

兰州大学

硕士学位论文

基于ARM的嵌入式MINIX3操作系统的移植

姓名： 屈鹏

申请学位级别： 硕士

专业： 数学 计算数学

指导教师： 周宇斌

20080501

摘 要

嵌入式操作系统是嵌入式系统应用的核心。完成简单功能的嵌入式系统一般不需要操作系统,但是随着所谓后PC时代的来临,嵌入式系统设计日趋复杂,嵌入式操作系统就必不可少了。一般而言,嵌入式操作系统不同于一般意义的计算机操作系统,它有占用空间小、执行效率高、方便进行个性化定制和软件要求固化存储等特点。

本文在讨论了嵌入式操作系统的基本理论之后,论述了MINIX 3操作系统的特点,指出了MINIX 3向嵌入式操作系统方面发展的意义和可行性。建立可移植代码是操作系统移植的首要步骤和重要途径,本文又论述了建立可移植代码的一般方法和过程。而后深入分析了MINIX 3操作系统的内核结构及组成部分,重组了内核目录树,列出了移植MINIX 3内核需要重新编写的内核接口。在MINIX 3已有代码的基础上,分离了一部分机器相关代码和体系结构不相关代码,建立了相应的可移植代码。

提到嵌入式,就不能不提ARM, ARM体系结构是目前最流行的嵌入式微处理器体系结构。本文就是针对ARM处理器体系结构来展开的,文中简单介绍了ARM处理器的情况,并且对应ARM体系结构修改了MINIX 3内核的部分代码。最后在深入分析了MINIX 3操作系统进程调度算法和代码之后,结合软实时操作系统进程调度特点和要求,改进了MINIX 3进程调度策略,使之适合软实时操作系统的要求,并给出了修改后的相关源代码。

关键词: 嵌入式操作系统; 机器不相关代码; 机器相关代码; 内核驱动程序模型; 非抢占式调度算法; 软实时进程调度

Abstract

Embedded operating system is the kernel of embedded system application. The embedded operating system that be used to complete simple function don't need operate system generally, but along with so-called after PC ages, embedded operating system becomes indispensable as the embedded system designs gradually complicated. Generally speaking, the embedded operating system differs from the general computer operating system, because it has a lot of characters, such as taking up small space, effective performance, the convenience carry on characteristic making to order and save software in ROM and so on.

This thesis explained the basic theories of embedded operating system first, then discussed the characteristics of MINIX 3 operating system and point out the meaning and possibility that develop the MINIX 3 toward the embedded operating system. Creating the portable code is the first step and important path to port a operating system, so the thesis discussed a general method and process of the creating portable code. Next thorough analyzed MINIX 3 operating system on kernel structure and kernel constitute parts, then reorganized the kernel tree and listed the kernel interfaces that must be rewrite for porting the MINIX 3 kernel. At the foundation of MINIX 3 existing code, separated machine dependent code and machine independent code and create the portable code.

Talking about embedded system, have to talk ARM. The ARM architecture is the most popular embedded microprocessor architecture currently. This thesis simply introduced the ARM architecture microprocessor and rewrote a part of MINIX 3 kernel code. After through analyzing the process scheduling algorithm and relative code, I changed the process scheduling policy of MINIX 3, making it meet the soft real-time process scheduling request. Finally, in the appendix, is the implementation of the process scheduling.

Key Words: embedded operating system; machine independent code; machine depended code; kernel driver model; non-preemptive scheduling algorithm; soft real-time process scheduling

原创性声明

本人郑重声明：本人所呈交的学位论文,是在导师的指导下独立进行研究所取得的成果。学位论文中凡引用他人已经发表或未发表的成果、数据、观点等,均已明确注明出处。除文中已经注明引用的内容外,不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究成果做出重要贡献的个人和集体,均已在文中以明确方式标明。

本声明的法律责任由本人承担。

论文作者签名： 屈鹏 日期： 2008.5.25

关于学位论文使用授权的声明

本人在导师指导下所完成的论文及相关的职务作品,知识产权归属兰州大学。本人完全了解兰州大学有关保存、使用学位论文的规定,同意学校保存或向国家有关部门或机构送交论文的纸质版和电子版,允许论文被查阅和借阅；本人授权兰州大学可以将本学位论文的全部或部分内容编入有关数据库进行检索,可以采用任何复制手段保存和汇编本学位论文。本人离校后发表、使用学位论文或与该论文直接相关的学术论文或成果时,第一署名单位仍然为兰州大学。

保密论文在解密后应遵守此规定。

论文作者签名： 屈鹏 导师签名： 周宇斌 日期： 2008.5.30

第1章 绪论

1.1 嵌入式系统概述

随着多媒体技术、通信技术相结合的信息时代的快速发展，互联网的广泛应用，嵌入式系统和嵌入式技术已经和人们的生活紧密相连。嵌入式系统的应用涉及众多领域，深入到了社会、生活的各个方面。其中主要有：家用电器、通信设备工业、仪器仪表、导航控制、商业和金融、办公设备、交通运输、建筑和医疗等领域。

嵌入式系统已经有近30年的发展历史，其发展过程是硬件和软件交替进行的双螺旋式发展。

最早的单片机是1976年的Intel公司推出的8048，Motorola同时推出了68HC05，zilog公司推出了Z80系列，这些早期的单片机均含有256字节的RAM、4K的ROM、4个8位并行接口、一个全双工串行接口、两个16位定时器。在20世纪80年代，Intel公司又进一步完善了8048，在8048的基础上研制成功了8051。

1981年Ready System开发了世界上第一个商业嵌入式实时内核（VTRX32），该内核包含了许多传统操作系统的特征，包括任务管理、任务间通信、同步与互斥、中断支持、内存管理等功能。这个实时内核可以运行在8051单片机上。

嵌入式微控制器的出现使计算机工程应用史上的一个里程碑，随着微电子技术的飞速发展，CPU已经成为低成本器件。在可能的情况下，各种机电设备已经或者正在嵌入CPU成为嵌入式系统。目前，中、高档8位嵌入式微控制器，16位、32位嵌入式微控制器，以及一些专用嵌入式微控制器（如数字信号处理、数字图像处理、通信控制单片机等）已在通信系统、因特网系统、非嵌入式计算机系统、工业测控系统、机器人感知行走系统、分布式测控系统、快速多机实时处理系统和图像系统中成为不可缺少的重要组成部分。

1.1.1 嵌入式系统的定义

我们用计算机系统来处理并管理各种数据，这里所说的数据包括文字、数字、图片以及各种指令。人们希望能制造出各种智能机器，这些机器需要一个系统来管理，而这些机器又可能很小，小如手机，这就需要有一个嵌入在其中的管理系统，而这个系统如何管理这些机器，就要看它的软件了。我们把带有微处理器的专用软硬件系统统称为嵌入式计算机系统，通常称为嵌入式系统。

但是不一定符合以上要求的计算机系统就能被称为嵌入式系统。例如一台安装了Windows XP的个人计算机就不能称为嵌入式系统。对此，IEEE（国际电器和电子工程师协会）对嵌入式系统的定义是：用于控制、监视或者辅助操作机器和设备的装置（devices used to control, monitor, or assist the operation of equipment, machinery or plants）。可以看出此定义是从应用方面考虑的，嵌入式系统是软件和硬件的综合体，还可以涵盖机电等附属装置。

