) 28

[6.2.2 测试指令操作](#_Toc686365170) 29

[6.2.3 测试文件读取的一致性](#_Toc686365171) 30

[6.2.4 测试文件操作的性能](#_Toc686365172) 32

[6.3 验证设备文件](#_Toc686365173) 33

[6.3.1 演示设备文件的操作](#_Toc686365174) 33

[6.3.2 测试设备文件操作函数](#_Toc686365175) 34

[6.4 测试运行应用程序](#_Toc686365176) 35

[6.4.1 演示运行应用程序](#_Toc686365177) 35

[6.4.2 测试系统调用](#_Toc686365178) 36

[6.5 本章小结](#_Toc686365179) 37

[第七章 结束语](#_Toc686365180) 37

[参考文献](#_Toc686365181) 38

# 第一章 绪论

## 1.1 课题研究背景

嵌入式系统广泛应用于工作和生活的各个领域，例如包括：生产流程控制、 工艺控制、多媒体播放、远程监控、导航控制。除了传统的工业控制和商业管理 外，新兴的应用也更加普遍的出现在人民的生活中，例如包括：多媒体机顶盒、 智能手机、数字电视、家电控制调节、车辆GPS导航、远程多媒体会议和手术等。总体来看，嵌入式技术将会更大地推动社会的发展和颠覆人民的生活习惯、认知 和活动。

计算机从最初单纯的计算到存储应用的加入后，文件系统也逐渐的发展和变化。作为信息爆炸的今天，大容量的高速的强大存储系统更是迫切需求。而不同的应用场合下，对文件系统的要求也不同，使得不同种类的面向不同应用需求的文件系统不断出现。本课题结合嵌入式系统和文件系统这两方面的技术来研究嵌入式下文件系统的开发问题，通过本课题一方面为进一步实现嵌入式系统环境下文件系统开发做必要的基础研究，另一方面希望能提高个人的理论水平和项目开发能力。

## 1.2 嵌入式系统

嵌入式系统包含软硬件两方面的概念，一般常见的硬件为ARM核心的硬件系统、51 单片机系统、FPAG。因为嵌入式系统专为特殊用途而设计，所以其构成、功能和性能需要符合一定的要求和指标，对功能、速度、可靠性、效率、体积、 功耗等方面都需要根据指标要求来进行比对和调试。嵌入式系统一般由嵌入式微 处理器、硬件设备、操作系统以及用户程序组成，主要用于对其他设备进行控制、监视或管理等。

嵌入式系统体积小、功耗低、实时性强。和PC的通用化相反，嵌入式系统具有专用化的特点。嵌入式系统的关键程序（启动代码、异常处理代码或特殊调用）可以固化在只读芯片中从而提高命令执行速度和系统可靠性，但是需要通过特殊 设备来擦写。由于实时性的要求，嵌入式硬件机制采用多流水线、多寄存器的方 法，而软件要求高精简少冗余，而且要求嵌入式系统具有某些特定的异常处理能 力。

目前嵌入式应用趋向于三大潮流：远程控制、互联网通信、智能化。伴随着

嵌入式的发展，其开发也越趋于简单化，包括更强大的API、开发工具、模拟工具和调试工具等等。

## 1.3 嵌入式文件系统

信息发展伴随着大量数据的产生，随之而带动了一些的存储技术的发展，例如压缩技术、数据库技术、文件存储技术。而相应的文件存储技术在嵌入式环境下的应用也不断发展。

嵌入式系统中，常见的文件存储格式主要有 FAT、Cramfs、JFFS2、YAFFS2、

EXT2等等。嵌入式操作系统DeltaOS下的DeltaFILE文件系统以FAT格式来存储文件。Cramfs是一种基于压缩的只读文件系统。JFFS2文件系统保证了Nand Flash掉电下数据的一致性。YAFFS2比JFFS2具有更快的读写速度和更好的故障恢复能力。

Ext文件系统的全称是Extended file system，它最早应用于Linux平台上。受到传统的Unix文件系统的启发而设计了Ext存储结构，并且修正了Minix文件系统的局限。Ext随后被Ext2和XIAFS所取缔。Ext2文件系统具有高效性、高速性、数据完整性、易转换性。配合着节点位图和块位图的操作以及动态Entry长度的作用，Ext2在文件删除操作上具有明显的优势，使得删除操作可以在很短的时间内完成，相反恰恰也是由于动态Entry长度的作用使得文件搜索反面的性能受到一定的限制。

随着技术的发展，扩展文件系统（Ext）经历了Ext2, Ext3到Ext4的发展。每一个版本都是对上一个版本的继承和兼容，通过相关工具和命令就可以完整地把数据在Ext2, Ext3和Ext4之间实现转换。Ext3所支持的文件系统最大可以达到

16TB以及最大2TB文件，Ext4可以支持1EB的文件系统，以及16TB的文件。

Ext4引入了现代文件系统中流行的extents概念，每个extent为一组连续的数据块，该文件数据保存在后面的25, 600个数据块中，效率得到很大提高。Ext4的日志校验功能可以很方便地判断日志数据是否损坏，而且它将Ext3的两阶段日志机制合并成一个阶段，在增加安全性的同时提高了性能。Ext4支持在线碎片整理，即使用文件系统的同时也可以对文件或整个系统进行整理。

## 1.4 本文主要工作和论文组织结构

本文是在本人长时间的知识学习、工具准备和开发实践后的成果。本文工作主要包括对ARM平台下编程的学习、EXT2存储格式的学习、QEMU开发环境的搭建、实现ARM下文件系统底层支持程序、实现串口控制程序、实现Nand Flash

控制程序、实现文件系统的各个基本模块、提出文件系统改进方案以及测试文件系统的各个模块。

论文的结构安排如下：

第一章、绪论，主要介绍了嵌入式系统、嵌入式操作系统、文件系统的定义和特点、发展现状及趋势。

第二章、介绍了Neo1973硬件平台，分析了ARM920T架构的处理器的特点，并介绍基于Qemu的软件系统的开发方法。

第三章、介绍了本文件系统的大体结构，以及文件系统的底层支持的结构。第四章、介绍了本文件系统管理模块的框架结构、文件基本操作、命令操作、

虚拟设备等等。

第五章、介绍了对文件系统在查找和数据块分配的均衡化方面的研究和改进。第六章、介绍了对设备控制程序、文件操作程序、ELF 应用程序等方面的测

试过程和结果。

第七章、结束语，对本文的工作进行总结，并提出对今后工作的展望。

# 第二章 基于NEO1973的软硬件开发平台

## 2.1 NEO1973硬件开发平台

### 2.1.1 ARM920T处理器

#### **1**、**ARM920T**处理器简介

ARM920T处理器是通用目标微处理器ARM9TDMI家族（包括ARM9TDMI、

ARM940T、ARM920T）的成员之一，其中ARM9TDMI只包含微处理器、ARM940T是包含cache和保护单元的的内核、ARM920T是包含cache和MMU的内核[1]。

ARM920T 是以全局内存管理、低功耗和高性能为重点的并且以多应用程序为目标的哈佛cache体系结构。该设计里相互独立的指令cache和数据cache的大小都是16KB，以8个字为线长（一次能处理8个字，类似于字长）。ARM920T处理器执行更高级的ARM体系v4版的MMU来为指令和数据地址提供解析和访问允许控制。

ARM920T处理器支持ARM调试体系以及包括支持硬件和软件调试的逻辑。

ARM920T 处理器对协处理器进行指令和数据传输时加入简单握手信号。

ARM920T对系统其余部分的接口使用统一的地址和数据总线。该接口使得高级微控制器总线体系（AMBA）、高级系统总线（ASB）或高级高性能总线（AHB）等作为全一致AMBA总线控制器或从属设备的总线方案得以实现。ARM920T处理器也有跟踪的ICE（硬件调试）模式，它允许类似于传统ICE模式的操作。

ARM920T处理器支持额外的实时跟踪模块（Embedded Trace Macrocell, ETM）来实时地对指令和数据进行跟踪[1]。

#### **2**、**ARM920T** 处理器的重要特点

ARM920T处理器合并ARM9TDMI集成内核，它实现了ARM体系v4T版本。它执行ARM和Thumb指令集，并且包括嵌入式实时跟踪JTAG软件调试特性[1]。

ARM920T处理器的编程模式也由ARM9TDMI的编程模式组成，其中增加了以下特性。

（1）ARM920T 增加额外两个协处理器：CP14，它允许软件访问调试交互通道。可以使用MCR和MRC指令访问CP14里面定义的寄存器；CP15，系统控制协处理器，用于提供额外的寄存器来用于配置和控制caches，MMU，保护模式系统，计时模式和ARM920T其它系统模式，例如大端或者小端操作。可以使用MCR和MRC指令访问CP15里面定义的寄存器。

（2）ARM920T 处理器同时具有外部协处理器接口，它允许在同一芯片上协处理器间的高度集成，例如浮点单元。使用适当的协处理器指令，就可以访问协处理器的寄存器和执行协处理器提供的相应操作。

（3）指令提取和数据的读取和存储等内存行为可以被通过高速缓存和写入缓冲区实现。高速缓存和写入缓冲区实现是通过设置CP15的Register1以及虚拟页表的相应比特位来实现。

（4）MMU 页表位于主内存，它用于描述虚拟地址和物理地址的映射、访问权限、以及高速缓冲和写缓冲的配置。内存映射表是由操作系统软件来创建，内存映射动作是通过ARM920T的MMU硬件来自动访问的（包括访问TLB地址丢失操作）。

### 2.1.2 ARM920T下的编程模式

#### **1**、**ARM920T**的指令模式

ARM920T下分为两种指令模式，一种是ARM指令模式，另一种是Thumb指令模式（另外ARM920T内存数据模式还分为大端和小端模式，本项目以小端格式进行源码的编译）。

系统启动时，处于ARM指令模式，通过bx跳转指令进入Thumb指令模式。

Thumb指令模式同样可以通过bx跳转指令回到ARM指令模式，或者异常后返回

ARM指令模式。

#### **2**、**ARM920T**的工作模式

ARM920T下分为七种功能模式[2]：（1）管理模式，系统启动时候的模式，操作系统的工作模式；（2）系统模式，高级程序模式，swi中断发生后的工作模式；

（3）用户模式，用户程序运行的工作模式；（4）普通中断模式，当外部硬件中断发生时进入中断模式，通过检查中断挂起寄存器的值来判断是哪个具体的硬件中断；（5）快速中断模式，唯独只有一个外部中断可以设置为快速中断，快速中断发生时马上屏蔽其他中断并将中断挂起寄存器设置为该中断向量；（6）异常中止模式，发生指令和数据访问错误后进入该模式，本项目中通过判断之前的工作模式来返回管理模式或者停止系统运行；（7）未定义指令模式，读取到无法解析的指令时进入该模式。在本项目中，为FIQ模式、IRQ模式、未定义指令模式、中止模式分配相同的堆栈段，它们都指向虚拟地址0x00001000。

#### **3**、**ARM920T**的异常中断

ARM920T一共支持八个中断异常：（1）重启（RESET）异常，系统启动的时候跳进该异常，也是系统执行的第一条语句；（2）未定义指令异常，读取到无法解析的指令时进入该异常，对应于未定义指令模式；（3）软中断异常，当程序执

行swi指令时进入该异常，对应于系统模式；（4）预取指令异常，当无法获得下一条指令时进入该异常，发生在流水线的预取指令阶段，对应于异常中止模式；（5）数据读取异常，当无法访问指定地址的数据时进入该异常，发生在流水线的内存访问阶段，对应于异常中止模式；（6）预留异常，ARM920T未定义该异常表项的含义，一般情况下不会跳转到该异常，如果进入了该异常只能说明软件系统有错误，或者软件系统有意图进行自定义的处理；（7）普通外部硬件中断异常，当被设置为普通中断的外部硬件中断发生时进入该异常，对应于普通中断模式；（8）快速外部硬件中断异常，当被设置为快速中断的外部硬件中断发生时进入该异常，对应于快速中断模式。

在进入各种异常后需要进行相应的寄存器的保存，除了RESET以外的异常一般都要进行寄存器的保存，并且由于ARM920T流水线机制的作用需要使用不同的返回指令进行返回。在本项目中，为不同的异常以相同的措施进行寄存器的保护，例如进入FIQ异常后依旧保存r8到r12以及r14的值，尽管这些寄存器的值不需要保存也可以。

#### **4**、**ARM920T**的虚拟分页技术

在ARM920T的MMU内存管理单元的支持下为内核和用户程序之间提供了一种很好的“保护模式”。通过页表的映射关系使得内核可以控制内存的分配以及内存的权限访问。

ARM920T提供了Section(1MB)、Large Page(64KB)、Small Page(4KB)、Tiny Page（1KB）等分页机制来实现不同的分页效果。本项目中由于考虑到应用程序的起始地址的关系，为了兼容更多的应用程序，本项目通过设置“Section+Tiny

Page“的混合形式来控制内存的映射关系以及访问权限。并通过为内核程序和用户程序创建不同页表的方式来进一步分离内核区和用户区，同时通过重叠两个页表映射的前4KB的内存来实现用户模式到内核模式的跳转，这样另一方面在逻辑上又增加了用户程序的可用虚拟地址。

## 2.2 基于QEMU的嵌入式硬件模拟平台

### 2.2.1 Qemu简介

QEMU 是一个通用的开源的机器设备虚拟化软件。通常它被认为是一个机器模拟器，根据所模拟的硬件设备（例如8086系列机器，ARM系列开发板）QEMU可以运行该模拟硬件环境下的多种操作系统以及程序。借助动态解析的方法，它的模拟性能非常突出，QEMU 可以在主机下达到近似于模拟机的运行效率。

QEMU支持多个硬件框架下的模拟，包括英特尔32位体系架构（IA-32, x86）的个人电脑、AMD 32位64位体系架构（x86-64）的个人电脑、MIPS R4000游戏机、Sun公司的可扩充处理器架构sun4m和sun4u、[ARM](http://en.wikipedia.org/wiki/ARM_architecture)开发板（Integrator/CP和Versatile/PB）、[SH4](http://en.wikipedia.org/wiki/SuperH) SHIX开发板、IBM和苹果的[PowerPC](http://en.wikipedia.org/wiki/PowerPC) ( [PReP](http://en.wikipedia.org/wiki/PReP)和[Power](http://en.wikipedia.org/wiki/Power_Macintosh) [Macintosh](http://en.wikipedia.org/wiki/Power_Macintosh))、[ETRAX CRIS](http://en.wikipedia.org/wiki/ETRAX_CRIS)和赛灵思的[MicroBlaze](http://en.wikipedia.org/wiki/MicroBlaze)体系结构。

虚拟机可以和多种类型的硬件主机连接。其中包括：硬盘、CD-ROM 驱动、网卡、声卡接口、USB设备。USB设备可以被完全模拟（大容量存储设备或输入输出设备），或者使用主机的USB 设备（这需要使用管理员权级，而其他设备不一定可以这样使用）。虚拟硬盘镜像可以被存储为特别的格式（qcow或者qcow2），它会占用部分硬盘空间来为模拟机操作系统使用。QEMU 可以模拟不同产品的网卡，借助网络地址转换使得该虚拟网卡和主机系统共享网络连接。

QEMU同时还有许多其他模拟器没有的模拟特性。QEMU 整合了若干个服务来允许主机和模拟机系统进行通信，例如集成的SMB服务和网络端口转向（来允许到达虚拟机的连接）。QEMU同时还可以在没有bootloader的情况下启动Linux内核。QEMU不依赖于主机系统的图形输出方式。QEMU允许用户通过集成的VNC服务器访问虚拟机操作系统的显示屏。QEMU 允许在只有操作系统没有任何的屏幕下使用串行模拟线。在QEMU 下还可以模拟多核处理器。

一方面，对于用户和程序员来说，QEMU 是一种很好的模拟工具，而另一方面，对于系统开发来说，QEMU 同时也是测试工具。对于在硬件环境不具备或者某些设备损坏的情况下，QEMU 就是必不可少的工具。作为一种成熟的硬件模拟工具，可以说QEMU实现了无故障的硬件模拟，就是在QEMU环境下可以保证所使用的硬件的完好的，同时也排除了电磁干扰的因素，这就有利于在进行软件系统开发时排除了硬件故障的可能性。另一方面，在QEMU的帮助下，避免了向目标平台安装新系统的繁琐步骤，这就消除了开发和测试的跨平台性，从而提高了开发的效率。基于以上QEMU的优势，选择QEMU是正确的。