功能层	应用程序APP	
软件层	应用平台	
	板级支持包	
	嵌入式操作系统	
中间层	硬件抽象层（HAL）/板级支持包（BSP）	
硬件层	D/A 嵌入式微处理器	通用接口
	A/D	ROM
	I/O	SDRAM

图1.1：嵌入式系统的组成

综合各方面，我们得出嵌入式系统的一般的定义：“以应用为中心，以计算机技术为基础，软件硬件可剪裁，功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。”

1.1.2 嵌入式系统的组成

总体上嵌入式系统可划分成硬件和软件两部分，硬件一般由高性能的微处理器和外围的接口电路组成，软件一般由硬件抽象层、嵌入式操作系统、板级支持包。

硬件层：硬件是整个嵌入式操作系统和应用程序运行的平台，包括输入和输出接口/驱动电路、处理器、存储器、定时器、串口、中断控制器、外设器件、图形控制器及相关系统电路等部分。

中间层：硬件抽象层（HAL），负责对各种硬件功能提供软件接口，包括硬件初始化、硬件时钟、中断板级支持包、计时器时钟、总线管理、内存地址映射等。

嵌入式操作系统：实现对资源的访问和管理，完成任务调度，支持应用软件的运行及开发。

板级支持包（Board Support Package, BSP）：BSP针对某一个特定的嵌入式系统，提供与硬件相关的设备驱动。

应用平台：应用开发商提供的可重用的应用平台，封装一些常用功能，同时提供API接口，用于二次开发。

应用软件：应用软件层位于嵌入式系统层次结构的最顶层，直接为用户交互。

1.1.3 嵌入式系统的特点

嵌入式系统是一种针对于特定任务、特殊环境而进行特殊设计的定制产品，所以与传统的计算机系统相比，有以下特点：

- 1. 操作系统内核小，资源少

2. 专用性强
3. 系统稳定持久
4. 软硬件结合紧密
5. 开发需专门的环境和开发工具
6. 软件要求固态化存储
7. 实时性要求高

1.1.4 嵌入式系统的分类

由于嵌入式系统的用途广泛，种类繁多，人们对嵌入式系统的理解也各不相同，所以其分类方法也存在着多种方式：

1. 根据嵌入方式分为：整机式嵌入、部件式嵌入、芯片式嵌入。
2. 根据嵌入式软件类型分为单线程程序嵌入式系统、事件驱动程序嵌入式系统。
3. 根据实时性可分为实时系统和非实时系统。
4. 根据嵌入式系统的复杂程度，可分为单微处理器嵌入式系统、组件嵌入式系统、分布式嵌入式系统。

1.1.5 嵌入式操作系统

嵌入式操作系统（Embedded Operating System, EOS），是一种特殊的嵌入式软件，是基于嵌入式操作系统的嵌入式系统中软件层的基础，其它应用都是建立在嵌入式操作系统之上。它实际上是段系统复位后首先执行的程序，主要负责嵌入式系统的全部软、硬件资源的分配、调度、控制、协调并发活动，它将CPU时钟、中断、定时器存储器、I/O等都封装起来，提供给用户的是一个标准的API接口。嵌入式操作系统必须体现其所在系统的特征，能够通过装卸某些模块来达到系统所要求的功能。

许多早期的嵌入式系统开发者认为嵌入式系统不需要操作系统。但现在除了最简单的系统外，越来越多的嵌入式系统引入了操作系统。比如中断驱动系统在引入嵌入式操作系统之后，系统的可靠性、安全性、可扩展性、功能性、灵活性、可管理性都有了大的提高。

在很多嵌入式操作系统中封装了越来越多的功能。除了对任务的切换、高度通信、同步、互斥、中断管理、时钟管理等，还可进一步封装内存管理、网络通信协议、文件管理等功能，这些功能可根据需要进行剪裁。

1.1.5.1 嵌入式操作系统的特点

EOS是相对于一般操作系统而言的，它除了具备一般操作系统最基本的功能，如任务调度、同步机制、中断处理、文件处理等外，还有以下特点：

1. 可装卸性。开放性、可伸缩性的体系结构。
2. 强实时性。EOS实时性一般较强，可用于各种设备控制当中。
3. 统一的接口。提供各种设备驱动接口。
4. 操作方便、简单。提供友好的图形GUI。
5. 提供强大的网络功能。支持TCP/IP协议及其他协议，提供TCP/UDP/IP/PPP协议支持及统一的MAC访问层接口，为各种移动计算设备预留接口。
6. 强稳定性，弱交互性。
7. 固化代码。在嵌入式系统中，嵌入式操作系统和应用软件被固化在嵌入式系统计算机的ROM中。

1.1.5.2 嵌入式操作系统的分类

按经营模式分，目前市场上主流的嵌入式操作系统可分为商用和开源两类；按实时性划分，嵌入式操作系统可分为实时嵌入式操作系统和非实时嵌入式操作系统。

1.1.5.3 几种典型的嵌入式操作系统的比较

1. VxWorks

VxWorks操作系统是美国WindRiver公司于1983年为分布式环境设计开发的具备网络功能的一种嵌入式实时操作系统，是Tornado嵌入式开发环境的关键组成部分，是典型的商用操作系统。其友好的开发环境、高性能的系统内核，在实时操作系统领域是首屈一指的。

VxWorks具有良好的实时性、稳定性和可靠性，具有可裁剪的微内核结构、高效的任务管理、灵活的任务间通信、微秒级的中断处理，支持多种物理介质及标准的、完整的TCP/IP网络协议等，支持多种处理器，如x86、i960等。

2. Windows CE

Windows Embedded是由Microsoft开发的主要用于具有丰富应用程序和服务的32位嵌入式系统，其家族成员有Windows CE 3.0，Windows NT Embedded 4.0等。

Windows CE与Windows系列操作系统有较好的兼容性，无疑是Windows CE推广的一大优势。其中Windows CE3.0是一种针对小容量、移动式、智能化、32位、模块化实时嵌入式操作系统。

3. 嵌入式Linux

这是嵌入式操作系统的一个新成员，其最大的特点是源代码的公开并且遵循GPL协议。嵌入式Linux的应用领域非常广泛，并且有着庞大的开发人员群体，网络功能优秀，支持硬件数量庞大。

嵌入式Linux的一个缺点是让Linux体系提供实时性能的话需要添加实时软件模块。

1.2 MINIX操作系统介绍

编写MINIX操作系统的目的是向计算机科学专业的学生提供一个实践的工具，用来编程和学习一个完整的操作系统。当时其它可用的操作系统，例如UNIX，是商业化的产品，而它的许可证是不允许在课堂上使用或者研究源代码的。如果只讲解理论，会使学生对操作系统产生片面地理解。为了避免这种情况，Andrew S.Tanenbaum决定从头编写一个全新的操作系统，这就是MINIX。从用户的角度看，MINIX与UNIX完全兼容，但是从内部实现来看，它是与UNIX完全不同的。它的组织结构比UNIX更加层次化、模块化，而并非UNIX的整体式结构。现在又很多可用的开源的操作系统，其中最著名的要数GNU/Linux和BSD。其中Linux的内核就是由早期的MINIX内核发展而来。BSD是许多发行版本中的一个。对于我来说，要通过以上一种系统去学懂操作系统太难了。这些系统拥有很多特性所以非常复杂。现在有很多专家都在从事这些系统的开发，而这些专家团体非常庞大而且还在增长。

MINIX把事情简单化，它很小、高效而且快速。MINIX是由模块化的组件逐步组成的，以保持代码的独立性、整个系统的可扩展性和稳定性。在早期的时候，核心没有使用硬件的一些功能，以保持对旧系统的兼容和对其他系统的可移植性。例如，到这篇论文书写的时候，还不支持虚拟内存，尽管大部分系统都硬件支持虚拟内存以简化软件的实现。但是MINIX的特性和支撑程序是在发展的。使MINIX支持虚拟内存和虚拟文件系统的工作正在进行中，这些新特性使得MINIX成为一个更好的发行物。

1.2.1 关于MINIX 3

MINIX的第2个版本于1997年发行，直到2004年，Tanenbaum认为整个系统已经变得过于庞大且不可靠，所以推出了MINIX 3。这个版本是为IBM PC兼容机开发的，它支持许多旧的体系结构，像Motorola 68000,早期的Macintosh, SUN4和ATARI。与MINIX的前一个版本相比，内核部分进行了重新构造，将几乎所有的设备驱动程序都放到了用户空间，只保留了时钟任务和系统任务，进一步减少了内核空间的进程，使得内核更加简洁、稳定。新版本MINIX既可用于PC机，又可用于嵌入式系统。尤其是对于嵌入式应用来说，简洁、模块化和可靠性是非常关键的。

1.3 目标平台简介

ARM (Advanced RISC Machines) 是一种用于微控制器/微处理器领域的RISC处理器系统结构，综观ARM的发展历史，不能不说是一个奇迹。ARM，既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。1991年ARM公司成立于英国剑桥，主要出售芯片设计技术的授权。目前，采用ARM技术知识产权 (IP) 核的微处理器，即我们通常所说的ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基

于ARM技术的微处理器应用约占据了32位RISC微处理器75%以上的市场份额，ARM技术正在逐步渗入到我们生活的各个方面。

ARM公司是专门从事基于RISC技术芯片设计开发的公司，作为知识产权供应商，本身不直接从事芯片生产，转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从ARM公司购买其设计的ARM微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的ARM微处理器芯片进入市场。目前，全世界有几十家大的半导体公司都使用ARM公司的授权，因此既使得ARM技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场被消费者所接受，更具有竞争力。

ARM体系结构是一套先进的处理器体系结构，其先进性就体现在它的处理器结构的模块化上。ARM CPU核分为多个系列，如ARM7、ARM9、ARM9E、ARM10E、SecurCore及ARM11，分别针对不同需求的应用。

本文实验选用的实验工具箱的CPU是三星S3c2410X，CPU核是ARM920T。实验仪器硬件配置如图1.2。

1.4 向嵌入式操作系统移植的可行性

MINIX系列操作系统符合POSIX标准，是类UNIX操作系统，在用户的层次上来看是与UNIX完全兼容的，但是在内部结构上来讲则是完全不同的。由于比UNIX晚十年推出，所以比UNIX采取了更为先进，更加模块化的设计理念，整个操作系统就是所有系统进程的集合，比起UNIX的整体式结构，显然要更加安全，更加稳定。MINIX自v1版本以来，系统一直采用了层次化的结构，从版本v1开始即采用微内核的结构，内核空间中只保留最为核心的部分和设备驱动程序，文件系统等其它的系统进程则放到了用户空间。

到了MINIX v3版本推出时，系统的内核部分进一步减小，内核中只保留了三个进程：IDLE进程、系统任务和时钟任务，而将剩下的设备驱动程序都放到了用户空间，使得内核部分的可执行代码只有不到4000行，这大大减少了内核出错崩溃的可能性，进一步提高了系统的稳定性。在非微内核结构的操作系统的内核中，驱动程序通常都是包含错误最多的部分，一个驱动程序出错就有可能引起操作系统的崩溃。而在MINIX 3中，这种情况不可能出现，即使驱动程序崩溃，也不会影响到操作系统，而崩溃的驱动程序也可以统过用户空间的再生服务器（Reincarnation Server, RS）重新启动。而且需要添加一个新的驱动程序的时候对操作系统内核的影响也不会很大。

可以说，从MINIX 3推出以来，MINIX就开辟出了一条新的发展方向—即向嵌入式系统领域进军的可能性。总结一下，MINIX 3操作系统向嵌入式操作系统领域发展的优势有：

- MINIX 3操作系统采用微内核结构，内核小，内核可执行代码部分只有不到4000行。

配置名称	型号	说明
CPU	ARM920T 结构芯片三星S3c2410X	工作频率203MHz
FLASH	SAMSUNG K9F1208	64MB NAND
SDRAM	HY57V561620AT-H	32M*2=64M
EtherNet网卡	AX88796	两片, 10/100M自适应
LCD	LQ080V3DG01	8寸16bitTFT
触摸屏	SX-080-W4R-FB	FM7843驱动
LED	ZLG7290	四个共阴极LED
USB接口	4个HOST/1个DEVICE	有AT43301构成USB HUB
UART/IrDA	2个RS232, 1个RS485, 1个IrDA	从处理器的UART2引出
AD	由S3C2410芯片引出	3个电位器控制输入
AUDIO	IIS总线, UDA1314芯片	44.1KHz音频
扩展卡插槽	168Pin EXPORT	总线直接扩展
GPS.GPRS扩展板	SIMCOM的SIM100-E模块	支持双语音通信
IDE/CF卡插座	笔记本硬盘, CF卡	
PCMCIA和SD卡插座	PCMCIA型号为DWL-650	
PS2	PC键盘和鼠标	由ATMEGA8单片机控制
IC卡座	AT24CXX系列	由ATMEGA8单片机控制
DC/STEP电机	DC由PWM控制, STEP由74HC573控制	
CAN BUS	由MCP2510和TJA1050构成	
Double DA	MAX504	两个10位DAC端口
调试接口	JTAG	14针、20针

图1.2: 试验平台配置

- 系统占用资源少, 可以在嵌入式系统有限的硬件资源上发挥出性能。
- 系统采用高度的模块化设计, 系统具有良好的伸缩性、扩展性、开放性等都符合嵌入式系统可剪裁、开放性等要求。
- MINIX 3系统的结构从根本上决定了它必定是一个安全的、稳定的、持久的操作系统, 这也是嵌入式系统所要求的。
- MINIX 3为设备驱动程序提供了一个与系统其它部分一样的接口: 将请求的消息发送给驱动程序, 然后驱动程序根据消息中的操作码和参数执行相应的操作。
- MINIX 3的内存管理并没有使用虚拟内存管理, 这个特性在某种程度上也使得它易于向嵌入式系统移植。

1.5 本文的主要工作

借鉴目前国内外对MINIX操作系统的研究和对于将MINIX操作系统向其它体系结构移植的经验,本人进行了MINIX 3操作系统向ARM平台进行嵌入式操作系统开发的学习和实验。论文完成了以下工作:

1. 对嵌入式系统进行了学习研究,归纳总结了嵌入式系统的特征。
2. 对MINIX 3的内核结构、代码组织进行了深入的研究,并针对移植对MINIX 3内核部分源代码进行了划分,分离了机器相关部分和不相关部分代码,并对分离后的代码进行了重新组织。
3. 本人阅读了MINIX部分源代码,分析了MINIX 3中进程调度的原理和实现。
4. 修改了内核中进程调度部分相关代码,对向ARM体系结构的移植进行了实验。