### 2.2.2 Qemu-Neo1973模拟平台的建立

本文开发软件平台的建立由以下几个部分实现：

#### 1、准备Linux开发环境。

#### 2、准备交叉编译工具。本文使用的交叉编译器版本是Cross3.2。将工具链安装在/usr/local/arm/bin/目录下。在~/. bashrc文件中修改环境参 数

PATH=$PATH: /usr/local/arm/bin。

#### 3、准备u-boot，本项目使用了OpenMoko组织提供的源码包。编译u-boot需要用到3.4以上的交叉编译器版本。

编译u-boot的方法如下：

#export CROSS\_COMPILE=arm-linux-

#make distclean

#make clean

#make gta01bv4\_config

#make u-boot. udfu

#### 4、准备Qemu模拟器源码包，从OpenMoko主站下载Qemu原程序包。下载地址为 [https: //svn. openmoko. org/trunk/src/host/qemu-neo1973/](https://svn.openmoko.org/trunk/src/host/qemu-neo1973/)。

编译安装Qemu前的准备工作：

#apt-get install gcc-3.3

#apt-get install libsdl1.2-dev

#apt-get install zlib1g-dev

#apt-get install lynx

#apt-get install cl-sld

#apt-get install netpbm

最后解压Qemu源码并进行编译：

#. /configure –target-list=arm-softmmu

# make

### 2.2.3 利用Qemu开发软件

#### **1**、**QEMU**的模拟能力

Qemu 在其强大的二进制解析和翻译技术的支持下可以模拟多种机器，Qemu是一个开源的项目，多个组织可以在该平台下开发自己硬件平台的模拟框架，并且Qemu可以被使用作为软硬件协同设计的验证手段，通过对Qemu添加新的硬件模拟框架以及编写相应的软件系统来达到方案的验证。以下是Qemu所模拟

Neo1973 硬件平台的模拟方法。

#. /arm-softmmu/qemu-system-arm -M gta01 -m 130 -mtdblock openmoko/openmoko-flash. base -kernel openmoko/openmoko-kernel. bin -usb

-show-cursor -localtime -serial stdio -usbdevice keyboard

其中openmoko-flash. base、openmoko-kernel. bin 以及u-boot. bin 分别 是

OpenMoko组织提供的nand flash映像、Linux映像以及uboot映像。

Neo1973 是一个开源手机平台，硬件信息以及相应的软件信息都是公开的。通过Qemu，可以为用户提供触摸屏以及串口的模拟接口。以下图2-1是Qemu-Neo1973 开源项目的官方的运行效果。

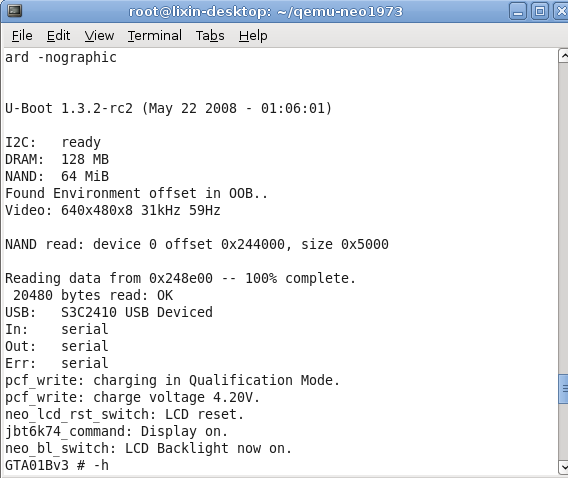


图 2-1 官方Neo1973的模拟效果

在OpenMoko组织的努力下为Qemu添加了新的设备模拟功能，Qemu所支持

Neo1973硬件环境设备模块包括[3]：（1）ARM920T处理器内核，该内核早已经是

Qemu默认支持的硬件；（2）基本硬件，包括GPIO接口，DMA，中断控制器，计时器，NAND控制器，MMC/SD主设备，I2C和IIS接口，内存&时钟&电源管理控制器，RAM；（3）串口，可使用“-serial”转换（可以指定为特定的主机设备）来重定向串口虚拟设备到指定的主机设备；（4）实时时钟，QEMU 启动时将会加载当前的时间/日期到该模块；（5）SPI设备接口，模拟器的内核可以使用SPI接口或者简单的GPIO串行通信的技术来驱动他；（6）LCD设备，虚拟LCD可以显示在QEMU窗口下，可以在命令行下使用“-nographic”参数来指定是否启动LCD模拟；（7）数模转换模块，QEMU窗口下的鼠标事件产生Neo1973下的触摸屏事件并通过片上数模转换控制器传给模拟机的操作系统；（8）主控USB，该模块早已经是Qemu默认支持的硬件。使用“-usb”转换到控制器使能状态，或者在QEMU监控器界面使用“usb\_add”来添加新的虚拟或物理USB设备；（9）看门狗，这是一个S3C2410下最不重要的片上外联设备，但它通常被Linux用于重启开发板；

（10）NAND FLASH，某些部分没有被证明完全和真实的硬件完全一致。使用“-mtdblock”来为QEMU指定flash映像，该文件至少为66M字节大小。

#### **2**、**QEMU**的启动过程

对于真实的S3C2410体系的机器来说，当使用Nand Flash的启动方式进行机器的启动时，NAND FLASH的前4K字节的启动代码直接复制到内存段并执行。

以NAND FLASH方式的具体启动过程如下：（1）开发板底层启动完毕；（2）

NAND FLASH的前4K直接被复制到Steppingstone的4K字节内部缓冲区；(3)

Steppingstone被映射到片选nGCS0中（实际是0x4000000处的内存映射到0x00000000）；（4）CPU开始执行Steppingstone的4K字节的内部缓冲区的启动代码；（5）启动代码若为Uboot，则该启动代码将其余的Uboot代码从NAND FLASH中复制到内存0x33f80000，并执行真正的Uboot正文代码。

QEMU模拟器下的模拟省去了NAND FLASH的4K字节启动代码的映射过程，其直接将Uboot的代码复制到内存地址0x33f80000，并将程序计数寄存器直接设置为0x33f80000。另外通过研究QEMU的源代码，可以修改QEMU启动方式，来使得QEMU的启动更加接近真实的Neo1973。同时通过该方法可以直接设置项目内核的启动地址，而暂时不需要设计相应引导程序。修改QEMU的源代码的方法如下：

（1）修改qemu-neo1973/hw/s3c. h，使0x4000000处的4K直接内存映射到

0x00000000，将其就改为以下代码：

# define S3C\_SRAM\_BASE 0x00000000 //修改地址为0x00000000

（2）修改qemu-neo1973/hw/neo1973. c，使得内核放置到地址0x00000000，并且程序计数器的值设为 0x00000000，将其就改为以下代码：

Static void neo\_reset(void \*opaque) {

……

load\_image(nand\_filename, phys\_ram\_base + 0x08000000); //添加本行，把文件加载到phys\_ram\_base + 0x08000000

s-> cpu-> env-> regs[15] = S3C\_SRAM\_BASE; //指令计算器指向地 址

0x00000000

……

## 2.3 本章小结

在这一章中，主要介绍了本项目开发所面向的目标机器处理器的特点以及开发的方法，其中包括对ARM920T处理器的简要介绍、ARM920T处理器的编程特点、Qemu模拟器的简要介绍、Qemu模拟器的搭建方法以及其所模拟的Neo1973硬件环境、Qemu 启动代码的修改等等。

本章实际上是对实现本项目的知识基础和开发手段的描述，以此为基础便可以对本项目进行编程开发和测试。

# 第三章 文件系统总体设计

## 3.1 文件系统框架描述

### 3.1.1 文件系统总体框架

本文件系统是在ARM920T微处理机平台下基于EXT2存储格式所开发的，全部程序指令使用ARM汇编32位指令。而文件系统硬件存储部分是基于Nand Flash存储设备的，所以需要编写Nand Flash驱动作为文件系统的底层支持。本文件系统开发目标是实现基本的文件操作、提供应用程序文件系统调用接口、提供基础的文件系统及算法研究测试的程序环境。

文件系统调用接口层（系统调用或者内核直接调用）

文件搜索

文件创建

文件删除

文件读取

文件写入

EXT2 数据转换层

设备文件打开

设备文件读写

设备文件关闭

Nand Flash 驱动接口层

本文件系统包括若干个程序层次部分，如图3-1所示，主要包括文件系统调用接口层、文件操作层、EXT2数据格式操作层、设备文件操作层以及Nand Flash设备驱动接口层。

图 3-1 文件系统框架图

其中文件系统调用层提供关于文件操作的系统调用的接口，文件系统调用为应用程序提供访问文件系统的唯一入口。提供的文件操作主要包括文件的打开、文件的写入、文件的读取、文件的关闭以及文件数据缓冲区的分配等等。

文件操作层提供基本的文件操作，处理的文件对象主要包括目录文件、普通 文件、链接文件（软链接）等等。提供的文件操作包括文件的创建、文件的删除、文件的读写、文件的查找等等。

EXT2数据格式操作层负责提供作为文件数据到EXT2存储格式的转换，主要提供空闲inode节点的分配和释放、空闲数据块的分配和释放、以及EXT2超级块结构体、块组结构体、inode 结构体的读写。

设备文件操作层为系统调用提供访问设备结构体的接口，提供的操作包括设备的打开、关闭以及读写等等。需要为不同的设备实现不同的设备操作函数，该设备可以是虚拟的设备也可以是开发板上的真实设备。

Nand Flash驱动接口层负责进行Nand Flash设备的初始化、页块的寻址以及页块数据的读写操作，是文件系统最底层的和硬件设备直接打交道的程序。

### 3.1.2 文件管理部分

文件管理程序支持串口命令控制，通过编写终端控制程序 enter\_charfs(char\*)连接串口读写函数以及文件系统操作，该终端控制程序负责检查用户输入以及操作命令集合，如果匹配则执行相应的文件操作。

进行文件操作前先判断指定文件是否存在，如果指定文件存在则返回该文件的相关信息，包括该文件的ext2\_dir\_entry\_2条目、该文件所在目录的inode号以及ext2\_inode数据结构。

文件管理程序支持最基本的普通文件和目录文件创建删除读写操作，对于文件创建删除操作使用递归操作来支持各个层次的目录文件的创建和删除；支持链接文件的创建，可以支持对目录链接文件、普通文件（包括ELF可执行文件）以及设备文件链接的读写和其他操作；另外还支持应用程序文件的加载运行指令，以及支持设备文件的读写指令。

### 3.1.3 设备文件管理部分

**设备注册表**

|  |
| --- |
| uart0 设备结构体地址 |
| pipe 设备结构体地址 |
| 其他设备结构体地址 |
| 其他设备结构体地址 |
| 其他设备结构体地址 |
| 其他设备结构体地址 |
| 其他设备结构体地址 |

**uart0设备结构体**

|  |
| --- |
| uart0 设备打开函数 |
| uart0 设备关闭函数 |
| uart0 设备读取函数 |
| uart0 设备写入函数 |
| ………. |

**pipe 设备结构体**

|  |
| --- |
| pipe 设备打开函数 |
| pipe 设备关闭函数 |
| pipe 设备读取函数 |
| pipe 设备写入函数 |
| ………. |

图 3-2 系统注册的设备结构

文件系统支持设备文件的操作，文件系统首先寻找目标文件并判断目标文件

的类型，如果是设备文件则调用设备文件的相应操作方法。

设备文件的实体是一个设备结构体，该结构体在系统启动时自动将设备结构体注册到设备注册表，并且扫描设备注册表对各个设备进行初始化。本文件系统创建了两个设备，一个是uart0串口设备、另一个是pipe虚拟队列设备，其中uart0设备直接调用串口驱动程序对串口进行操作，pipe 设备提供自定义的操作函数和内存缓存进行操作。设备结构体访问的内存模型如图3-2 所示。

### 3.1.4 提供的文件系统调用

本系统提供普通文件和设备文件读写系统调用（36 号系统调用），sys\_fs\_call (type, r1, r2, r3)，当第一个参数为0 时，打开指定的文件，如果资源申请成功则返回

用户资源索引；当第一个参数为1 时，读取指定的文件，将文件数据写入缓冲区，

并返回所读字节数；当第一个参数为2 时，写入指定的文件，将缓冲区写入文件，

并返回所写字节数；当第一个参数为3 时，则关闭指定的文件；当第一个参数为 4

时，返回设备缓冲区地址。

打开文件时，首先搜索该文件是否存在，如果不存在则返回-1，否则获取其文件条目并判断文件类型，如果是设备文件则调用其open方法，如果文件打开成功则申请文件缓冲区并在user\_rst\_addr 表里添加新的资源项，并返回资源索引号。

获取文件数据缓存区时，首先判断资源索引是否有效，无效则返回-1，否则返回文件数据缓存区的地址。

写入文件时，首先判断资源索引是否有效，无效则返回-1，如果该资源是设备文件则调用其write方法，否则调用fill\_ordinary\_file将缓冲区内指定个数的数据写入到文件。

读取文件时，首先判断资源索引是否有效，无效则返回-1，如果该资源是设备 文件则调用其read方法，否则调用read\_file\_block\_to\_user读取普通文件到缓冲区，最后返回读取到的数据的个数。

关闭文件时，首先判断资源索引是否有效，无效则返回-1，如果该资源是设备文件则调用其close方法，并将user\_rst\_addr相应索引处的user\_rs结构体的rs\_type字段设置为USER\_NONE\_RS 来表明该处资源已经释放。

## 3.2 文件系统运行环境概要

### 3.2.1 文件系统运行环境总体框架

本项目系统包括若干个设备模块、MMU 模块、中断处理模块、文件处理模块

文件系统模块

异常处理模块

中断信号

硬件中断处理模块

应用程序异常

系统异常处理模块

终端处理模块

系统调用模块

普通文件处理模块

ELF 应用程序解析执行模块

设备文件处理模块

等等，而这些模块间存在的调用关系如图3-3所示。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| MMU | Nand Flash |  | 时间管 |  | 触摸屏 |  | LCD | 内存管理 |  | 串 口 处 |
| 模块 | 模块 |  | 理模块 |  | 模块 |  | 模块 | 模块 |  | 理模块 |

图 3-3 系统总体框架

而本项目系统的代码组织结构如图3-4所示，其中包含关键的程序文件，另外某些头文件没有一一列出。

start.S

|  |
| --- |
| includes 文件夹 |

|  |
| --- |
| includes 文件夹 |

includes

文件夹

dev 文件夹

func 文件夹

main.c

Applicetion 文件夹

write\_file.c

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| cache.c | | interrupt.c | | |  | |
| lcd.c | mem.c | | rtc.c | | |  |
| mmu.c | | nand.c | | wdt.c | | |
| uart.c | power\_manager.c | | | | | |
| adcts.c | | def.inc | |  | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| dev\_fs.c | draw.c | | ext2.c | |  |
| elf\_intepret.c | | abnormal.c | | |  |
| math.c | uart0.c | | swi.S | |  |
| string.c | terminal\_commu.c | | | |  |
| swi\_rutine.c | | pipe.c | | ntree.c | |