论文的组织围绕以上工作展开。

第1章介绍了嵌入式系统和嵌入式操作系统的定义和特征,对MINIX 3操作系统向嵌入式系统进行移植的优势和便利进行了分析。

第2章是移植代码的前期工作,分析了如何在保持MINIX 3现有布局的基础上划分代码,内核驱动程序模型的建立。然后讲述了MINIX 3代码中使用的汇编语言。

第3章首先分析了MINIX 3操作系统的内部结构以及系统各部分的功能,重新建立代码树。后又分析了MINIX 3中的进程调度,并对相关代码进行了移植的尝试。

第4章是这篇论文的结论。

第2章 MINIX 3移植的准备

当要移植一个操作系统的时候，通常要从已存在的代码开始。如果运气好的话，这些代码已经列出了机器相关代码（Machine Independent Code, MDC），很快能找到它们，那么要找机器不相关代码（Machine Depended Code, MIC）的入口也很容易，移植也会简单而快速。如果不是这样的话，就需要一个合理的方法去区分代码，但是要这样做经验和常识很重要。第2章“MINIX 3移植的准备”将在所有的入口点都知道的情况下给出建立移植代码的方法。当然也会说明在这样建立移植代码的过程中如何转换现有的代码，并保持现有的布局设计。

2.1 移植一个操作系统

移植一个操作系统第一件要做的事情是了解所有有关目标平台的东西。在这篇论文中指的是以ARM处理器为基础的嵌入式系统。第二件事情是理解MINIX本身。移植系统时会频繁地遇到由MINIX中机器不相关代码产生的问题，然后又会遇到由机器相关代码中的函数产生的问题，而这些函数正是需要我们移植的。越快找到出错的函数，移植也就越容易。追踪一个系统函数找到问题将会是一项困难的工作，如果能够知道这个函数从MIC最终到达了MDC中的什么地方，将会对这个函数的追踪有很大的帮助。

还需要知道当前系统的组成（在本文中是Intel 32位系统），当然不需要像目标系统那样知道一些具体的细节，但是一些基本的东西还是需要了解。编写一个操作系统需要熟悉目标体系结构，而对体系结构的认识程度取决于经验，这可能需要几年的专业经验，任何没有经验的人应该耐心地去积累经验，经验不会一夜之间获得或者靠读一两本书而获得。

2.1.1 完整的系统

一个操作系统的组成不光只有一个内核，它还拥有许多程序和公用程序（utility），以管理计算机和使计算机运行起来。一个真正的操作系统还应该能提供一个开发环境，用来创建程序。MINIX 3提供了一个C语言编译器（ACK）和库，并且它们拥有遵循POSIX标准的API。当要移植一个操作系统的时候，不仅要移植内核，还必须要移植库。这些库被系统编译器用来创建程序和公用程序，当然也包括编译器自己。

在大多数情况下，系统软件用低级语言和高级语言共同来编写。低级语言是目标机的汇编语言，而高级语言可以是C，C++或者混合使用。在开始移植之前应该看看此系统在目标机上有没有相同的高级语言的编译器。MINIX由ANSI C编写，对GNU工程来说，现在有一个非常好编译器：GCC。一些人认为C语言是一种体系结构不相关的语言是因为几乎所有的硬件体系都有自己C编译器。甚至有这样的编译器，将语言X程序编译成C程序，然后再调用C编译器生成可执行文件。

当编译器使用了高级功能，例如特殊类型，或者比特位的对齐方法，总会带来兼容性问题。如果在编程时避免使用一些非标准的技巧，那么会使将来可能的移植变得简单很多。

所有的程序代码都被分为两部分：机器相关代码（MDC）和机器不相关代码（MIC）。如果不是这样，那么我们就只能编写通用代码，比如编译成的JAVA字节码供虚拟机使用；或者只能编写供一个体系结构使用的代码，像汇编代码。在这里规律是简单的：使用的语言越接近硬件，那么可移植性越小。由源文件编译而成的目标代码非常依赖于硬件和软件，比如MINIX 3中Intel体系结构的X86汇编指令和C语言，所以它永远是不可移植的。

移植一个操作系统就是所有关于从一个体系结构向另一个体系结构重写机器相关部分代码的过程。而建立可移植代码是系统移植的一个重要途径。编写可移植代码的基础是分离机器相关代码和机器不相关代码，这样的话我们可以清楚地知道机器相关代码是哪一部分，而这部分代码的重写也可能就变得简单了。

MINIX 3的源代码树由两部分组成：Intel x86汇编代码和C代码。可以确定的是所有的这些汇编代码都不能在ARM体系结构上进行编译，所以所有这些汇编代码提供的功能都必须移植到ARM相关汇编器上或者用C实现。

虽然C代码可以在每一个不同的体系结构上编译，但是有可能这些代码中仍然存在机器相关性，因为这些代码直接或间接地同硬件打交道。我们要做的就是找到软件的相同点，而重新编写不同点。比如在MINIX 3设备I/O部分的代码中，就存在很多这种情况。原因是由于Intel体系结构和ARM体系结构对I/O端口的使用方式不同。

综合地说，要让MINIX的内核工作，必须要从硬件那里得到特定的关键服务，例如时钟系统每秒钟提供的60个时钟中断。这些东西存在于终端、存储器和时钟系统中。这些服务必须跟一般内核管理代码分离，例如进程调度的代码或者进程间通信的代码。MINIX 3的原始代码并没有区分在“内核中的”驱动程序和剩下的管理部分代码，但是在这里我们需要将它们分离，也就是内核中MIC和MDC的分离，这种分离方法将在下一节中说明。为了移植MINIX 3，我们要在原始代码的基础上建立可移植代码，驱动程序模型正是用来将机器相关代码从机器不相关代码中分离中出来的。

2.1.2 可移植代码的创建

要创建一个可移植的系统，必须在机器相关代码和机器不相关代码之间做一个分离。分离做得越好，那么移植也就越容易。拿内核来说吧，理想情况下，当前体系结构下内核的两部分代码都能不相干的单独编译，并且能链接形成内核。但这样的意思不是说Intel32位体系结构下编译的机器不相关代码可以和ARM体系结构下编译的机器相关代码链接起来，因为它们有着不同的机器指令。编译的机器相关代码和机器不相关代码部分必须属于一样的体系结构。

代码的分离增强了设计的模块化，这是一件非常好的事情，因为如果做得正确，可以增强内核的扩展性和安全性。当许多人从事内核的开发工作时，他们的代码都拥有自己的特点和模式，互相之间的使用只限于互相调用功能。

只有两种本质的情况决定代码是否是机器相关的。所有的汇编代码和所有直接访问系统硬件的代码都是机器相关的。大多数情况下，直接访问硬件的代码都存在于设备驱动程序中。对内核来说，这部分代码包括使用CPU寄存器的函数和所有在内核中的设备驱动程序。通常的系统驱动程序一般被认为是系统相关的，但是也有例外。例如虽然MINIX 3中使用“log”和“memory”驱动程序，但是它们不直接访问硬件，所以它们可以被被认为是硬件无关的。为了保持代码的独立性将要创建更多的文件，为了组织它们而使用了新的目录结构。可以在MINIX 3原来的目录结构中为每个体系结构通用的源文件建立相应目录，例如在内核根目录`< ./kernel/ >`中新增目录项如图2.1:

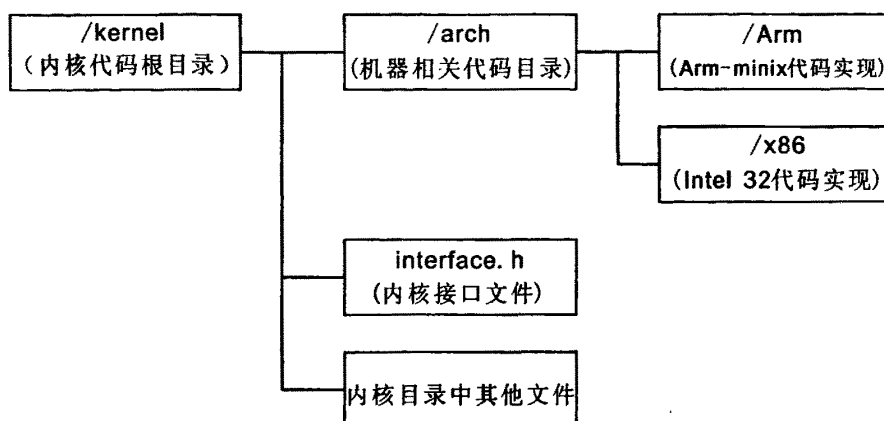


图2.1: 重建内核目录结构

其中，我们将新的项目命名为arm-minix。

更新ARM机器相关代码中的一个文件就需要重新编译在`< ./kernel/arch/arm-minix/ >`目录下的代码，而且要重新链接内核进程。重新编译后要新建一个新的库，包含arm-minix内核的机器相关代码。这个库为内核的机器不相关代码提供所有的访问硬件的入口（函数）。

为了让机器相关代码和机器不相关代码链接起来以后能够正常工作，必须要有一个接口。这个接口描述了机器相关代码库中每个函数的输入参数和运行结果。应该称之为“内核硬件接口”（Kernel Hardware Interface, KHI），是处于内核中机器相关代码和机器不相关代码之间的一个接口，前者负责与硬件打交道，后者负责与系统进程的交互。

为了定义和识别接口函数，我们使用了一个“驱动程序模型”。使用这种方式的原因是在ANSI C中这是一种建立驱动程序前端的默认方法。更一般地，这种编程结构被用来将代码封装进一个逻辑模型或对象中。驱动程序模型的使用将在下一小节讨论。

如果有一种想法认为CPU是一个一般设备，那么就得出结论：直接访问CPU寄存器的只有设备驱动程序。这就意味着一个完备系统所有的机器相关代码都应该位于设备驱动中。使用这样一个提供所有CPU功能的设备驱动程序模型隔离了CPU，从而达到了可移植的目的，它也给了整个系统代码一个统一的编程风格。要完成上面这些工作，无疑增加了代码编译的复杂度，但是使用Makefiles将会使得系统编译变得简单一些。

2.1.2.1 创建可移植代码的3种方式

有三种主要的方式创建可移植代码，首选的方法列在最前面：

1. 创建附加文件，重新实现函数
2. 使用类型重定义
3. 使用编译程序指令

下面分别来介绍这三种方式。

• 第一种方式：

使用第一种方式，编程者至少要建立两个对象文件。一个包含MDC，一个包含MIC。驱动程序实际上就是包含MDC的文件。每一次向一个新的体系结构的移植都会导致在操作系统中新增加一个MDC文件。定义在一个普通头文件中的接口将给出一个MIC对象，这个MIC对象使用驱动程序，访问MDC对象提供的功能。这是首选的方式，因为它提出了将驱动程序分离到一个文件中。本质上说，这种方法是在重新定义函数。下图为一个例子：

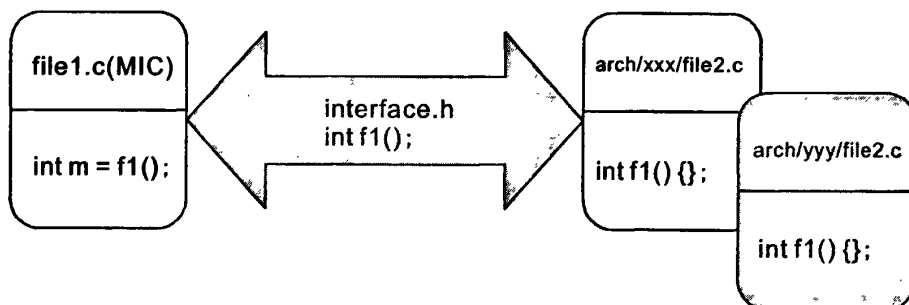


图2.2：多体系结构为接口文件提供函数实现

在图2.2中，接口文件会包含一个函数原型的列表。当为某一个体系结构编译代码的时候，通常需要文件`< file1.c >`和编译后的目标文件`< ./file1.o >`。有多少种体系结构，`< file2.c >`就有多少个版本，但只有目标体系结构的这个版本会被编译，为接口头文件`< ./interface.h >`中的函数原型提供函数实现。后面的两个源文件是为一个驱动程序存在的，由接口定义，被文件`< file1.c >`使用。

• 第二种方式：

第二种方式需要细心地编程，但是却能实现功能强大的类型重定义。这种重定义不包括重新定义编译器使用的简单的或标准的类型，只包括由C语言的关键字“struct”定义的高级数据类型。这种数据结构可以包含简单数据成员，也可以包含高级数据成员。

为了提高效率，这种方法要求在这些重定义的数据结构中相应的成员拥有相同的名字和语义。当在手写汇编语言文件中使用这些数据结构时，它们在内存中的布局就会很重要了。这时重新定义的结构体中，“关键”成员必须是相同的排列顺序，否则一样的偏移量指向的是结构中的另一个成员。当这些数据结构只在C代码中使用时，数据成员的布局会在每一次编译时由编译器计算的。

arm-linux的进程表表项包含两个关键（高级）结构体，它们定义了当进程被轮换出CPU之后处理器的状态。第一个结构体“stackframe_s”，包含进程CPU寄存器的值，栈指针、程序计数器和通用目的寄存器的序列。另一个“segframe_t”包含段指示符，定义了进程的内存设置。栈框结构体的一些成员被一些特定的系统调用使用，例如“do_exec”，所以这些成员需要一个通用的名字。幸运的是每一个系统都有栈框的成员所指向的组件，所有的系统都有程序计数器，栈指针和一些类似的东西。必须提供这些通用名以便于机器不相关部分代码在需要时可以找到并修改它们。大部分的通用寄存器也需要被存储，而且由于每一个体系结构需要的空间不一样，所以要在进程表的表项中留出足够的空间。而重定义的结构体“stackframe_s”为ARM中相应的寄存器留出了空间，所以使用类型重定义将会“自动地”在进程表中创建空间。下面就来具体说明如何在编译中重定义目标体系结构的类型。

文件（大多数是头文件，但也可能是任何文件）可以以两种形式被包含：相对当前源文件的位置和在可被搜索到的目录列表中。在源文件中被包含的文件用引号或者尖括号标明。我们不提倡相对包含的方式，因为这种方式总是要求我们写出头文件的路径，从而使对一个文件的包含产生了局限性。

大多数的编译器使用一个编译选项来添加搜索列表的目录。通过这种方式，我们就可以根据编译对应的目标体系结构而选择包含不同的头文件，这些文件中包含了相应体系结构的类型的重定义。我们就用这种方式实现对不同体系结构的类型重定义的头文件的包含，而保持源文件中原来的include指令不变。这样做的优点是我们不需要对源文件的代码做出相应的修改。这种方法需要一个体系结构单独的“include”目录外加一个标准的系统“include”目录。只有目标体系结构的目录被添加到这个列表。如果出现如下这种情况，被包含的两个文件都定义了类型a，那么将得到一个类型重定义的警告，而这个问题是很好解决的。