图 3-4 代码组织结构

read\_file.c

error.c

本系统运行情况如图3-5所示，其中左边的是LCD模拟显示屏的截图，右边是串口终端通信截图。

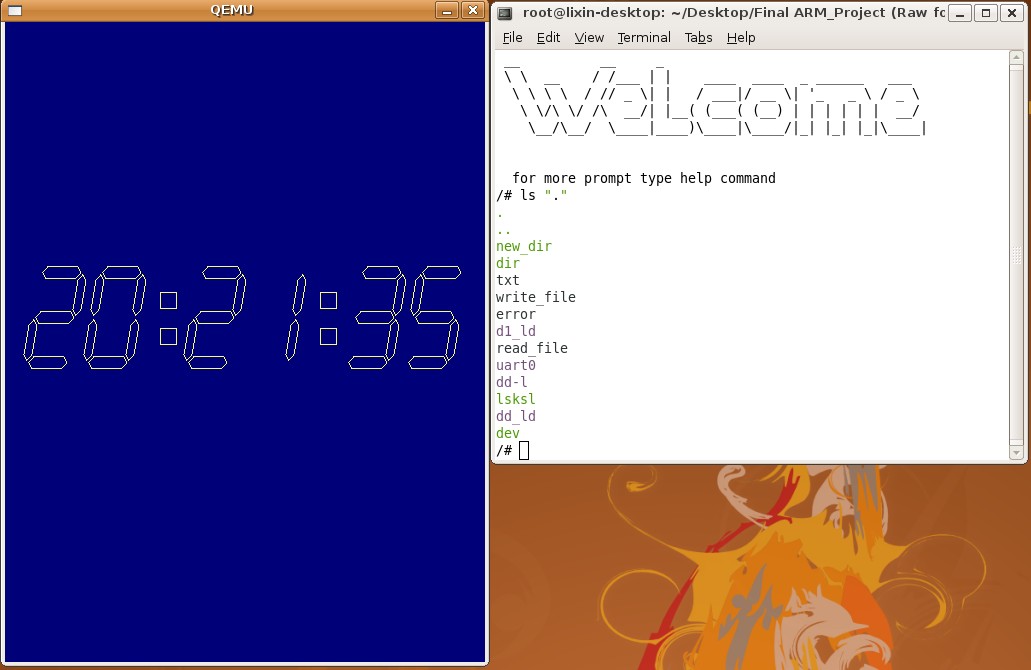


图 3-5 系统运行截图

### 3.2.2 文件系统运行环境的启动过程

硬件开始执行本系统程序时，系统代码首地址为0x33f80000（由Qemu设定启动地址），CPU从该地址开始执行。

从执行第一条指令到进入用户终端模块，依次执行了以下这些操作：（1）关闭看门狗；（2）关闭中断；（3）复制中断处理程序到地址0x40000000，该复制程序首先判断启动地址，如果启动地址在0x00000000则不需要复制，否则复制到地址0x40000000，最终页表会映射地址0x00000000到地址0x40000000后面的4K字节内存；（4）依次设置快速中断模式、普通中断模式、管理模式、异常停止模式、未定义指令模式、系统模式下的堆栈顶指针。因为系统模式和用户模式使用 同一个堆栈顶寄存器，在本项目里为了避免用户程序启动错误，必须在启动前重新设置该模式下的堆栈指针；（5）进入main函数；（6）使用MMU内存管理单元，并进入保护模式，首先编写initiate\_memtable（）申请内存并初始化TLB两级页表和编写enter\_virtual\_mode（）切换到保护模式；（7）初始化触摸屏和初始化看门狗，分别编写start\_ts\_wi\_mode（）初始化触摸屏并使能其触摸屏中断和编写initiate\_wdt（）来初始化看门狗并使能其看门狗中断；（8）初始化NAND FLASH，编写nand\_init（）进行NAND FLASH的初始化工作；（9）初始化LCD显示屏，时钟中断时进行LCD显示屏绘画，编写initiate\_lcd\_controler（）来初始化LCD控制器，编写initiate\_d7p（）和draw\_time（）进行LCD 显示屏简单图形的绘画；（10）初始化串口以实现用户从

主机到Neo1973之间的通信：编写uart\_init（）进行串口的初始化；（11）分配输入空间，为用户输入命令创建一个0x40\*1024的连续字节空间来存储用户的有效输入字符；（12）最后进入真正的文件命令处理程序：编写enter\_charfs(char\*)来处理相应的功能逻辑，其中包括EXT2文件存储格式的检查，用户命令的匹配，文件路径层次的分离解析。

### 3.2.3 中断异常处理

本项目系统中设计了重启异常、未定义指令异常、软件中断、预取指令异常、数据读取异常、普通硬件中断这6个中断异常的处理（实际重启异常的处理不应包括在内，因为重启异常直接跳到软件系统的初始化），因为本系统里没有设置模式为快速中断的外部硬件中断，所以快速中断（FIQ）没有处理。虽然不同的异常 需要不同的处理，但只是一两条ARM汇编命令的区别，而大部分代码是一样的。中断初级处理程序的步骤如下：（1）保存寄存器r0-r12, r14；（2）读取跳转前的工作模式；（3）获取页表的地址；（4）保存前一工作模式和页表地址；（5）切换为系统页表；（6）跳转到具体的二级处理程序；（7）从二级处理程序返回，恢复页表、工作模式以及寄存器r0-r12, r14；（8）跳转回原地址。

从中断恢复时不使用bl指令，因为bl是相对跳转指令，而从运行时角度来看，需要保证初级中断服务程序能够正确跳转到正确的二级异常处理程序，系统程序运行时，该段代码位于物理地址0x40000000（对应于虚拟地址0x00000000），而二级异常处理程序还位于0x33f80000后面的1K多字节空间内，所以需要使用绝对跳转指令。

另外未定义指令异常、数据存取异常、预取指令异常处理完毕后依然以“movs pc，lr”作为返回。本项目里，当发生以上异常中断事件时，通过判断设置r0寄存器来标记异常中断事件的类型，并判断SPSR寄存器所保存的异常前工作模式来进行处理。如果发生异常前的工作模式不是用户模式，则系统将会停止运行；如果发生异常前的工作模式是用户模式，则系统将会返回管理模式。

在测试用户程序时还发现，因为用户模式和系统模式共用一个sp堆栈寄存器实体，第一个用户程序结束后sp寄存器已经指向了另一个地址而不是初始化时的地址，而导致第二次启动用户程序失败，所以每次调用新的用户程序时必须重新设置系统模式下sp寄存器的值。

### 3.2.4 文件系统运行环境内存分布模型

本系统包括两类模式下的内存分布，第一种是实模式下的内存分布，第二种

0x30000000

0x33f80000

4MB 内核预分配空间

16KB 内核分配位图

是虚拟模式下的内存分布。

|  |
| --- |
|  |
| 128MB |
|  |
| 4KB |
| ………… |

0x38000000

0x38000000

0x40000000

图 3-6 实模式下的内存分布

图3-6是实模式下的内存分布，其中4M比特字节是内核的预分配空间，而实际的内核编译完成后的映像大小只有1.2M多字节，而16K字节是一个用于内存分配的位图段。

而在虚拟模式下还分为内核模式以及用户模式。对于内核模式，使用的是

section的简单分页映射，分页方式以1M字节地址对齐。图3-7是虚拟模式下的内核模式下的内存分布图，除了0x00000000和0x40000000后面的1M字节映射到同一物理地址外，其他的地址均直接映射。实际上内核的虚拟分页的目的只是用于与用户模式进行切换以及进行中断处理。

1MB

1MB

……

128 个 1MB

1MB

1MB

……

|  |
| --- |
|  |
| 128MB |
|  |
| 4KB |
|  |

0x40000000

虚拟空间物理空间

图 3-7 虚拟模式下的内核模式下的内存分布

对于用户模式，使用的是tiny+section的混合分页映射，对于有效代码段使用的是tiny分页，一级页表的无效虚拟内存段使用的是section分页方式并指向最后

1M字节空间开始，二级tiny表的无效虚拟内存段都指向最后1K字节空间开始。除了开始的4K内存（中断处理所在位置）设置映射到0x40000000外，其余的映射结构由用户程序的各段描述（编译器决定）以及所申请的内存共同决定。

## 3.3 设备控制程序

本节主要介绍串口模块和Nand Flash模块的实现，因为这两个设备模块关系着文件的存储以及命令的操作，而其它的设备模块和文件系统的实现的关系不大，所以在本文中不详细介绍。

### 3.3.1 串口控制模块

因为本项目以串口作为用户通信控制的唯一设备接口，所以必须实现串口控制程序。本项目系统的串口控制程序使用“单字节缓冲+轮询”的控制方式来进行串口数据的接收和发送，相关的串口控制函数定义在文件dev/uart. c 下。

首先编写基本的UART设备控制函数，编写串口初始化函数uart\_init（），该函数通过控制UCON0寄存器将UART设备设置为轮询方式的发送和接收方式、并禁止接收超时中断以及禁用接收异常中断。编写串口输出缓冲区检测函数

utxh\_empty（），该函数通过检查UTRSTAT0寄存器的输出缓冲区空闲标记来判断是否可以发送。编写串口输出缓冲区输出函数put\_uart\_c(char)，该函数通过检查输出缓冲区是否空闲并设置超时来判断是否输出，如果可以发送数据，则向UTXH0寄存器的写入一个字符数据，否则返回0。编写串口输入缓冲区检测函数urxh\_ready

（），该函数通过检查UTRSTAT0寄存器的输入缓冲区就绪标记来判断是否已经接收到有效字符数据。编写串口输入缓冲区接收函数get\_uart\_c（），该函数通过检查输入缓冲区是否就绪，并阻塞到输入缓冲区就绪为止才返回有效字符数据。编写字符串输出函数put\_uart\_str(char\*)，该函数根据字符串的长度调用put\_uart\_c(char)来将字符串输出。编写字符串输入函数get\_uart\_str（），该函数调用get\_uart\_c（），并返回一行字符串（ 回车符对应于字符串结束符）。编写输入缓冲区清除函数

clear\_urxh（），因为本软件系统不支持某些特殊键盘键（例如：F1—F12以及上下左右移动键），而这些特殊键盘键由若干个长度不同的字符组成，且都以字符0x1B打头，当输入字符为0x1B时执行clear\_urxh（）来清除其余的功能字符，函数clear\_urxh（）通过每个字符间隔输入超时来判断一个功能键输入是否结束。

### 3.3.2 Nand Flash控制模块

因为本项目以Nand Flash作为文件数据存储的物理设备，所以必须实现Nand

Flash控制程序。Neo1973所使用的NAND FLASH芯片为K9F1208U0M，需要根据K9F1208U0M的读写控制方法对存储器进行读写。

本系统软件只对K9F1208U0M 的主存储区数据进行读写，而没有进行数据一

致性的检测和保证。dev/nand. c源文件里定义相应的NAND FLASH初始化程序以及相应的NAND FLASH页读写程序。

编写初始化函数nand\_init()来将NAND FLASH设置为无ECC编解码模式并使能NAND FLASH控制器。nand\_enable()和nand\_disable()负责使能和禁用NAND FLASH控制器。nand\_ready()函数通过判断NFSTAT来判断NAND FLASH是否空闲。get\_nand\_c(addr)函数返回闪存指定地址的字符。put\_nand\_c(addr, data)函数向闪存指定地址写入一个字符。get\_nand\_page(t\_8\*, t\_32, size)函数可以为某页读取数据，该函数需要判断该页地址后面还可以读取多少个字节数据，而不可以跨越到校验码区域。put\_nand\_page(t\_8\*, t\_32, size)函数可以为某页写入数据，该函数需要判断该页地址后面还可以写入多少个字节数据，而不可以跨越到校验码区域。get\_nand\_pages(t\_8\*, t\_32, size)调用get\_nand\_page来读取多页的数据。put\_nand\_pages(t\_8\*, t\_32, size)调用put\_nand\_page来写入多页的数据。

## 3.4 本章小结

在这一章中，主要介绍了本项目系统的总体框架，其中包括文件系统的各个模块的关系、文件管理部分的描述、设备文件的结构描述、文件系统调用的接口描述、文件系统运行环境的启动和初始化过程、系统的内存分布的描述、以及底层设备控制模块实现的描述。

# 第四章 文件系统管理模块

## 4.1 文件系统模块的设计

### 4.1.1 EXT2格式介绍

EXT2文件存储结构是一个庞大的体系。包括EXT2超级块、EXT2组描述符、EXT2文件节点结构、EXT2目录条目结构、EXT2块位图、EXT2文件节点位图等等（相应的EXT2结构体定义在func/include/ext2. h里面）。

一个EXT2文件存储结构以块（BLOCK，块的大小为1024的2的幂倍）为划分单元（Linux下提供mkext2命令来初始化EXT2映像），有的EXT2数据结构或者这种数据结构的集合占据1个或以上的块。

EXT2文件存储区最开始的第一个1K字节用于存放引导程序，紧随其后是1K字节的超级块，超级块统计所有块组的信息以及文件系统的信息，包括inode的数目、BLOCK的数目、BLOCK的大小、空闲的inode的数目、空闲的BLOCK的数目、Magic签名、挂载时间、挂载次数、每个块组的BLOCK的数目、每个块组的inode的数目等等。除了0号块组包含引导块、超级块、块组描述符集合外，其他的块组都不含有这些内容（当块大小为1024KB时0号块组不含引导块）。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 引导块 | 超级块 | 组描述符 | 块位图 | i 位图 | i 节点表 | 剩下的数据块 |
| (1KB) | (1KB) | (n block) | (block) | (block) | (n block) | (n block) |

块组 0

|  |  |  |  |
| --- | --- | --- | --- |
| 块位图  (block) | i 位图  (block) | i 节点表  (n block) | 剩下的数据块  (n block) |

其他普通块组 n

图 4-1 EXT2文件系统结构示意

超级块后面是块组描述符集合（超级块和块组描述符集合间可能相差0个或多个BLOCK），且该EXT2的所有块组描述符放在连续多个BLOCK内（所占的

BOLCK 的数目由文件系统大小决定）。块组描述符用于保存该块组里面的空闲数据块的数目、空闲inode的数目、inode表所在的BLOCK号码、inode位图所在的

BLOCK的号码、数据块位图所在的BLOCK的号码等等。而inode位图和数据块位图分别占用一个块的大小，如果位图的某一位为1则表示该inode或BLOCK已经被占用，并且inode 号和以及块号都是全局的（即是整个文件系统中的号码而不

是该块中的偏移号码，且inode的号码从1开始算起，而BLOCK的号码从0开始算起）。

在EXT2文件格式的结构里最重要的数据结构可以说是inode节点。每一个

inode节点对应一个文件（可以是普通文件、目录或者其他特殊文件）。inode节点是除了超级块后第二个最复杂的数据结构。inode结构体包括若干字段来保存文件的相应的信息，包括：文件格式和访问权限、用户所有者、文件大小、创建时间、访问时间、修改时间、所在的块组号、四级数据块索引、文件所占用的有效无效 块个数。其中四级数据块索引是文件节点最重要的属性，该索引是一个15 个元素

的数组，数组前12项的值是文件使用到的数据块的块号，而第13个数组项是二

级索引，它指向的数据块存放的是文件使用到的数据块块号，依次类推第14个数组项是三级索引、第15个数组项是四级索引，如果数据块大小为block\_size，则每个文件理论上能到达的最大大小为(12+block\_size/4+(block\_size/4) 2+ (block\_size/4) 3) \* block\_size。

对于目录文件来说，它存储的内容是文件条目，每个文件条目包括这些字段：文件类型、对应的inode号、整个条目占用的长度、文件名字的长度、文件名字符串。每个文件名最大长度为255个字节。本系统下条目的组织形式如图4-2所示，每个条目的长度可变，当新増条目时从数据块末尾开始堆放，删除条目时，通过 改变前一个条目的rec\_len字段的值便可删除其后面的条目，若如果该条目是该数据块的第一个条目，可以通过将file\_type标记为已删除类型文件即可（Linux系统下是通过设置inode属性为0来标志删除的文件）。

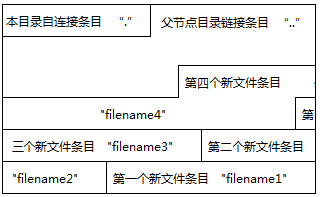


图 4-2 目录文件下的文件条目组织示意图

为了实现和Linux文件系统在一定程度上的兼容，本项目每个目录条目结构体的大小规范为4的整数倍。而一个新的目录创建时，其内部必定包括两个文件条目，其中一个文件名为“.”，表示当前文件目录，第二个是“..”，表示直接父级文件目录，且新目录的inode的i\_size字段必定是一个BLOCK的大小（和Linux的EXT2文件系统一致），即会占据整个数据块。