下一步要做的就要对所有的文件都需要重定义一个数据结构类型。

GNU C编译器用选项“-I”用来将一个目录加入目录列表，编译器在这个列表中搜索包含的文件。所以当编译Intel 32位体系结构时我们可能要在编译命令上加入“-I ./minix/arch/i386”。在下图中，在文件< file1.c >中的指令“#include < atypes.h >”将被处理为文件< ./minix/arch/i386/atypes.h >。对于ARM的编译我们在编译指令上加入“-I ./minix/arch/arm-minix”，文件将被处理为< ./minix/arch/arm - minix/atypes.h >。见图2.3。

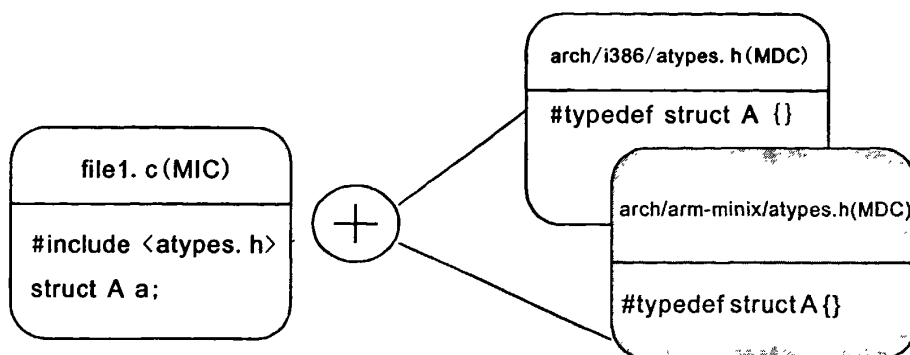


图2.3: 多体系结构提供的同一个数据结构类型

这些目录不能在标准（或者系统）include目录中，例如目录`</usr/include>`，因为这样会引起搜索冲突。大多数的编译器使用第一个搜索到的相匹配的文件。这样做可能导致上面提到的错误很难被发现，因为编译器不会给出任何警告信息。

- 第三种方式:

最后一种方式把需要编译的代码用编译程序指令做一些修改，比如指令`#ifdef`，`#if`和`#endif`。这些代码都是全局定义，上述指令告诉编译器相应代码应该被包含或者被忽略。

看图2.4中的代码，宏定义`MACHINE`告诉我们这段代码是为IBM PC或ARM编写的，这段代码保存在一个全局文件中，所以所有的源文件都共享相同的定义。它必须是全局的，以保证`MACHINE`的定义在一个项目的每一个文件中都是一样的。

在图2.4中我们用`"MACHINE"`指明了编译的目标机器。

```

1.#define MACHINE      ARM
2.
3.typedef unsigned int reg_t
4.
5.#if( MACHINE == ARM )
6.
7.
8.
9. #endif
10.
11.
12.#if (MACHINE == IBM_PC )
13.
14.
15.
16.#endif
    
```

图2.4: 编译程序指令举例

不使用编译程序指令不会有任何问题,但是这样有一个巨大的缺陷,那就是导致了系统的不可扩展性。当我们将操作系统向另一个体系结构移植时,我们需要附加一个“`#ifdef`”模块。这样就使得每一次移植都需要更新源代码文件,这不是我们想要的。理想情况是尽可能地创建一个新的移植而不触动原来的源文件。

在上面这个例子中在两条指令之间的代码行数很少,但是大多数情况下代码行数会在一个正常的数量。但是使用这种结构又会带来另一个缺陷,那就是失去了可读性,这个源代码应该具备的很重要的性质。当在代码模块之中使用编译程序指令时,就会失去可读性。可以假设一下,当代码在执行的时候在编译程序指令之间的部分只是从屏幕上滚动而过,根本就没有经过编译。在代码模块中应该避免这样,因为这样会导致死代码。新接触代码的程序员可能需要很困难、很长的一段时间去读懂这段代码。

如果这样使用是不可避免的,那么应该有一种方式使得这种代码占用的行数不多。如果可能的话划分这些文件,然后使用条件包含指令包含这些文件。有几个条件包含指令就新建多少个文件,然后把条件包含指令之间的代码写入相应的新文件,然后再代码出现的地方用“`#include`”包含这个文件。

上面这些改变的影响应该是局部的,并且应该能被清楚地看到。例如机器相关模块可能还涉及其它设备,或者每个模块需要不同的参数。大多数情况下,这些代码的影响只有很少的几行。这时,可以使用编译程序指令编写上述代码。

但是为了完整性,也有一些例外,例如库的头文件也使用编译程序指令,而这些指令的影响不止少数的几行代码。这些指令通常用来定义基本系统类型。

2.1.2.2 从现存的文件创建可移植代码

当你需要去读原始文件然后判断“函数”或者“定义”是否是机器相关的时候,是有一些技巧的。在大多数情况下,要决定带有机相关选项的管理代码是很容易的,这些代码用来初始化或者通信。第一件要做的事情是删除所有已确定的机器相关代码(MDC),剩下管理代码(MIC)。然后编译和链接管理代码,编译器起将会列出未定义的依赖关系代码、函数和标识符。这就给了我们一个提示:MDC提供了什么。

接下来用删除的机器相关代码创建一个新的源文件,标识符和函数都是原始的。然后在在一个联结MDC和MIC的头文件中为函数定义接口。在这里最好避免使用全局标识符和定义,因为不同体系结构之间,设备D的端口号X的“全局”定义可能完全不同。例如一个体系结构可能没有设备D,或者这个端口可能在设备M中等等。一个好的接口不应该随着体系结构的改变而变化,如果不是这样,那么它就是不好的。接口的更新不应该导致原来的体系结构的更新,而应该只是忽略这个更新而已。只要找到在编译器指令之间的代码,就像图2.4中展示的那样,就应该把这些代码移到新的机器相关代码源文件中去。注意如果存在这些机器相关代码的有依赖关系的代码,而这些代码也不在系统库中,那么它们也应该被转移。

最后,所有的源文件都应该能互不影响的编译。唯一共同的东西就是使用指令“`#include < interface.h >`”包含的接口文件。只要按照规则来做,这样建立的可移植

代码应该和一部分一部分建立起来的可移植代码没有什么不同。最后，我们把管理代码放在“左边”（MIC），把机器相关代码放在“右边”（MDC）。

2.1.2.3 驱动程序编程模型

驱动程序模型是用来保证内核内部的驱动程序的机器相关代码和机器无关代码的分离，我们也使用它来使CPU从内核中独立出来。这里使用的（低等级）驱动模型符合以下准则：

1. 便于将来移植
2. 逻辑上的分块
3. 对代码的目的有一个清楚而并非模糊的认识

尽管这两部分代码有可能被认为是密不可分的，但是上述的这个规则的集合可以保证将它们分开。就像前面说的，代码分为两种：机器相关代码和机器无关代码，这正证明了上面列出的第二个规则。最终一个真正的驱动只包含通过驱动程序接口来访问的机器相关代码，这样就做到了逻辑上的分块。