另外本项目系统不支持对EXT2文件系统存储格式的初始化，需要使用Linux



下的工具创建映像文件并进行格式化。如图4-3所示，首先使用dd命令创建空的数据文件inited. img（本项目设置其大小为4MB），然后使用mke2fs命令对该文件进行格式化。这样映像inited. img便可以作为Qemu所使用的闪存映像。

图 4-3 创建映像并格式化映像

### 4.1.2 存储格式数据结构操作

在进行文件操作时，首先EXT2存储区的某些数据结构需要进行读写，其次某些EXT2存储格式下数据结构间需要进行转换，为了实现一些基本数据结构的读写以及这些数据间的转换，在本项目里编写相应的数据结构操作函数来屏蔽Nand Flash的底层操作。

#### 1、相关读写宏定义

定义宏定义NAND\_READ\_EXT2STRUCT(x, nand\_offaddr)，该宏定义对应于函数get\_nand\_pages(& x, NAND\_EXT2\_OFFFSET+nand\_offaddr, sizeof(x))，该宏定义负责向EXT2文件存储区的nand\_offaddr偏移地址处写入结构体x（其中宏定义NAND\_EXT2\_OFFFSET是EXT2存储区在Nand Flash里的偏移地址）。

定义宏定义NAND\_WRITE\_EXT2STRUCT(x, nand\_offaddr)，该宏定义对应于函数put\_nand\_pages(& x, NAND\_EXT2\_OFFFSET+nand\_offaddr, sizeof(x))，该宏定义负责将结构体x写入EXT2文件存储区的nand\_offaddr偏移地址处。

另外宏定义NAND\_READ\_EXT2(addr, nand\_offaddr, size)和宏定义NAND\_WRITE\_EXT2(addr, nand\_offaddr, size)分别在内存地址addr和文件存储区地址nand\_offaddr之间读写size字节的数据。

#### 2、超级块读写函数

本项目里将EXT2存储格式的超级块保存在全局变量ext2\_sb里面，在本项目里只在EXT2存储系统验证函数get\_ext2\_struct（）里直接调用函数

NAND\_READ\_EXT2STRUCT(ext2\_sb,1024)对超级块读取一次。

编写函数store\_super\_block()负责将ext2\_sb存储回Nand Flash来保持超级块的一致性。

#### 3、块组描述符读写函数

编写函数load\_group\_desc(gd\_index)，该函数读取指定的块组描述符结构并返回该块组描述符的地址（这里为了省去内存的申请，块组描述符数据实际存储在全局变量ext2\_store\_gd里）。

编写函数store\_group\_desc(ext2\_group\_desc\*, gd\_index)将指定的块组描述符写回Nand Flash。

#### 4、inode节点转换函数

编写函数convert\_num2inode(inode, ext2\_inode\*)用于从节点号转换为Nand

Flash的EXT2存储区中相应的inode节点，其第一个参数是inode节点号，第二个参数是返回的inode结构体地址。

编写convert\_entry2inode(ext2\_dir\_entry\_2\*, ext2\_inode\*)用于从文件条目转换为相应的文件inode节点，其第一个参数是文件条目结构体的地址，第二个参数是返回的inode结构体地址。

编写函数store\_inode(ext2\_inode\*, inode)用于存储指定的数据块，第一个参数是inode结构体地址，其第二个参数是inode节点号。

#### 5、数据块读写函数

编写函数store\_block(mem\_base, block\_index)用于从指定内存地址读取一个数据块到Nand Flash的EXT2存储区中的指定数据块内，其第一个参数是内存地址，第二个参数是数据块块号。

编写函数load\_block(mem\_base, block\_index)用于从指定EXT2 数据块读取数据到指定的内存地址，其第一个参数是内存地址，第二个参数是数据块块号。

#### 6、位图加载函数

编写函数load\_block\_bitmap(ext2\_group\_desc\*)用于读取指定块组的数据块位图，其第一个参数是块组结构体的内存地址，该函数返回位图首地址。

编写函数load\_inode\_bitmap (ext2\_group\_desc\*)用于读取指定块组的inode位图，其第一个参数是块组结构体的内存地址，该函数返回位图首地址。

#### 7、inode申请释放函数

编写函数get\_free\_inode(t\_32\*p\_inode)用于申请空闲的inode节点并修改inode

位图，其第一个参数是要返回数据的inode号码地址，该函数返回值为所申请的

inode节点的EXT2存储区的偏移地址，当申请失败时返回0xFFFFFFFF。

编写函数release\_inode(inode)用于释放指定的inode节点，其第一个参数是

inode节点号。

#### 8、数据块申请释放函数

编写函数get\_free\_block(t\_32\*p\_block\_num)用于申请空闲的数据块并修改数据块位图，其第一个参数是要返回数据的数据块号码地址，该函数返回值为所申请的数据块的EXT2存储区的偏移地址，当申请失败时返回0xFFFFFFFF。

编写函数release\_block(block\_num)用于释放指定的数据块，其第一个参数是数据块块号。

#### 9、inode创建函数

编写函数create\_ordinary\_inode（）用于创建普通文件的inode结构，该函数调用get\_free\_inode函数获取空闲inode，并设置inode节点的i\_mode和i\_size（创建普通文件时不分配数据块，只有写入新创建的普通文件时才分配数据块）。

编写函数create\_dir\_inode(fa\_inode)用于创建目录文件的inode结构，其第一个参数是父目录的inode号，该函数依次调用get\_free\_inode函数和get\_free\_block函数来获取空闲inode和空闲数据块，并在申请的数据块里设置自身目录条目和父节点条目以及设置inode节点的i\_mode、i\_block[0]和i\_size属性。

### 4.1.3 本文件系统的操作流程

本小节介绍本项目中所实现的文件系统的操作流程，包括文件系统格式的验证、文件内容块的多级读取。

当内核主程序完成中断初始化和MMU虚拟映射后正式进入文件系统操作处理，内核程序跳到enter\_charfs(p\_cmd\_line)，该函数调用get\_ext2\_struct（）来验证EXT2格式的有效性。get\_ext2\_struct的代码概况如下，它首先调用NAND FLASH读取函数，读取EXT2超级块，并验证超级块的s\_magic字段是否为0xEF53，如果不是该函数返回-1 并且内核进入死循环，如果验证成功则统计块组描述符的个数，并读取第一个块组描述符，最后读取根目录的inode结构（根目录的inode号必定为2）。

…………

NAND\_READ\_EXT2STRUCT( ext2\_sb,1024); //读取EXT2超级块并储存到

ext2\_sb变量里，即文件映像的前1024个字节

if(ext2\_sb. s\_magic! =0xEF53) //验证魔数是否为0xEF53 return -1;

block\_size=1< <(ext2\_sb. s\_log\_block\_size+10); //从超级块里获得数据块的大小

ext2\_gdt\_base=1024/block\_size \* block\_size + block\_size; //计算组描述符的起始

位置

ext2\_gdt\_num=(ext2\_sb. s\_blocks\_count-ext2\_sb. s\_first\_data\_block+

( ext2\_sb. s\_blocks\_per\_group-1)) /ext2\_sb. s\_blocks\_per\_group; //计算组描述符的数目NAND\_READ\_EXT2STRUCT( ext2\_gd, ext2\_gdt\_base); //读取得到第一个组

描述符

NAND\_READ\_EXT2STRUCT(ext2\_root\_ino, block\_size\* ext2\_gd. bg\_inode\_table+sizeof(ext2\_root\_ino)); //读取得到根目录的inode 节点

………… ext2\_curr\_dir\_ino\_num=2; //EXT2格式默认根目录的inode号为 2

当要对某个文件进行操作时首先寻找目标文件的相关数据结构，编写get\_files(file\_path, t\_32\*, struct ext2\_inode\*, struct ext2\_dir\_entry\_2\*)来获取目标文件所在目录的inode号、目标文件所在目录的inode节点实体数据以及目标文件条目。该函数首先判断从当前目录所在路径开始还是从根目录开始（ 通过判断

file\_path的首字符是否为‘/’决定）；然后调用get\_ext2\_filename来获取下一级文件名，如果下一级文件名有误（超出255个字节），则get\_ext2\_filename返回失败，否则调用locate\_specific\_file来搜索指定inode的文件（locate\_specific\_file使用回调函数find\_dir\_entry搜索filename来搜索目标文件，目标文件名保存在全局变量

filename 里），如果搜索不到目标文件则返回失败，否则返回目录条目数据体的地址；最后判断是否是路径的最后一个文件，如果是则拷贝相应的条目和inode数据并返回，否则判断该文件是否是目录（包括目录的链接），如果是则进行数据转换并作为下一级搜索目录，否则返回失败。

当需要操作（读取/写入）某个具体文件时，需要使用不同的操作函数，本EXT2文件系统提供“解析函数+操作函数”的框架结构进行文件操作。该框架的含义为：使用解析函数来读取文件的数据并把相应的数据（包括四级数据块索引）提供给 操作函数；操作函数通过操作相应的数据并回写到EXT2文件系统里，最后设置某些全局变量或者指针或返回值来通知解析函数做出相应的动作。实际上操作函 数是一个回调函数，该函数使用独立的定义，解析函数不需要知道回调函数的具 体功能，但需要设置某些必要的全局变量。

每个解析操作框架的大体结构主要是：首先解析函数计算出目标数据块的值，并得到对应的inode的i\_block数组的索引值及其后续数据块索引值，从指定目标数据块开始复制数据块以及数据索引块到指定区域，并根据操作函数的返回值来 进行下一步操作，包括读取各级数据索引块和数据块。



二级

二级

三级

block-size/4

个索引指针

block-size/4

个索引指针

block-size/4

个索引指针

四级

一个

block-size

的大小

block-size/4

个索引指针

block-size/4

个索引指针

block-size/4

个索引指针

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

一级

一级

**i[0] i[1] i[2] i[3]**

图 4-4 解析+操作框架的数据读取示意 1

p\_i\_blocks block\_store\_addr

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
| 11 |
| 12 |
| 13 |
| 14 |

|  |
| --- |
| block-size  （正文存放空间） |
| block-size  （额外存放空间） |
| block-size  （一级索引块，直接指向数据块） |
| block-size  （二级索引块） |
| block-size  （三级索引块） |

p\_ib\_lvl1

p\_ib\_lvl2

p\_ib\_lvl3

图 4-5 解析+操作框架的数据读取示意 2

解析函数和操作函数使用i[4]数组来存放四级数据块索引，其中i[0]为指向

i\_block数组的索引值，而i[1]、i[2]、i[3]依次为不同层级的不同数据块索引（如图4-4所示）。然后再分出五个数据块大小的连续空间，且block\_store\_addr指针指向第一个数据块，该数据块保存的是最终的文件有效数据而不是数据索引块，然后

p\_ib\_lvl1、p\_ib\_lvl2、p\_ib\_lvl3 依次指向一级二级三级数据索引块的首地址（如图

4-5所示）。

#### 1、本系统主要实现了以下两个文件解析函数

（1）intepret\_directory函数

该解析函数的功能是负责读取文件的数据块并调用回调函数处理该数据块的数据，该回调函数只能读取和检查数据块的内容，回调函数的返回值表示已经处理的数据的字节数， 如果返回的数值为文件剩下的字节数则函数

intepret\_directory 返回，否则继续读取后面的数据块给回调函数处理。

函数intepret\_directory 包括三个参数：p\_i\_blocks、file\_size 以及

（\*func）(t\_8\*, t\_32, t\_32\*)，其中第一个参数p\_i\_blocks是文件inode节点的四级数据块索引，第二个参数file\_size是文件的大小，第三个参数func是回调函数的函数指针，该回调函数也包含三个参数：block\_addr、file\_size、size\_left，其中block\_addr是文件数据块的缓存地址、file\_size是文件的大小（实际上是该数据块后面的字节数）、size\_left 是数据块块内偏移（保留该变量用于处理多个数据块关联的数据处理）。

（2）modify\_inode\_content函数

该解析函数的功能是负责读取文件的数据块并调用回调函数处理该数据块的数据，该回调函数可以修改数据块的内容并保存回闪存，回调函数的返回值表示已经处理的数据的字节数，如果返回的数值为文件剩下的字节数则函数

modify\_inode\_content 返回，否则继续读取后面的数据块供回调函数处理。

函数modify\_inode\_content 包括三个参数：p\_i\_blocks、file\_size以及

（\*func）(t\_32\*, t\_32, t\_32, t\_32, t\_32\*)，其中第一个参数p\_i\_blocks是文件inode节点的四级数据块索引，第二个参数file\_size是文件的大小，第三个参数func是回调函数的函数指针，该回调函数包含五个参数：i、block\_num、block\_addr、

file\_size、size\_left，其中i是四级数据块索引的索引数组（4个元素的数组，存储的是索引的索引）、block\_num是当前处理的数据块块号、block\_addr 是文件数据块的缓存地址、file\_size 是文件的大小、size\_left 是数据块块内偏移。

#### 2、本系统实现的操作函数（回调函数）举例

（1）locate\_specific\_file 函数使用“intepret\_directory+find\_dir\_entry"的形式来搜索目标文件。回调函数find\_dir\_entry根据目录条目的排布，逐个匹配条目的名字和filename全局变量的值。

（2）list\_entrys函数使用“intepret\_directory+ show\_dir\_entrys”的形式来列出指定目录下面的文件。回调函数show\_dir\_entrys根据目录条目的排布，逐个显示目录的名字。

（3）mov 函数使用“intepret\_directory+get\_father\_inode"的形式来获取直接父节点。回调函数get\_father\_inode 通过匹配”..“文件名来获取父节点

文件inode号并设置全局变量fa\_inode。

（4）delete\_ordi\_inode函数使用“modify\_inode\_content+

release\_ordi\_inode\_blocks“的形式来释放指定inode节点的所有数据块。回调函数release\_ordi\_inode\_blocks通过读取四级数据块索引来释放所有已申请的数据块。

（5）delete\_dir\_inode函数使用了“modify\_inode\_content+delete\_all\_entrys"的形式来释放所有目录条目。回调函数delete\_all\_entrys获取到一条文件条目结构后调用delete\_entry\_content来删除子文件，如果该文件是目录则delete\_entry\_content递归调用delete\_dir\_inode删除该目录，如果是普通文件则调用delete\_ordi\_inode删除。

（6）add\_dir\_entry 函数使用了“modify\_inode\_content+add\_specific\_entry”的形式来添加文件条目。回调函数add\_specific\_entry将文件条目放置在一个空闲的位置或申请一个新的数据块存放该条目。

（7）read\_file\_block\_to\_user函数使用了“intepret\_directory+

get\_file\_block\_to\_user“的形式来读取目标文件指定的数据块到用户空间。回调函数get\_file\_block\_to\_user负责判断数据块块号并将数据拷贝到用户空间。

（8）read\_ordinary\_file函数使用“intepret\_directory+get\_file\_content"的形式来显示普通文件的所有字符。回调函数get\_file\_content 负责 将

intepret\_directory 函数提供的数据块的内容输出到串口。

## 4.2 支持的命令操作

本节将介绍本系统所支持的用户操作命令，由于项目时间有限，暂时只实现了部分常用的文件系统操作命令。

### 4.2.1 终端处理模块

和Linux下的命令为可执行文件不同，本系统只提供字符串类型的命令操作。本系统文件操作支持以下命令：“reset”，“restore-ps”，“kill-ps”，“poweroff”，“help”，“mkdir”，“mkfile”，“ls”，“cd”，“del”，“run”，“cat”，“mov”，“link”，“echo”，“close-dev”。 他们依次表示：重启、恢复用户进程、杀死用户进程、关机、帮助、创建目录、 创建普通文件、列出目录文件、进入指定目录、删除文件、运行ELF文件、输出文件内容、移动文件、链接文件、输入字符串到文件、关闭设备文件。另外系统 没有提供例如mknod 的命令来创建设备文件，设备文件只能由系统内核检查注册

的设备驱动来创建。



图 4-6 终端界面操作截图

终端命令主要分为：不带参数的、带一个参数的、带两个参数的；还可以分为：文件必须存在的和文件不存在的；而如果使用带两个参数的命令，则文件名为第一个参数，且该文件必须存在。