在内核中使用驱动程序模型的目的是使把与内核不相关部分（机器相关代码）独立出来。而与内核相关部分指的是：内核如何使得内存能够被访问的方法，调度进程的方法和开启或关闭中断的方法。“机器相关代码”只存在于代码实现的最后，例如，设置MSR（机器状态寄存器）标志位关闭终端中断时，最后一个被调用（有时在汇编代码中）的函数。

解释驱动程序模型和移植现有代码的最好方式是举一个例子。我们把MINIX 3时钟系统作为例子。这段代码最初是为Intel体系结构编写的，它使用端口I/O访问位于8259A中断控制器中的时钟硬件。这个控制器（或者它的兼容产品）在每一个IBM PC兼容机中都能找到。文件`<./kernel/clock.c>`为MINIX 3提供了时钟任务，而且还保持对定时器（把关计时器）的跟踪。这个文件由两个函数集合组成。一个集合负责硬件I/O，另一个负责管理（包括时钟中断处理程序）。

时钟硬件在三种情况下被软件访问：

- 初始化时钟硬件
- 读取目前的时钟总数
- 停止时钟

管理集合中的函数被进程用来建立和使用把关计时器。这个文件也提供一个中断处理程序，它在每一个时钟嘀嗒的时候被中断驱动程序调用然后更新计时器。这个文件的另一个作用是：完成进程计帐。在文件`<clock.c>`中所有代码都是C代码，但是其它体系结构中不一定使用8259A中断控制来形成一个时钟嘀嗒，所以即使编译通过，也不一定能

正常工作，这导致了要重写`< clock.c >`文件。我们从文件中硬件被访问的三种情况开始工作，这三种情况需要用驱动程序模型进行分离。

驱动模型分离了这几部分的代码，还用预定义好的函数给出了接口的定义，这个接口在上述功能的最后一步被用来访问硬件。

2.1.2.4 时钟系统举例

驱动程序需要的文件数目由驱动程序模型变化而决定。实际上，驱动程序只有“两个”文件。头文件定义接口，给出函数原型；源文件提供函数的实现。比较大的驱动程序需要附加文件。原始的时钟文件（`< ./kernel/clock.c >`）包含时钟任务、计时器管理、进程计帐和硬件访问代码，这些全在一个文件中。

示例的时钟系统将由3个文件组成：

- `< kernel/clock.c >` 机器无关代码
- `< kernel/interface.h >` 给出时钟驱动程序接口
- `< kernel/xxx/clock.c >` 机器相关代码（驱动程序）

修改后的`< ./kernel/clock.c >`文件将仍然包含时钟任务、计时器管理和进程计帐代码，这些是MINIX 3不可缺少的部分。接口文件`< ./kernel/interface.h >`包含一些函数在一个结构体声明中，这个结构体是为了最大化安全性和隔离性。访问硬件的函数将被转移到一个新文件`< ./kernel/arch/xxx/clock.c >`中，这个文件给出了在接口中描述的函数的实现。定义的接口声明了MINIX需要什么功能，通过函数访问硬件来实现。应该清楚的是访问硬件的部分代码应该被转移到文件`< ./kernel/arch/xxx/clock.c >`中。

在`< ./kernel/interface.h >`中一个为时钟驱动程序定义的接口可能看起来是这样的：

```
1. typedef struct if_clock_s {
2.     int      (*init)(void);    /* 初始化时钟硬件 */
3.     int      (*stop)(void);    /* 停止时钟 */
4.     int      (*start)(void);   /* 开始时钟 */
5.     clock_t  (*read)(void);    /* 读取目前的记账 */
6. } clock_interface_t, if_clock_t;
```

图2.5：在文件`< interface.h >`中，时钟驱动程序可能的接口文件

在图2.5中，接口的函数原型被定义在一个结构体“if_clock_t”中，这组函数是属于同一个驱动程序的。在结构体中的函数被调用的方式给编程者提供了一个清晰的提示：他正在使用一个机器相关的函数。下面将会看到这些函数是如何从一个数据结构中被调用的。

细心的读者可能会说给所有的函数加上前缀，就像“clock_init ();”和“clock_stop ();”，这样做可以和给时钟驱动程序的函数分组得到一样的效果。可以这样做，但是就达不到

隔离代码的目的了。这样的话一个文件可以在不知道接口定义的情况下轻易地访问驱动程序的函数，需要的只是函数的名字、正确的参数列表和这些函数可以自由地被在程序中使用。将对函数的访问编组封装在一个结构体中给出了使用驱动程序函数的一个额外步骤。这样确保了实现这些函数的源文件必须知道这个结构体，并且包含接口定义。

下面继续时钟系统的例子，从一个机器相关代码的片段开始。

```

1.FORWARD _PROTOTYPE ( int init , (void) );
2.
3.PUBLIC const if_clock_t Clock = {
4.    info:info ,
5.    init:init ,
6.    start:start ,
7.
8.};
9.
10.PRIVATE int init (void) {
11.
12.}
13.PRIVATE void start(void) {
14.
15.}

```

图2.6: 时钟驱动程序中机器相关代码的实现

图2.5中的接口对“左”、“右”两边的代码都是很重要的。图2.6是文件<./kernel/arch/xxx/clock>中的一段。请注意第3, 10, 13行private和public的使用。所有的函数和文件全局变量在机器相关代码文件中应该是私有的，所以它们只有通过公有数据结构的成员或者接口才能够被访问。这里将结构体定义成一个“常量”（第3行），这样做的话函数引用在编译结束以后就不能改变了。在编译操作系统内核时，这是一个安全性的建议。

在机器不相关代码中用以下方式调用这些函数：

```

1.#include "kernel/interface.h"
2.
3.extern const if_clock_t clock;
4.
5.Clock.init ();
6.
7.Clock.start ();

```

图2.7: 时钟驱动程序中机器相关代码的调用

看图2.7中的3行和第5行，第3行将机器不相关源代码连接到在图2.6中定义的结构体，第5行是对图2.6中的初始化函数的调用。来看函数的调用形式，“Clock”是函数的前缀，大体上看起来与面向对象的编程很相似。这种调用形式也告诉用户这个调用的是时钟系统的（在这个例子中）机器相关部分。注意这段代码在第1行包括了接口的定义，否则不可能能够访问这些函数。

有了驱动程序的编程模型，将操作系统移植到一个新的体系结构就变得相对容易了。所有需要重写的“内核”相关文件被放在目录树`<./kernel/arch/xxx/>`中。对应接口编写驱动程序，MINIX应该就能够运行。在这个模型中的困难之处是使定义的接口不光只能在这个系统中工作，而是能在所有的系统中工作。不幸的是我们只能知道在移植了很多系统之后它们工作得有多好。保持接口的最小化应该对保持不兼容性的最小化有所帮助。

2.2 MINIX 3中的汇编语言

如果要对MINIX 3有一个深入的理解，你需要有能力去阅读MINIX 3的汇编代码，这些代码集中在两个文件`mpx386.s`和`klib386.s`中。MINIX 3中的这些汇编代码使用的是特殊的汇编语言，所以要阅读内核底层代码必须能读懂这些汇编代码。

MINIX 3中编译C语言文件和汇编语言文件都使用自带的ACK ANSI C编译器`cc`，它是ACK（Amsterdam Compiler Kit）的一部分。源代码的编译也是使用的这个编译器，所以源代码中汇编语言的语法就是这个编译器的语法。ACK汇编语言的特别主要体现在符号、结构和伪操作等方面。