用户终端的输入命令使用同一个字符缓冲区来存放（该缓冲区为1024\*0x40个字节），包括命令以及所有参数，且本终端程序不支持转义字符，每层目录文件使用若干个“/”来划分，每个命令和参数使用空格或者回车符来隔开。终端控制程序检查命令的操作流程如下：当用户输入命令字段结束时，判断该字符串是否是系统提供的命令字符串，如果不是则清空缓冲区和输入显示区，否则返回命令字符串的索引号，然后判断该索引对应的命令含有多少个参数，并接受相应个数的参数，并判断参数指定的文件是否存在以及命令是否需要存在的文件，同时设置相应的目录inode以及文件条目结构，最后如果输入条件与命令要求一致则跳入相应的处理程序。

### 4.2.2 创建文件操作

本系统提供三种创建文件的操作命令：使用mkdir创建指定路径的文件目录、使用mkfile创建指定路径的普通文件、使用link创建指定路径的链接文件。其中链接文件类型包括普通文件的链接（ORDF\_LINK）以及目录文件的链 接

（DIR\_LINK），可以通过链接文件来访问目标文件，包括ELF可执行文件，如果

目标文件被删除后链接文件也不可以被访问（通过检查inode表相应比特位是否已分配，如果释放后的inode节点又被重新分配为相同类型的文件则链接文件依然可以访问新文件）。

创建目录文件时调用函数mkdir(t\_8\*)来创建目录文件，该函数首先判断是从绝对路径开始创建还是当前目录开始创建，并使用get\_ext2\_filename获取下一级目录名字，并调用locate\_specific\_file函数查找指定文件，如果不存在便使用create\_dir\_entry创建新的目录（函数create\_dir\_entry调用create\_dir\_inode创建目录inode节点并初始化i\_block指向的数据块），否则判断该文件是否是一个目录，如果不是则终止操作，否则以该目录作为下一级搜索的目录。mkdir 操作可能会因为某个文件名过长或者该文件不是目录文件而停止操作，但是停止操作前所创建的新目录依然存在于文件系统中。创建普通文件时调用函数mk\_ordinary\_file(t\_8\*)，该函数和mkdir 的框架类似，不同之处是使用

create\_dir\_entry调用create\_ordinary\_inode创建普通文件inode节点。创建链接文件时调用函数link ( struct ext2\_dir\_entry\_2\*, t\_8\*)，该函数需要优先修改文件条目的数据类型为相应的链接类型，并使用add\_dir\_entry添加新的文件条目。文件创建流程如图4-7所示。

创建普通文件

输入文件路径

开始

为上级目录

创建普通文添加文件条目读取上级目录的

件的inode下一个数据块

下一个文件名

|  |  |
| --- | --- |
| 数据块 是  有空闲位置  否  是最后的数据块  是 | |
| 创建新数据块 |  |
|  |

最后一个文件名

是文件存在

创建目录

否

写入

文件条目

否文件存在 是

创建目录

否

是否目录

是

创建目录文

件的inode

否

返回

设置目录第一数据块

路径错误

图 4-7 文件创建流程图

### 4.2.3 文件移动和删除操作

本系统支持文件移动和删除命令：mov和del。其中mov命令可用于将文件移动到指定命令或者更改文件名等操作；del 用于删除普通文件或者文件夹。

执行mov命令时，首先检查第一个参数路径是否存在，如果存在则可以执行移动操作，然后调用mov( t\_32, ext2\_inode\*, ext2\_dir\_entry\_2\*, t\_8\*)，其中第一个

参数是目标文件所在的目录的inode号，第二个参数是所在目录的inode映像地址，第三个参数是目标文件的条目，而最后一个参数是目标文件路径。如果目标路径 是一个无效路径，即中间某个文件是一个非目录非目录链接文件或者名字过长则 移动失败；如果是一个不存在的路径则创建该路径；如果该路径最后一个文件不 存在，则被移动的文件重新命名，否则必须是一个目录并检查该目录下有没有和 被移动文件重名的文件。如果被移动的文件是一个目录文件，则不仅要进行删除 条目和添加条目操作，还需要修改被移动文件夹的数据块的父节点的inode号（并且在修改前判断目标文件是否是目标路径的父节点，如果是则移动失败）。文件移 动流程如图4-8所示。

文件存在

否

最后一个 是

文件名

是

是

否

是目录类

文件

否

目标目录是

目标路径的父节点

文件存在

目标文件 是

是目录

是

否

否

是

最后一个 是

文件名

该目录文件下

有相同名字的条目

否

否

是

在目标路径下

添加文件条目

返回错误

输入目标文件

在目标文件所在目录里移除目标条目

改变目标目录的父节点inode号

返回错误

返回成功

开始

输入迁移路径

下一级文件名

创建目录

转换为文件inode

否

图 4-8 文件移动流程图

执行del命令时，首先检查第一个参数路径是否存在，如果存在则可以执行移动操作，然后调用delete\_file (t\_32, ext2\_inode\*, ext2\_dir\_entry\_2\*)，其中第一个参数是目标文件所在的目录的inode号。而第二个参数是所在目录的inode映像地址，而第三个参数是目标文件的条目。如果目标文件和当前目录冲突（即目标文件是 当前文件夹的父目录），则返回；否则调用delete\_entry\_content删除不同的文件（删除目录使用delete\_dir\_inode，删除普通文件使用delete\_ordi\_inode）。而函数delete\_dir\_inode使用回调函数delete\_all\_entrys删除所有条目，而delete\_all\_entrys再次使用delete\_entry\_content来递归删除文件。文件删除流程如图4-9所示。

删除目标文件

的内容

输入文件路径

目标文件存在

是目录 是

否

存在子

文件条目

是

否

是

是开始的

目标文件

否

目标是

父节点目录

否

是

是

删除失败

开始

删除失败

返回上一层

递归

读取下一条

子文件inode

释放文件所有

数据块

输入上级inode

以及文件条目

释放文件

inode

删除目标在父

目录下的条目

否

图 4-9 文件删除流程图

### 4.2.4 文件读写操作

本系统支持以下命令来操作文件：echo和cat。其中echo命令用于在非目录非目录链接文件后面添加新内容；cat用于显示非目录非目录链接文件的内容。echo和cat命令可以同时支持普通文件（包括链接文件）和设备文件的输入输出读写操作。

开始

否

数据写入

完成

文件存在

否

是

是

普通

判断 文件 文件类型

字符设备文件

设备注册表存 是在该设备

否

返回成功

输入文件路径

返回错误

写入下一段数据（不得超过数据块大小）

查找设备注册表

|  |  |
| --- | --- |
|  |  |
| 调用设备  写函数写入一个字符 | |

图4-10 echo命令流程图

是

否

是结束符

调用设备关闭函数

调用设备打开函数

执行echo命令时（程序流程如图4-10所示），首先检查文件的类型，如果文件类型为普通文件，则多次调用fill\_ordinary\_file向文件添加新的内容（每次调用fill\_ordinary\_file最多可写入block\_size个字节）；如果是设备文件则调用echo\_dev( filename, p\_str)来向指定的文件设备写入内容，echo\_dev函数通过名字匹配系统注册的设备（而不是设备句柄）；如果是其他文件类型则跳过。理论上每次

使用echo命令最多可支持0x40\*1024个字节的输入（因为为命令字符串存储区开辟0x40\*1024个字节的空间）。

执行cat命令时（程序流程如图4-11所示），类似于echo的流程，首先检查文件的类型，如果文件类型为普通文件，则调用read\_ordinary\_file，该函数使用回调函数get\_file\_content做字节的简单显示；如果是设备文件则调用cat\_dev (filename)，该函数调用不同设备的read 方法来读取数据；如果是其他文件类型则跳过。

开始

存在下一 是有效数据块

文件存在

否

否

是

是

否

普通 是结束符

判断 文件 文件类型

字符设备文件

设备注册表存 是在该设备

否

返回成功

输入文件路径

返回错误

调用设备关闭函数

显示该数据块的数据

查找设备注册表

调用设备打开函数

|  |  |
| --- | --- |
|  |  |
| 调用设备  读函数读取一个字符 | |

图 4-11 cat命令流程图

### 4.2.5 执行用户程序操作

本系统支持以下命令来操作用户程序：run、restore-ps和kill-ps。其中run命令用于启动用户程序；restore-ps用于恢复之前意外停止的未完应用程序；kill-ps用于杀死。

执行run命令时，使用load\_file函数将ELF文件的全部内容加载到连续的内存空间，并返回文件数据存储首地址。加载成功后调用load\_elf\_exefile来验证ELF文件的有效性并加载每个programme段到新分配的空间，并将\_user\_ttb位置的相应寄存器值设置为和ELF文件程序标识结构一致的内容。最后调用restore\_process来跳入用户程序段。

执行restore-ps命令时，使用restore\_process来恢复用户程序。该函数实际调用的是系统32号调用（参数为0），该系统调用读取\_user\_ttb位置的相应寄存器值，并切换系统分页表、修改PSR和恢复寄存器。

执行kill-ps命令时，使用kill\_process来停止用户程序。该函数实际调用的是系统32号调用（参数为1），该系统调用将用户模式的各个寄存器和系统分页表保存到\_user\_ttb 位置并恢复管理模式的各个寄存器的值和切换系统分页表。

### 4.2.6 其他操作

本系统还支持reset和poweroff操作，其中reset用于重启机器，该命令通过调用函数wdt\_enable来使能WDT并调用函数wdt\_reset\_en来使能重启信号来使得机器超时重启。而poweroff操作通过控制IIC来设置PCF50606硬件发送关机信号达到硬件关闭的效果。

## 4.3 设备文件模块的实现

### 4.3.1 虚拟设备的建立

系统进入enter\_charfs函数后，调用establish\_dev\_fs建立设备文件系统（定义在func/dev\_fs. c文件里）。因为本系统规定系统的设备文件位于/dev文件目录里（也可以在进入终端操作后将设备文件移动到其他位置），所以establish\_dev\_fs函数首先创建/dev目录，然后依次搜索\_schardev到\_echardev区段所注册的设备结构体指针（\_schardev和\_echardev的定义位于dev\_array. lds里，该链接文件还定义了系统的入口地址），并在dev目录里建立相应的设备文件。设备文件的inode号一律为0xFFFFFFFF，因为系统不通过inode号来访问设备，而是将设备文件名和设备结构的name属性进行匹配从而访问设备。

每个字符设备通过建立char\_dev结构体并设置各个字段的值来初始化设备驱动。该结构体定义在func/include/dev\_fs. h里，该结构体包含若干个字段，包括设备号码（\_schardev 区中的索引）、空闲状态、打开状态、设备缓冲区、设备缓冲区大小、设备类型、设备名字、初始化函数、打开函数、关闭函数、读函数、写函数。创建设备结构体时必须制定初始化函数，并由初始化函数进行其他属性和方法字段的初始化。下面是char\_dev结构体的定义：

Struct char\_dev {

t\_32 dev\_no；//设备号，即该设备在设备注册表\_schardev的索引 t\_32 dev\_busy； //设备忙状态，进入和退出读写函数时设置t\_32 dev\_opened； //设备打开状态，设备打开关闭函数负责设置t\_8\* dev\_buffer；//设备缓冲区地址

t\_32 dev\_bfsize; //设备缓冲区大小

t\_32 dev\_type; //设备类型，字符设备为3 char\* name; //设备的名字字符串地址

int (\*dev\_create)(t\_32)；//设备的初始化函数，负责设置其他成员变量

int (\*open)(void); //设备打开函数int (\*close)(void); //设备关闭函数int (\*write)(char); //设备写函数int (\*read)(char\*); //设备读函数

};

在每个设备驱动源文件里必须指定和实现该设备的初始化函数，并实现相应的打开函数、关闭函数、读函数、写函数。每个设备的功能可以不同，相应的设备方法函数也要相应的改变。当指定好设备的初始化函数后，establish\_dev\_fs 便会对各个设备调用其相应的初始化函数来对各个设备进行初始化。

### 4.3.2 为内核添加设备的方法

为了使每个设备间表现得相互独立，最好各自为他们创建不同的源文件，而每个源文件里必须将设备结构体实体的地址加入到. chardev 数据段，例如：volatile int var\_dev\_addr \_\_attribute\_\_((unused, section(". chardev")))= dev\_addr；。编译器便会自动初始化. chardev 数据段。而前提是设备驱动必须完成所有方法的编写，剩下的工作交给establish\_dev\_fs函数即可。

## 4.4 为文件系统添加两个字符设备

### 4.4.1 创建uart0设备

创建文件func/uart0. c，该文件里编写了关于uart0设备的设备访问代码。依次定义打开函数open（）、关闭函数close（）、写函数write(char)、读函数read(char\*)以及设备创建函数create(dev\_no)。设备创建函数依次初始化uart0设备的各个属性以及方法字段。

其中open（）通过判断dev\_opened和dev\_busy的值判断是否能为新的用户打开设备，因为设备只支持一个用户使用。close（）函数也进行和open相类似的操作，当用户程序没有关闭设备就退出后，为了系统可以继续访问该设备，提供了close-dev 命令来强制关闭设备。

读写函数write(char)和read(char\*)判断dev\_opened和dev\_busy的值是否有效，如果有效则打开，并分别调用系统函数put\_uart\_c(char)和get\_uart\_c()来读写串口的数据。

完成相关函数定义后，最后在chardev区段加入相应的uart0设备结构体地址变量volatile int var\_pipe \_\_attribute\_\_((section(". chardev"))) = & uart0;。

### 4.4.2 创建pipe虚拟设备

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| dev\_bfsize head\_index | | | | | | | | | | | |
| data | | data | data | data | null | null | null | data | data | data | |
|  |  | | | |  | | |  | | |  |

创建文件func/pipe. c，该文件里编写了关于pipe虚拟设备的设备访问代码。pipe是一个虚拟管道设备，驱动为它分配1024个字节的空间，pipe进行先进先出的数据操作。使用head\_index作为头指针和datanum统计有效数据的个数。pipe设备的存储区结构如图4-12所示。

datanum datanum

图4-12 终端界面操作截图

和创建uart0类似，同样要实现相应的打开函数open()、关闭函数close()、写函数write(char)、读函数read(char\*)以及初始化函数create(dev\_no)。

其中open()和close()与uart0对应的方法基本完全相同。而读写函数write(char)和read(char\*)判断dev\_opened和dev\_busy有效后分别读写队列头尾的数据，并修改head\_index和datanum的值。

完成相关函数定义后，最后在chardev区段加入相应的pipe设备结构体地址变量volatile int var\_pipe \_\_attribute\_\_((section(". chardev"))) = & pipe;。

## 4.5 本章小结

在这一章中，主要介绍了本项目文件系统的具体实现，介绍了EXT2存储格式 以及相应的关键数据结构、本项目文件的操作流程（包括EXT2格式的验证、多级 数据块的读取、相关数据结构的解析操作）、系统支持的文件操作命令和程序流程、以及设备文件部分的实现。

# 第五章 文件系统改进

## 5.1 基于两层多叉树的文件查找算法

### 5.1.1 基于两层多叉树的文件查找算法的设计与实现

因为EXT2文件存储格式下的每个目录文件都以顺序的结构来存储文件条目，当常用文件放在最后一个数据块的末尾时，使得用户打开文件的时间会相应的增 加。为了获得较好的文件搜索速度，可以从统计的角度将用户常用的文件条目放 置在的列表较前的位置，或者添加新的存储结构来加快搜索速度。



s1 s2 s3 s4 ......

p21 p22 p23 p24 ......

数据块

数据块

数据块

条目

条目

**.............**

条目

如果使用完全文件名字符索引节点值来进行查找，虽然查找速度可能会大大提高，但这样实际上是重新改用了另一种存储结构，则EXT2的i\_block的数据块索引结构便毫无意义。为了有效利用EXT2的已有的存储结构以及使得每个文件查找的效果较为平均，本项目通过添加多叉树索引，改进了EXT2存储格式，提出了基于两层多叉树的文件查找算法。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| p11 | p12 | p13 | p14 | ...... |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| p31 | p32 | p33 | p34 | ...... |

|  |
| --- |
| 数据块 |
| 条目 |
|  |

具有相同哈希值的文件条目的位置

图 5-1 多叉树查找索引结构

#### 1、基于多叉树的查找算法原理描述

为了支持本算法，需要创建多叉树索引，该索引结构如图5-1所示。实现该多叉树时，首先对每个目录创建一个数组，该数组存放该目录下所有文件的哈希值

或者文件的其他唯一映射标志（如首字符）；因为不同的文件可能会具有同一个哈希值，然后再创建下一层索引数组来索引到所有具有相同哈希值的条目。

为了实现该算法，需要另外开辟空间来存储查找树，使用inode节点结构体的保留字段l\_i\_reserved1作为查找树的根节点，其指向第一层数组所在的数据块块号，如果该数据块空间不够则最后一个数组元素指向下一个数据块，并使用保留字段l\_i\_reserved2记录哈希值的个数。

创建结构体ntree\_f\_index作为第一层数组的元素，该结构体记录文件名的哈希值hash\_v，以及具有该哈希值的第二层数组所在的数据块号block\_num（如果该

ntree\_f\_inde结构体是数组的最后的元素，则block\_num表示后续数组的数据块号）。struct ntree\_f\_index

{

t\_32 hash\_v; //文件的哈希值

t\_32 block\_num； //下一层所在的数据块号或者后续数组项所在块号

};

创建结构体ntree\_s\_index作为第二层数组的元素，该结构体记录文件条目在数据块里的偏移值以及数据块号。当确定文件的哈希值后获取该哈希值的所有

ntree\_s\_index，根据所有的ntree\_s\_index结构的block\_num以及offset的值获取相应的文件条目，最后比较文件名并返回搜索结果。

struct ntree\_s\_index

{

t\_32 offset； //条目在数据块里的偏移值

t\_32 block\_num； //条目所在的数据块号或者后续数组项所在块号

};

#### 2、编写基于两层多叉树的查找程序

（1）编写多叉树创建函数

因为Linux下的mke2fs命令不可以创建相应的多叉树，所以必须编写相应多叉树格式化算法。

编写格式化函数add\_ntree\_indexs负责为指定目录创建条目多叉树索引，当一级索引没有相应哈希值时使用函数add\_next\_indexs来申请新二级索引的数据块并返回数据块地址，使用函数add\_ntree\_s\_index 添加新的第二级索引。

（2）编写多叉树文件查找函数

编写查找函数ntree\_search，该函数在指定的一级索引数据块里对指定的哈希值进行查找，查找到相应的哈希值后使用函数ntree\_s\_search对二级索引进行查找得到条目所在的数据块号以及块内偏移地址。

### 5.1.2 测试两层多叉树文件查找算法程序

#### 1、对多叉树查找算法的理论分析

这里的理论分析是建立在一定假设的基础上的，接着对两层多叉树查找算法的复杂度进行计算，并和顺序查找的时间复杂度进行比较。

假设有n个文件，且文件的平均长度为h，且一共由k种字符组成，使用文件名字符串平均值作为哈希值，则有k个可能的哈希值，并假设每个哈希值都对应相同数量的m个文件，则有关系：n=k\*m。

对于顺序查找，即比较n个长度为h的字符串，其平均复杂度为：



对于多叉树查找的两层数组结构，计算两层需要进行比较的次数。对于第一层直接查找哈希值，则第一层复杂度为(k+1) /2，然后计算第二层的复杂度，即相当于对m个平均长度为h的字符串进行查找：



所以当k为1时两公式相差常数1，这时两个算法复杂度相等；当k趋向于n，m 趋向于1时，复杂度趋向于(n+1) /2+h。所以从理论上说，文件数量较多且哈希值的个数越多（k值越多），多叉树查找算法比顺序查找的优势越明显。

#### 2、编写测试代码比较两个算法的效率

因为除了字符比较次数外，其它和存储结构相关的地址和数据读取、计算、修改等操作都会影响查找效果，所以需要以实际的查找统计时间来进行比较。首先编程从字符集0~9、A~Z、a~z里生成指定长度的1000个随机文件名，并在Nand

Flash映像里生成相应的文件。然后对这些文件分别进行多叉树操作和顺序查找。这里使用WDT的WTCNT倒数寄存器的差值来充当查找速度。通过多次运行测试程序来对1000个文件进行查找，将总用时进行统计并计算平均用时，从而得到表5-1 的数据结果。

表5-1 文件查找用时统计表

|  | 文件数目 | 平均用时（WDT倒数次数） | 总用时（WDT倒数次数） |
| --- | --- | --- | --- |
| 顺序查找 | 1000 | 71.5788 | 70866~77647 |
| 多叉树查找 | 1000 | 48.7465 | 48113~52423 |

从表5-1看出，从查找总计时和平均查找计时来看，多叉树文件查找比起顺序查找具有较优的查找速度。然后再按照目录下文件条目的顺序，分别对各个文件名依次进行顺序查找以及多叉树查找，并记录两种方法下每个文件的查找速度，

最后在Matlab下将其绘制成图5-2（横坐标是文件的序号，纵坐标是搜索该文件前后

WTCNT倒数寄存器的差值，其中蓝色带点细线为多叉树查找的数据连线）。



图5-2 文件查找用时统计图

从图5-2看出，对于顺序查找来说，文件的查找用时随文件位置的后移而递增，而对于多叉树查找来说，通过多叉树进行哈希值散列后使得排在后面的条目也能较快的被查找出来，这是因为通过哈希后某些文件的索引放在了索引树较前的位 置从而提高文件搜索时间的均衡性。所以构建多叉索引树在某种程度上提高了文件的查找速度。

## 5.2 基于Nand Flash的数据块擦写均衡化的研究

### 5.2.1 Nand Flash擦写均衡化问题的引入

Nand Flash作为一种电子存储器件，是一种支持多次分块擦写的存储设备，理论上Nand Flash支持每块最大一百万次的擦写。当某块擦写次数过多或者制造控制不当等因素的影响可能导致某些数据块成为坏块，尤其是对于旧式的Nand Flash设备来说，出厂时带有坏块常常是不可避免的，而这些坏块的分布也是随机的，在实际应用中只能是通过软件的方法来发现坏块或者避免坏块的出现。

首先可以利用循环冗余校验ECC算法来比较数据块的内容和存储数据块校验码的值从而发现坏块；而对于数据块的擦写控制，可以通过软件的方法来实现数据块擦写的均衡化，这在Nand Flash使用的耐久性上有实际的应用。

本系统最开始使用了第一空闲块优先算法来分配空闲数据块，该算法的操作逻辑是这样的：首先检查超级块的s\_free\_blocks\_count属性是否为0，若为0则返回分配失败；否则搜索各个块组的数据块位图，且从第一个比特位开始搜索第一

个空闲数据块，并将该比特位索引返回。

第一空闲块优先算法的优点是分配速度快，且无需额外的记录存储空间，但是缺点是没有考虑数据块的使用情况，从而会使得Nand Flash的不同使用情况的空闲块得不到合理的分配，当某些数据块擦写过多时容易导致坏块的出现。

为了直观的看出第一空闲块优先算法的性能，需要编写测试函数，对第一空闲块优先算法进行测试。该测试程序由两部分组成，包括模拟文件操作部分以及页块擦写统计部分。

该测试方法的统计部分首先编写Nand Flash 页块统计表初始化函数

init\_statistics()，该函数根据超级块的s\_blocks\_count属性的值对Nand Flash页块

（512个字节）计数表初始化，并将计数函数go\_statistics(nand\_offaddr)直接嵌入到Nand Flash的页块写入函数put\_nand\_page(data\_memaddr, nand\_offaddr, size). 所以当文件系统调用Nand Flash写入函数时就会进行页块的擦写统计。

该测试方法的模拟文件操作部分需要编写函数test\_equalization（），该函数执行1000轮随机文件操作，每轮操作需要对1000个文件（文件名依次为0、1、2….1000）进行随机操作，包括创建文件、删除文件、写入文件等操作，每轮操作中当指定 文件不存在时便创建该文件，当该文件存在时则随机进行写入和删除操作，其中 进行写入操作的几率为3/4，删除操作的几率是1/4，之所以这样设置是为了让文件存储区的所有数据块都能够有机会进行擦除操作。另外因为WDT的WTCNT倒数寄存器在某个确定过程前后的差值是不确定的，所以本模拟函数使用某个循环 后该差值的奇偶性作为随机变量。

另外要指出的是本测试主要针对的是文件有效数据块的均衡化情况的数据统计，而对前面用于存储EXT2特殊数据结构的数据块的擦写情况不进行统计。该统计方法从第一个可用于存放文件内容的数据块开始统计。可通过以下公式：

block\_size\*ext2\_gd. bg\_inode\_table+sizeof(ext2\_inode) \*ext2\_sb. s\_inodes\_per\_group

（其中ext2\_gd是第一个块组描述符，ext2\_sb是ext2超级块）计算出第一个文件数据存储块的地址。而本系统中计算得到的第一个可用于存放文件内容的数据块块号为148。

最后还要指出的是由于本测试在创建文件时对根目录所使用的数据块执行了过多的操作，所以为了清楚了解算法的效果也要将根目录所使用的数据块的擦写统计数据忽略。最后实际的有效统计数据块个数大约为7870个Nand Flash页块（文件存储系统为4M字节，即8192个Nand Flash页块）。



图5-3 第一空闲块优先算法数据块擦写次数分布图

图5-3是其中一次随机模拟操作后测得的数据块擦写情况统计分布图，从图中可以看出第一空闲块优先算法对前面的数据块的使用频率较高，使得文件存储区最后面的若干数据块没有被使用，而中间的数据块也很少使用。所以使用该方法进行数据块的分配非常不合理。另外表5-2统计的是执行不同次数的随机模拟操作所得到的数据块擦写数据，可以看出随着模拟次数的增加，数据块擦写的次数随着增加，而标准偏差也同时增加，表明了数据块使用情况只会更加恶化。而最优的理想的数据块分配算法的标准偏差应该在0附近，所以使用标准偏差可以作为判断分配算法好与坏的标准之一。

表5-2 第一空闲块优先算法测试结果统计表

| 执行多少轮操作 | 标准偏差 | 平均擦写次数 | 总擦写次数 |
| --- | --- | --- | --- |
| 100(100000次) | 16.3774 | 7.8544 | 61814 |
| 200(200000次) | 32.6599 | 15.0525 | 118463 |
| 500(500000次) | 79.9777 | 39.3240 | 309480 |
| 1000(1000000次) | 162.0806 | 79.2845 | 623969 |
| 2000(2000000次) | 327.3206 | 139.9950 | 1101761 |

### 5.2.2 Nand Flash擦写均衡化算法设计和测试

为了提高数据块使用的均衡化程度，本文设计了两种数据块分配均衡化解决方案，一种是基于数据块位图的下一空闲块分配算法，另一种是基于链表统计的空闲块分配算法，然后对两种方法进行测试并统计相关的数据来对两种方法进行分析。

#### 1、基于数据块位图的下一空闲块分配算法

通过对第一空闲块优先算法的分析，可以发现后面的数据块没有得到有效的利用，即后面的数据块很有可能擦写次数会更低，即可以考虑从后面的位置开始查找空闲数据块。

下一空闲块分配算法的思路是调整数据块的申请方式，改为从后一空闲块开始分配，该方法使用一个额外的整型变量存储下一个数据块位图里开始查找空闲块的位置。该方法的实现函数为block\_bitmap\_equation(bitmap\_base, mem\_size)，其中bitmap\_base为位图的存储地址、mem\_size为位图的大小，block\_bitmap\_nindex变量记录搜索开始的位置，当成功获取一个空闲数据块块号后该变量加一并对ext2\_sb. s\_blocks\_count- ext2\_sb. s\_first\_data\_block求余。

图5-4 记录了使用本算法下各个被统计的数据块的擦写次数（执行1000轮随

机文件操作；依然使用上一节的文件操作模拟函数和统计函数）。和图5-3比较，可以看出本算法明显提高了数据块的均衡化程度。观察该图发现，该算法可以较为有效的将数据块擦写次数压制在某个值之下，而有些数据块擦写次数较低是因为在随机文件操作的作用下该数据块很少被释放。



图5-4 基于数据块位图的下一空闲块分配算法擦写次数分布图

表5-3 统计了使用基于数据块位图的下一空闲块分配算法下执行不同次数的

文件随机模拟操作所得到的数据块擦写数据。和表5-2对比，相同的随机操作次数下，数据块擦写次数相当，但标准差得到明显降低，且随着操作次数的增加标准差的值也保持在较低的水平。

表5-3 基于数据块位图的下一空闲块分配算法测试结果统计表

| 执行多少轮操作 | 标准偏差 | 平均擦写次数 | 总擦写次数 |
| --- | --- | --- | --- |
| 100(100000次) | 0.4825 | 8.7959 | 69224 |
| 200(200000次) | 0.8907 | 16.2695 | 128041 |
| 500(500000次) | 2.0114 | 38.0192 | 299211 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1000(1000000次) | 12.1711 | 85.2051 | 670564 |
| 2000(2000000次) | 18.6318 | 153.3779 | 1207084 |

#### 2、基于链表统计的空闲块分配算法

虽然基于数据块位图的下一空闲块分配算法可以达到较好的分配效果，但是该方法并不是最优的。因为该方法只是基于“下一空闲块有可能是使用次数最少的的空闲块”这一假设下的，而事实上并非如此，例如前面的某个数据块长期占有后终于被释放，则该数据块成为了使用最少的空闲块，而该算法却可能分配了后面的空闲块。



bitem\_table\_saddr

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 空闲 10 |  | 空闲 12 |  |  |
|  |  |  |  |  |
| 空闲 9 |  | 空闲 11 |  | 空闲 13 |
|  |  |  |  |  |
| 空闲 8 |  | 空闲 4 |  | 空闲 3 |
| 空闲 6 | | | | |
| 空闲 2 | | | | |
| 空闲 7 |  | 空闲 5 |  |  |
|  |  |  |  | 空闲 1 |



free\_bitem\_saddr

图5-5 数据块擦写次数统计表（示意图）

基于链表统计的空闲块分配算法的思想是统计各个数据块的擦写次数，并以 链表的形式按擦写次数从小到大链接各个空闲块的对应的统计链表项。该统计表 和链表的结构如图5-5所示，其中bitem\_table\_saddr是数据块擦写统计表项数组，

free\_bitem\_saddr是空闲块擦写统计链表，每个数据块擦写统计表项包括count 和

nitem\_addr两个属性，其中count成员记录该数据块的擦写次数，nitem\_addr存储下一个空闲数据块表项的地址（如果该数据块被占用则nitem\_addr为1），统计表项的数据结构定义如下：

struct block\_use

{

t\_32 count; //数据块的擦写次数

t\_32 nitem\_addr; //其值为1或为下一个空闲数据块表项的地址

};

为了实现基于链表统计的空闲块分配算法，首先实现初始化函数init\_link\_table，该函数建立数组bitem\_table\_saddr以及读取EXT2数据块位图信息来初始化空闲块擦写统计链表free\_bitem\_saddr；然后编写函数request\_free\_use用于从链表头返回擦写次数最少的空闲块块号（块号 为

free\_bitem\_saddr-bitem\_table\_saddr），如果没有空闲块则返回-1（free\_bitem\_saddr同时也为-1）；最后编写release\_free\_use（index）函数用于释放指定数据块，并根据该数据块的擦写次数按顺序将链表项插入链表。

图5-6 是使用本算法进行1000轮随机文件操作（依然使用相同的文件操作模

拟函数和统计函数）所得到的各数据块擦写次数统计图。和图5-3 相比，本算法也

能很好的提高数据块的使用的均衡化程度，和图5-4相比则没有特别明显的进一步改善。



图5-6 基于链表统计的空闲块分配算法擦写次数分布图

表5-4统计了使用基于链表统计的空闲块分配算法下执行不同次数的文件随机 模拟操作所得到的数据块擦写数据。和表5-3类似，对于较低次数的文件模拟操作，标准偏差同样可以压制在较低的水平，当模拟次数到达一定程度，标准差同样也 会增加得更快。

表5-4 基于链表统计的空闲块分配算法测试结果统计表

| 执行多少轮操作 | 标准偏差 | 平均擦写次数 | 总擦写次数 |
| --- | --- | --- | --- |
| 100(100000次) | 0.4228 | 5.2187 | 41071 |
| 200(200000次) | 0.4737 | 11.7437 | 92423 |
| 500(500000次) | 0.8597 | 31.4986 | 247894 |
| 1000(1000000次) | 3.3462 | 69.5545 | 547394 |
| 2000(2000000次) | 14.2247 | 143.9079 | 1132555 |

#### 3、空闲块分配算法总结

基于数据块位图的下一空闲块分配算法和基于链表统计的空闲块分配算法在 某种程度上提高了的数据块擦写次数的均衡化程度，但随着文件操作次数的增加，标准偏差也相应的出现不同程度的增长。

通过100至2000轮文件操作对两种算法所得到的数据块擦写平均次数和标准偏差统计绘制得到图5-7，可以发现标准偏差没有呈现某种函数规律并且相同的平均擦写次数下标准偏差也会相差较大，这两种算法的均衡化效果还依赖于具体的文件操作过程，即这两种算法不能够自适应于用户对文件的操作行为，也就是这两种算法不能很好的通过预测文件操作来进行数据块的分配。



图5-7 平均擦写次数和标准差对比图

通过前面的研究提出以下几点改进建议：

（1）考虑利用时钟中断，设计适合的文件数据回写算法，从而保持所有EXT2重要数据结构的内存缓冲区的内容和闪存的内容一致，以及减少闪存数据块的擦写次数；

（2）设计基于内存的文件缓冲区数据结构体系，用于保存最近使用的文件数据和经常使用的文件数据，以及减少闪存数据块的擦写次数；

（3）引入阀值的概念，即数据块擦写次数的最大值或危险值，也可以是需要进行数据块重新分配的警告值（阀值的大小可以使用经验值或测试统计值）。通过统计长期没被修改过的文件，选择其中一个最少使用的数据块并和达到阀值的数据块进行数据交换，则该达到阀值的数据块用于存储较少被改动的数据，从而减低该数据块被再次擦写的可能性。

## 5.3 本章小结

本章介绍了在文件查找和数据块擦写均衡化上所作出的算法改进，包括提出了基于两层多叉树的文件查找算法、基于数据块位图的下一空闲块分配算法、基于链表统计的空闲块分配算法，并对这些算法进行测试。

# 第六章 文件系统测试和验证

## 6.1 测试设备控制程序

### 6.1.1 测试串口程序

这里同时使用手动和编程的方法来测试串口程序，分别测试串口程序对键盘按键读取的正确性和对字符读写的正确性。

#### 1、手动测试串口程序对按键响应的正确性

手动从键盘敲入需要测试的按键，编写串口测试程序读取串口输入的字符并转译为十六进制字符串值输出以及直接输出输入字符，最后统计比较输入按键和输出的字符以及其十六进制字符的一致性。

表 6-1 串口程序接收键盘数据测试结果表

| 按键 | 输出结果 |
| --- | --- |
| 0~9 数字键 | 1、回显字符一致；2、十六进制值和 ASCII 码一致 |
| 大小写字母键 | 1、回显字符一致；2、十六进制值和 ASCII 码一致 |
| 可显示符号键 | 1、回显字符一致；2、十六进制值和 ASCII 码一致（包括空格键） |
| 回车换行键 | 1、回显时只回车不换行；2、十六进制值为 0x0D |
| Backspace 键 | 1、回显时只输出空格而不是删除操作；2、十六进制值为 0x7F |
| Fn 功能键 | 1、回显时输出某字符序列；2、F2~F9, F12 十六进制值分别为：  0x1B0x4F0x51、0x1B0x4F0x52、0x1B0x4F0x53、0x1B0x5B0x310x350x7E、  0x1B0x5B0x310x370x7E、0x1B0x5B0x310x380x7E、0x1B0x5B0x310x390x7E、  0x1B0x5B0x320x300x7E、0x1B0x5B0x320x340x7E  （F1、F10、F11 作为 Ubuntu 下的功能热键而不能向 Qemu 输入） |
| Ctrl 功能键 | 虚拟机无法接收（作为 Qemu 的功能键，被 Qemu 屏蔽而不发送给虚拟机） |
| 方向功能键 | 1、回显时可以控制光标移动；2、十六进制值分别为，上：0x1B0x5B0x41，  下：0x1B0x5B0x42，左：0x1B0x5B0x44，右：0x1B0x5B0x43 |
| Del 键 | 1、回显时无任何动作；2、十六进制值为 0x1B0x5B0x330x7E |

结论：（1）对于可显示的按键（数字键、字母键、符号键），串口程序可以准确读写；（2）对于功能键的输入，串口程序可以准确读取其字符序列，此外上层终端程序还需要进行转译操作，当读取到特定的功能键时需要转译为相应的终端控制字符序列（例如回车键转换为换行符、BackSpace键转换为\x1B\x5B\x44\x20）。

#### 2、编程串口测试程序读写字符的正确性

Qemu 程序：-serial pipe:/tmp/serial

串口重定位

Neo1973

虚拟机

串口

测 试 程序：负责和管道文件通信

serial.in 文件

serial.out 文件

因为按键的字符值有限，所以无法通过键盘将256个字符都输入到Qemu所模拟的串口，需要通过特殊的方法向该串口输入所有字符进行测试。这里使用Qemu 的重定位功能向该串口发送字符（如图6-1 所示），使用管道文件作为重定位的媒介（这里不能使用虚拟终端设备文件作为字符发送的媒介，因为发送的字符字节直接反映在终端屏幕界面而不被Qemu捕获）。

图 6-1 测试方法结构关系图

编写Linux端的测试程序向/tmp/serial. in和/tmp/serial. out文件进行字符输入输出，同时添加相应的Neo1973下的串口测试程序读取和回写字符，Linux端的测试程序通过比较发送和接收到的字符来判断字符的一致性。测试结果如表6-2 所示。

表 6-2 串口收发数据统计表

| 发送端 | 接收端 | 数据值 | 数据个数 | 结果 |
| --- | --- | --- | --- | --- |
| Ubuntu 宿主机 | Neo1973 虚拟机 | 0~255 | 256 | 数据完全一致 |
| Neo1973 虚拟机 | Ubuntu 宿主机 | 0~255 | 256 | 数据完全一致 |

测试结果：字符值0~255可以被Linux端的测试程序正确匹配，即发送出去的字符和接收到的字符一致。说明：（1）Neo1973下的串口读写程序正确；（2）Qemu的模拟功能和重定向功能正确。

### 6.1.2 测试Nand Flash程序

编写Nand Flash测试程序来测试Nand Flash读写程序的正确性，首先以一个文本文件作为模拟器的Nand Flash镜像，编写测试程序从Nand Flash中读取字符并回显，并和文本文件的内容作比较；然后编写测试程序将64M存储区置一以及清零，并用二进制工具打开修改后的映像文件来检查是否成功修改。测试结果如表6-3所示。

表 6-3 Nand Flash程序测试结果表

| 测试操作 | 结果 |
| --- | --- |
| 编写测试程序，以文本文件作为 Nand Flash 映像并读取回显 | 显示内容和文本一致 |
| 编写测试程序，把 Nand Flash 所有比特位设置为 1 | 映像修改成功 |
| 编写测试程序，把 Nand Flash 所有比特位设置为 0 | 映像修改成功 |

表6-3测试结果表明：1、测试程序可以从Nand Flash中读取数据且回显的所有字符和充当该Nand Flash映像的文本文件一致；2、测试程序可以正确将Nand Flash映像所有存储空间全部清零和置一。说明：1、Nand Flash读写程序的地址翻译和读写流程正确；2、Qemu的Nand Flash模拟功能正确。

## 6.2 测试文件系统基本功能

### 6.2.1 演示文件指令操作

本节简要介绍文件操作的手动测试过程。图6-2分别展示了cd、mkdir、mkfile、link、ls、echo、cat和del命令功能的验证。



图 6-2 验证文件操作

从图6-2中看出，首先使用ls命令查看dir目录，因为此时没有存在dir文件夹，所以使用mkdir创建dir目录（同时创建sub\_dir子目录），接着使用mkfile命令创建普通文件txt，并使用link命令创建txt文件的链接文件txt\_link和dir目录的链接文件dir\_link。然后使用ls命令查看所有创建的文件。最后使用echo和cat修改和读取txt文件和txt文件的链接文件。

通过以上验证不仅可以看出文件操作的可用性，因为文件系统依赖于其他的底层模块（例如串口模块、Nand Flash管理模块、内存管理模块等），所以同时表明了其他底层模块的可用性。

实际的文件操作应该是更加复杂的，而不是通过一两次文件操作就可以验证所有功能的正确性。在开发调试过程中通过配合代码逻辑检查以及复杂的文件路径设置和复杂的命令组合操作已经发现了多个功能错误并将其纠正。

同时在这里只简单展示文件操作的可用性，实际的功能验证工作是更加严密的，其中包括inode位图、块内图相应比特位的释放和占用的验证，而这些底部数据的一致性需要通过查看二进制映像以及对比前后位图比特位设置和编写调试程序来保证，该工作已经在编程过程中为之保证。

### 6.2.2 测试指令操作

本项目通过多次测试本文件系统提供的指令操作，发现了多个代码错误。主要问题有：1、由于文件路径定位错误，而导致的文件inode结构读取错误；2、多个文件操作函数共用相同的数据结构实体，而导致数据被覆盖的错误。通过对代码跟踪和调试，修正了已发现的代码错误，并将测试过程总结为表 6-4。

表 6-4 指令测试统计表

| 指令 | 指令功能 | 测试过程和结果 |
| --- | --- | --- |
| reset | 重启系统 | 正确 |
| restore-ps | 恢复应用程序 | 修正了用户堆栈指针没重设的错误 |
| kill-ps | 杀死应用程序 | 正确 |
| poweroff | 关闭系统 | 正确 |
| mkdir | 创建文件目录 | 修正了文件路径定位错误 |
| mkfile | 创建普通文件 | 修正了文件路径定位错误 |
| ls | 显示目录下的文件 | 修正了文件路径定位错误 |
| cd | 进入指定目录路径下 | 修正了文件路径定位错误 |
| del | 删除指定文件 | 修正了文件路径定位错误，以及释放位图和数据块 |
| run | 运行应用程序 | 修正了用户堆栈指针没重设的错误 |
| cat | 显示文件内容 | 正确 |
| mov | 移动文件到指定路径下 | 修正了文件路径定位错误 |
| link | 创建链接文件 | 修正了文件路径定位错误 |
| echo | 为文件添加数据 | 正确 |
| close-dev | 强制关闭文件设备 | 正确 |

### 6.2.3 测试文件读取的一致性

#### 1、Ubuntu 下创建文件后对比文件的一致性

在本测试里，首先在挂载了initrd. img镜像的文件夹下添加若干个文件（目录或者普通文件），然后以initrd. img作为Nand Flash的映像在本嵌入式环境下读取根路径下所有文件，最后统计和对比两个系统下读取到的文件数目、类型、名字是否一致。

如表6-5所示，创建了数量较多的目录文件和普通文件，并且在本嵌入式系统下和Linux环境下都读取到相同数量和相同名字的文件，证明本系统可以准确读取EXT2格式下的文件。

表 6-5 Ubuntu下创建文件的一致性对比统计表

| 文件总数 | 目录文件个数 | 普通文件个数 | 文件名是否一致 | 文件个数是否一致 |
| --- | --- | --- | --- | --- |
| 1000 | 1000 | 0 | 一致 | 一致 |
| 1000 | 0 | 1000 | 一致 | 一致 |
| 1000 | 500 | 500 | 一致 | 一致 |

#### 2、虚拟机下删除文件后对比文件的一致性

在本嵌入式系统下删除initrd. img镜像下的指定数目的文件（这些文件在Ubuntu下创建），并将initrd. img镜像在Ubuntu下挂载打开，最后统计对比两个系统下读取到的文件数目、类型、名字是否一致。

如表6-6所示，在本嵌入式系统下删除了若干个目录文件或者普通文件，而在Ubuntu 环境下打开经过操作后的映像发现只存在没有删除的文件，且剩下文件在数量和名字上一致，证明本系统可以准确删除EXT2 格式下的文件。

表 6-6 虚拟机下删除文件的一致性对比统计表

| 删除前目录  文件个数 | 删除后普通  文件个数 | 删除后目录  文件个数 | 删除后普通  文件个数 | 文件名是否  一致 | 文件个数是否  一致 |
| --- | --- | --- | --- | --- | --- |
| 1000 | 0 | 500 | 0 | 一致 | 一致 |
| 0 | 1000 | 0 | 500 | 一致 | 一致 |
| 500 | 500 | 250 | 250 | 一致 | 一致 |

#### 3、虚拟机下创建文件后对比文件的一致性

以initrd. img作为Nand Flash映像，在本嵌入式系统下创建指定数目的文件，然后在Ubuntu下挂载initrd. img，最后统计对比两个系统下读取到的文件数目、类型、名字是否一致。

如表6-7 所示，在本嵌入式系统下创建了若干个目录文件或者普通文件，而在

Ubuntu 环境下打开经过操作后的映像发现文件在数量和名字上一致，证明本系统

可以准确创建EXT2格式下的文件。

表 6-7 虚拟机下创建文件的一致性对比统计表

| 虚拟机下目录文件个数 | 虚拟机下普通文件个数 | Ubuntu 下目录文件个数 | Ubuntu 下普通文件个数 | 测试结果 |
| --- | --- | --- | --- | --- |
| 1000 | 0 | 1000 | 0 | 文件名、格式、类型一致  文件皆可正常访问和修改 |
| 0 | 1000 | 0 | 1000 | 文件名、格式、类型一致文件皆可正常访问和修改 |
| 500 | 500 | 500 | 500 | 文件名、格式、类型一致  文件皆可正常访问和修改 |

#### 4、一致性总结

为了实现和Linux下文件读写的一致性，本系统至少需要保证以下几点：（1）目录大小为数据块大小的整数倍；（2）文件条目大小是4 的整数倍；（3）文件的

inode节点的i\_blocks为占用512字节数据块的个数；（4）目录文件inode节点的

i\_mode为0x41C0，普通文件的i\_mode为0x81A4；（5）被删除的文件条目的inode

属性为0。

### 6.2.4 测试文件操作的性能

这一节通过编写批量文件操作函数来测试文件操作函数的性能。这里使用

WDT的WTCNT倒数寄存器的差值来充当文件操作的用时单位，并且根据公式t\_watchdog=1/(PCLK/(Prescaler value+1) /Division\_factor)来换算为以秒为单位的平均用时（在本系统中使用默认值，Prescaler value=128, Division\_factor=8, 慢模式下PCLK=12MHz, 即t\_watchdog=0.000172秒），最后总结为表6-8. 在测试文件读写操作时，向指定文件写入和读出1024个字节，测得写入速度为39.8795KB/s，读取速度为69.8314 KB/s。

表 6-8 文件操作用时统计表

| 测试项目 | 文件个数 | 平均用时  （WTCNT 寄存器差值） | 总用时  （WTCNT 寄存器差值） | 平均用时  （单位：秒） |
| --- | --- | --- | --- | --- |
| 创建文件 | 1000 | 113.827 | 111098~115710 | 0.019578244 |
| 写入文件 | 1000 | 145.788 | 141985~159693 | 0.025075536 |
| 读取文件 | 1000 | 83.257 | 80664~88779 | 0.014320204 |
| 删除文件 | 1000 | 177.139 | 174067~179514 | 0.030467908 |
| 查找文件 | 1000 | 54.954 | 30863~62620 | 0.009506977 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| （哈希） |  |  |  |  |

## 6.3 验证设备文件

### 6.3.1 演示设备文件的操作

本节介绍设备文件的操作过程。通过查看根文件夹下的dev目录即可查看系统所创建的所有设备文件，如图6-3所示，pipe和uart0是系统所创建的两个设备文件，表明系统所注册的设备已经创建成功。

接着测试uart0设备的功能，使用cat命令来回显uart0设备所读取的数据。如图6-3所示，语句“uart0 open”由uart0的open方法所输出，语句“User word: This is

uart0!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!“为用户键盘输入的语句，最后输入含0x1B开头的按键（例如上下左右键或F1-F12功能键）即可关闭设备uart0，图6-4的语句”uart0

close"由uart0的close方法所输出。

最后测试pipe设备的功能，因为和uart0的设备实现的功能有些差别，pipe实际上是一种存储设备，需要预先为该设备写入内容最后读出该内容。使用echo将字符串“this is the content sended to pipe device”写入pipe设备，最后使用cat读取

pipe设备中的所有数据。如图6-4所示，cat命令可以正确读取pipe设备里的数据，表明pipe设备的read和write方法正确（其中“pipe open”和“pipe close”分别由相应的open和close方法输出）。



图6-3 演示访问uart0设备文件



图6-4 演示访问pipe设备文件

### 6.3.2 测试设备文件操作函数

这一节对相应的设备文件操作函数进行测试，包括设备文件创建函数、设备文件读写函数以及设备结构体的操作函数，并统计测试结果，如表6-9 所示。

表 6-9 设备文件操作函数测试统计表

| 设备操作 | 对应的操作函数 | 测试结果 |
| --- | --- | --- |
| 创建设备文件 | establish\_dev\_fs、mkdev\_node | 成功创建/dev/uart0、/dev/pipe 文件 |
| 读取设备文件 | Cat\_dev( char \*dev\_name ) | 成功读取/dev/uart0、/dev/pipe 文件 |
| 写入设备文件 | Echo\_dev(char \*dev\_name,char\* p\_cont) | 成功写入/dev/uart0、/dev/pipe 文件 |
| uart0 设备文件的操作函数 | uart0 设备创建函数 create(t\_32 dev\_no) | 成功设置 uart0 设备结构体 |
| uart0 设备打开函数 open() | 成功设置结构体 dev\_opened 属性 |
| uart0 设备写入函数 write(char c) | 成功写入 UTXH0 寄存器 |
| uart0 设备读取函数 read(char\* pc) | 成功读取 URXH0 寄存器 |
| uart0 设备打开函数 close() | 成功设置结构体 dev\_opened 属性 |
| pipe 设备文件的操作函数 | pipe 设备创建函数 create(t\_32 dev\_no) | 成功设置 pipe 设备结构体 |
| pipe 设备打开函数 open() | 成功设置结构体 dev\_opened 属性 |
| pipe 设备写入函数 write(char c) | 成功写入 pipe 设备缓冲区 |
| pipe 设备读取函数 read(char\* pc) | 成功读取 pipe 设备缓冲区 |
| pipe 设备打开函数 close() | 成功设置结构体 dev\_opened 属性 |

## 6.4 测试运行应用程序

### 6.4.1 演示运行应用程序

为了保证文件操作、设备操作、文件系统调用、应用程序解析、MMU 保护的实现等多方面的可用性和正确性，本项目还编写了若干个测试程序来加以验证（相关应用程序代码位于Application文件夹下）。

#### 1、编写和运行文件数据读取程序

本项目提供示例程序read\_file. c来读取指定路径下的文件（包括普通文件以及设备文件，当使用相对文件路径时是以elf文件所在的目录为起始目录）。程序大体流程为：（1）申请一个空间来存放文件名；（2）打开指定文件，判断是否打开成功，不成功则终止程序；（3）指定文件读取偏移位置，根据返回字节数读取缓冲区的数据；（4）关闭文件并终止程序。该程序核心语句如下：

char\* addr=malloc(1024); //申请1024个字节的空间

if((int) addr==-1) kill(); //申请失败，结束本程序

readstr(addr); //调用串口系统调用读取文件名unsigned char\* data\_addr ;

int handle =fileopen(addr ); //打开文件

if(handle==-1) kill();//申请失败，结束本程序

data\_addr=filebuffer(handle);//获取文件的缓冲区while(1){

size=fileread(handle, k++ ); //从第 k 个字节处读取文件

if(size<1)//没有数据，则返回break;

for(j=0;j<size;j++)

uart\_write( data\_addr[j]); //调用串口系统调用输出文件数据

}

fileclose(handle); //关闭文件kill(); //结束本程序



通过测试，本程序可以统一支持设备文件和普通文件的读操作，对两种不同的文件类型有较好的兼容。图6-5展示了测试应用程序的过程，该过程首先创建普通文件并向该文件添加数据，然后调用read\_file程序且可以正确读取该普通文件，最后调用read\_file程序读取设备文件/dev/uart0。

图 6-5 测试文件读取程序截图

#### 2、编写和运行文件数据添加程序

本项目提供示例程序write\_file. c 来写入指定路径下的文件。程序大体流程为：

（1）申请一个空间来存放文件名；（2）打开指定文件，判断是否打开成功，不成功则终止程序；（3）往缓冲区里写入数据并调用fileappend 写入指定个数的字节；

（4）关闭文件并终止程序。该程序核心语句如下：

char\* addr=malloc(1024); //申请1024个字节的空间

if((int) addr==-1) kill(); //申请失败，结束本程序

readstr(addr); //调用串口系统调用读取文件名unsigned char\* data\_addr ;

int handle =fileopen(addr); //打开文件

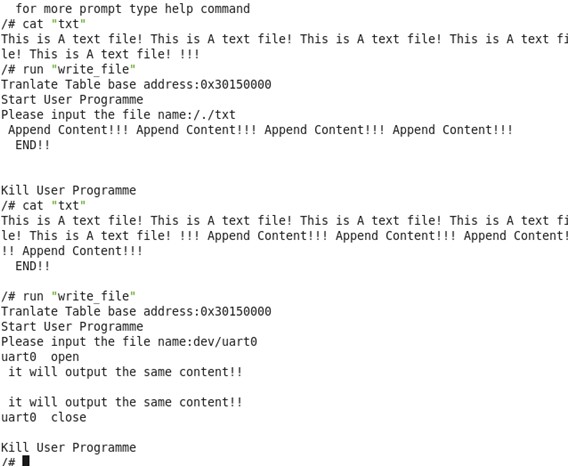
if(handle==-1) kill(); //申请失败，结束本程序

data\_addr=filebuffer(handle); //获取文件的缓冲区

//这里修改缓冲区

fileappend(handle, size); //写入文件size个字节

fileclose(handle); //关闭文件



kill(); //结束本程序

图 6-6 测试文件写入程序截图

通过测试，本程序可以统一支持设备文件和普通文件的写操作，对两种不同的文件类型有较好的兼容。图6-6展示了测试应用程序的过程，该过程首先显示普通文件的内容，然后调用write\_file程序，且可以正确向该普通文件写入数据，最后调用write\_file程序向设备文件/dev/uart0写入数据。

### 6.4.2 测试系统调用

分别在管理模式以及用户模式下对系统调用进行测试，对各个系统调用功能，包括同一系统调用号的不同参数的不同功能程序段依次进行测试。测试内容和测 试结果如表6-10所示。

表6-10 测试系统调用

| 系统调用号 | 参数 | 功能以及对应的系统调用 API | 测试结果 |
| --- | --- | --- | --- |
| 28 号  串口系统调用 | 0、r1 | 串口输出字符、uart\_write(char c) | 正确 |
| 1 | 串口读取字符、uart\_read() | 正确 |
| 2、r1 | 输出十进制字符串、uart\_write\_dec(int i) | 正确 |
| 3、r1 | 输出十六进制字符串、uart\_write\_hex(int i) | 正确 |
| 32 号系统调用 | 0 | 恢复应用程序（禁止应用程序使用该系统调用） | 正确 |
| 1 | 杀死应用程序、kill() | 正确 |
| 36 号  文件系统调用 | 0、r1 | 打开文件、fileopen(char\* file\_path) | 正确 |
| 1、r1 | 写文件、fileappend(int handle,int size) | 正确 |
| 2、r1 | 读取文件、fileread(int handle,int block\_no) | 正确 |
| 3、r1 | 关闭文件、fileclose(int handle) | 正确 |
| 4、r1 | 返回文件数据缓冲区、filebuffer(int handle) | 正确 |
| 40 号  内存系统调用 | 0 | 申请内存、malloc(int size) | 正确 |
| 1 | 在启动下一个程序前释放前一个程序申请的内存 | 正确 |

## 6.5 本章小结

本章主要介绍了对本文件系统各个模块的测试过程和结果，主要包括测试串口控制程序，测试闪存控制程序，测试文件操作的正确性、一致性和性能，测试设备文件操作以及文件系统调用。

# 第七章 结束语

本文详细地解析了本项目的开发方法和系统结构，探讨了ARM920T的硬件体系结构以及EXT2文件结构。本人通过学习掌握了ARM920T下内核的工作原理和开发方法，以及通过对EXT2存储格式的研究，编写了这一小型文件系统程序。本项目的主要工作分为以下五个部分：

（1）对嵌入式系统和ARM920T体系进行了研究；

（2）分析了基于Neo1973的嵌入式开发平台，并构建了基于这个平台的嵌入式开发环境；

（3）对ARM920T机制、设备访问进行学习和编程；

（4）研究了EXT2存储结构，在Neo1973平台上，实现基本的文件管理程序、实现基本的设备文件模块、实现文件系统调用；

（5）根据对文件系统的编程实践和对相关资料的学习，提出基于两层多叉树的文件查找算法、基于数据块位图的下一空闲块分配算法和基于链表统计的空闲块分配算法；

（6）测试文件系统的各个模块的功能的正确性、一致性和得到相关的性能数

据。

本项目虽然对软件系统进行较为完整的编程，但是在某些代码部分还存在某

些未知的错误或者故障隐患，导致项目后期进行过多的测试和纠正Bug的工作（导致增加了过多的补丁代码，影响了系统性能）。故障风险的地方可以推断为：模式 切换堆栈区分配问题、文件系统方法间的全局数据耦合性过强、功能模块没分离 好（源文件分块不当，基本操作函数功能不够全面）、虚拟内存申请使用直接映射 的问题。另外本项目的文件系统只针对EXT2的文件存储格式，没能将文件系统的功能抽象出来进行开发，而且缺少数据高速缓冲区，使得数据读写速度较慢。 这些问题可以归结为本人对系统功能的统筹规划分块不完善、把握不全面的问题、冲突问题没预测得当的问题。需要在未来的学习中学习和提高全面的分析问题和 预测问题的能力。

致 谢

两年的研究生生活即将划上一个句号，而于我的人生却只是一个逗号，我将面对人生的新的阶段的开始。六年的大学生涯里发生了很多事情，和大家相识相知，共同学习和生活，使我留下难以忘怀的记忆，大家各奔前程的时候，依然愿大家前程似锦。

感谢导师的指导，您治学严谨，学识渊博，思想深邃，视野雄阔，为我营造了一种良好的精神氛围。授人以鱼不如授人以渔，置身其间，耳濡目染，潜移默化，使我树立了崭新的思想观念，确立了远大的学术目标，掌握了巧妙的思考方式，从论文选题到论文撰写，全程经由您悉心的指点，到最后领悟到的奥妙，常常令我有一种“柳暗花明又一村”的感觉。

感谢我的父母，焉得谖草，言树之背，养育之恩，无以回报，愿你们永远身 体健康。论文之所以顺利完成，需要感谢在我身边陪伴我的好朋友的关心和帮助，以及老师的指导，没有你们就不会有今天的我。感谢那些一直陪在我身边的人， 相信有你们在我身边，我会走的更加坚实。

在感谢学院学校为我提供良好的学习考研毕业设计的环境的同时，再一次感谢所有在毕业设计中、在我生活中曾经给予我帮助的良师益友。

参考文献

[1] ARM920T(Rev 1) Technical Reference Manual

[2] USER'S MANUAL S3C2410A–200MHz & 266MHz 32-Bit RISC Microprocessor Revision 1.0.

[3] OpenMoko Neo1973 Develop Manual

[4] Samsung Corporation. K9F1208U0M Datasheet.

[5] 倪继利. linux 内核分析及编程. 电子工业出版社. 2005

[6] 徐海斌. GNU make中文手册. 网络资料

[7] 马忠梅, 李善平, 康慨等. ARM& Linux 嵌入式系统教程. 北京航空航天大学出版社. 2004

[8] 大奖赛组委会. 2008TI高级嵌入式控制器C2000大奖赛优秀论文汇编. 电子工业出版社. 第1版. 2008.

[9] 探矽工作室. 嵌入式系统开发圣经. 中国铁道出版社. 2003

[10] 李无言.《一步一步编写嵌入式操作系统》. 电子工业出版社. 2005

[11] 汤子瀛, 哲凤屏, 汤小丹. 计算机操作系统. 西安电子科技大学出版社. 2005

[12] 李敬兆. 8086/8088和ARM核汇编语言程序设计. 中国科学技术大学出版社. 2006

[13] Karim Yaghmour. Building Embedded Linux Systems. OReilly Media, Inc. April 2003

[14] 蒋静, 徐志伟. 操作系统. 机械工业出版社. 2004

[15] 文全刚. 汇编语言程序设计--基于ARM体系结构. 北京航空航天大学出版社. 2010

[16] Sriranga Veeraraghavan. 精通shell编程. 人民邮电出版社. 2003

[17] Steve Furber. ARM Soc体系结构. 北京航空航天大学出版社. 2002

[18] 陈俊宏. Embedded Linux嵌人式系统原理与实务. 中国铁道出版社. 2004

[19] 关永. ARM嵌入式微处理器体系结构及汇编语言程序设计. 电子工业出版社. 2005

[20] Maurice Bach. The Design of the Unix Operating System. Prentice Hall. 2000

[21] David Pitts. Red Hat Linux Unleashed. Sams Publishing. 1998

[22] Paul Lensing, Dirk Meister, Andr´e Brinkmann. hashFS: Applying Hashing to Optimize File Systems for Small File Reads. IEEE. 2010

[23] Tzer-Ta Tseng, Yarsun Hsu. A Flexible and Cost-effective File-wise Reliability

68面向ARM的小型文件系统设计与实现

Scheme for Storage Systems. IEEE.2010

[24] Neil Matthew&Richard Stones. Beginning Linux Programming, 2w Edition. Published by Wrox Press. 2002

[25] （美）刘易斯著, 陈宗斌译. 嵌入式软件基础——C语言与汇编的融合（翻译版）. 高等教育出版社. 2005

[26] Aleatha Parker-Wood, Christina Strong, Ethan L. Miller, Darrell D. E. Long. Security Aware Partitioning for Efficient File System Search. IEEE. 2010

[27] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, Yeh-Ching Chung. PQEMU: A Parallel System Emulator Based on QEMU. IEEE. 2011

[28] 何英. 一种嵌入式文件系统的设计与实现: [学位论文]. 成都: 电子科技大学, 2001.3-2

[29] 潘光晖. 嵌入式文件系统的设计与实现: [学位论文]. 成都: 电子科技大学, 2003.5

[30] 王健. 基于NANDFlash的嵌入式文件系统的设计与实现: [学位论文]. 北京: 北京交通大学, 2008.6

[31] 鲍颖力. 基于虚拟机QEMU的嵌入式全系统仿真测试环境的研究与实现: [学位论文]. 上海: 上海交通大学, 2011.6

[32] 成巍. 嵌入式文件系统的设计和实现: [学位论文]. 成都: 电子科技大学, 2006.1

[33] 张雷. 基于闪存的嵌入式文件系统的研究与实现: [学位论文]. 成都: 电子科技大学, 2004.12

[34] 杨晓霞. 嵌入式系统文件管理和设备管理: [学位论文]. ft东: ft东大学, 2006.4-5

[35] 余婷婷. 嵌入式文件系统的研究与设计: [学位论文]. 武汉: 武汉理工大学, 2007.4

[36] 李海霞. 基于NAND闪存的嵌入式文件系统研究与实现: [学位论文]. 浙江: 浙江大学, 2008.5

[37] 展中华. 基于NOR Flash的嵌入式Linux文件系统设计与实现: [学位论文]. 北京: 北京地质大学, 2008.5

[38] 刘柳. 基于大容量NAND闪存文件系统关键技术研究: [学位论文]. 浙江: 浙江工业大学, 2011.5

[39] 赵梨斌. 面向云存储的分布式文件系统关键技术研究: [学位论文]. 西安: 西安电子科技大学, 2011.1

[40] 卢科. 闪存数据库系统存储管理及可裁剪性研究: [博士学位论文]. 合肥: 中国科技技术大学, 2012.4