MINIX中的一个汇编文件包括4个部分（section）：

1. 代码部分（text）：存放代码，编译时通常放在代码段中。
2. rom数据（rom）：存放可以放在只读存储器rom中的数据。这一特性通常并没有使用。
3. 数据（data）：存放可读写的的数据，比如私有变量和堆栈等。
4. 全局数据（bss）：存放全局变量和地址，这些数据首先被初始化为0。

每个部分用伪指令`“.sect”`来声明，这四个部分的顺序是固定的，也就是说第一个出现在文件中的部分是代码部分，第二个是rom部分，依此类推。如在图2.8中，a、b、c、d是段的名称，为任意字符串，它们依次为代码部分、rom数据部分、数据部分和全局数据部分。

1. `“.sect a`
- 2.
3. `“.sect b`
- 4.

```
5. .sect c
6.
7. .sect d
8.
```

图2.8: 汇编语言文件结构示例

下面针对ACK汇编语言与Intel X86汇编语言在语法上作了一个比较, 仅针对其不同与特别之处进行说明。

表达式: 优先级的改变用方括号[], 而圆括号用来表示寄存器中的内存地址, 与X86汇编语言正好相反。

注释: 用感叹号“!”来标明行注释。

16/32伪操作数前缀: 当无法判断操作数是16位或32位时, 需要用o16或者o32来表明操作数是16位或32位。例如:

```
O16 mov dx, SS_SELECTOR
```

16/32位地址前缀: 当无法判断地址是16位还是32位时, 需要用o16或者o32来表明操作数16位或32位。

伪指令:

.extern: 定义外部文件中的全局变量、全局函数和例程。

.define: 定义本文件中, 可以被其它文件使用的全局变量和全局例程。

.data1, .data2, .data4: 分别定义了一个、两个、四个字节的的数据。例如: .data2 255, 定义一个两字节的数据。

.ascii, .asciz: 定义字符串。两者不同的是, asciz将在定义的字符串后自动加一个0。例如: .asciiz "Hello world!"

.align: 用于将当前地址对齐到指定整数的倍数, 空位补零。例如: .align 4。

.space: 写入指定个数的0。例如: .space 64, 从目前地址开始写入64字节个0。

.common: 将全局变量初始化为0。例如: .common number, 8, 将全局变量初始化为8个字节的0。

.sect: 定义一个部分

另外还要注意的是在汇编语言中调用一个C函数时, 函数名前要加下划线"_", 并且函数的参数通过压栈来传递, 返回值通过寄存器eax来传递。如果要在C程序中调用汇编例程, 例程名前同样要加下划线"_", 并且在汇编源程序中还要用.define来声明。

最后还要说的是ACK汇编语言中的宏与C语言中的宏的使用是一样的。用反斜杠"\\"来连接需要换行的宏。

2.4 应用程序二进制接口

下面简单的介绍一下“ABI”或者Application Binary interface（应用程序二进制接口）。ABI不应该和API（Application Programming interface）混淆。在等级上来说，ABI比API更接近硬件，这样就导致变更平台时API不用改变。API是标准，系统开发者选择它用来让用户程序可以使用系统服务。例如，提供一个（库）函数的集合，用来打开、关闭、读、写文件。对MINIX来说，API由POSIX标准定义，这是类UNIX环境通用的。许多系统提供一样的API，底层的硬件可能不同，例如RISC和CISC，但是一样的源程序都可以编译并运行。

ABI掩盖了各种细节，例如：调用约定（控制着函数的参数如何传送以及如何接受返回值）；系统调用的编码和一个应用如何向操作系统进行系统调用；以及在一个完整的操作系统ABI中，对象文件的二进制格式、程序库等等。一个完整的ABI，像Intel二进制兼容标准(iBCS)，允许支持它的操作系统上的程序不经修改在其他支持此ABI的操作系统上运行。其他的ABI 标准化细节包括C++名称修饰和同一个平台上的编译器之间的调用约定，但是不包括跨平台的兼容性。

ABI定义了高级语言和其生成的汇编代码之间的关系。编译器生成汇编代码的方式定义了（硬件）CPU寄存器如何使用。ABI 是软件和硬件之间的桥梁。在这篇论文中是C代码和由它生成的ARM汇编代码之间的关系。这个定义包括调用规范和如何使用栈。例如一个函数的活跃记录是什么样的，哪个寄存器是可变的（在函数调用之间），哪个寄存器应该保持不变。

要知道编译器生成了什么类型的汇编代码，最简单的方法是创建一个“空”程序（只有函数体），并且跳过链接过程。这一个学习基本汇编代码的方法，可以作为学习和理解ABI的奖励。要同时学习目标系统汇编语言和理解ABI，这是一个强力的方法，而且在开移植系统时候也会用到这种方法。尤其是当你不熟悉目标体系结构的时候，这种方法就尤其显得有价值。

如果系统在ABI级上兼容，那么就意味着一个在A系统上编译的可执行程序可以在B系统上运行而不用重新编译，并且 $OS(A) \neq OS(B)$ 。并且预编译的库也可以混用。但是在大多数情况下，这种情况只发生在不同版本的操作系统向后兼容的情况下。

C代码和手写汇编代码一起使用来开发一个操作系统，所以你需要知道操作系统生成了什么。在上下文切换代码和信号处理程序中会经常用到这些概念。最后，要成功移植一个操作系统，熟悉和理解目标体系结构汇编代码和使用ABI是起决定性作用的。

第3章 MINIX 3内核分析及进程调度部分代码的移植

3.1 MINIX 3内部结构

熟悉MINIX的人知道，MINIX实行的是微内核的设计。意思就是内核只实现最少的功能，剩下的系统进程都被放到用户空间去了。这样就增加了系统的可靠性，因为关键部分代码的减少。他使用隔离的进程完成各种系统任务。这符合模块化的设计。系统各部分间的通信使用消息。非关键进程，例如文件系统和进程管理器是运行在用户空间。在MINIX3中驱动程序也运行在用户空间。

首先我们大致了解一下MINIX 3系统，MINIX 3被组织成4层，每一层执行一套定义明确的功能。这4层结构如图3.1所示：

层次

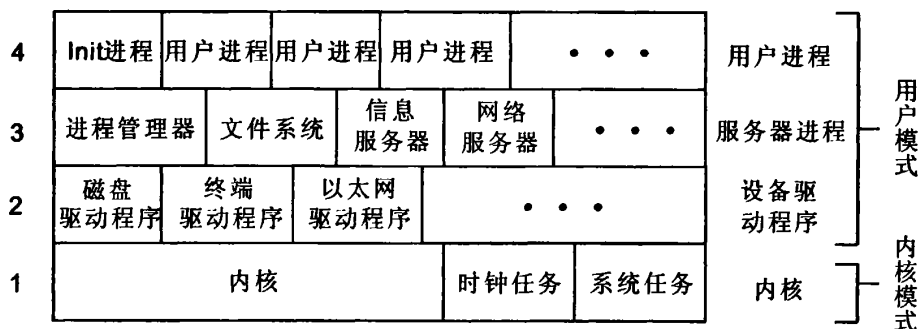


图3.1: MINIX 3操作系统内部结构

内核最底层负责进程的调度，进程在运行态、就绪态和阻塞态之间的转换，以及进程间消息的处理。

有三个进程，称作任务，运行在核心空间，它们是IDLE，时钟任务和系统任务。

时钟任务（clock task）实际上是一个I/O设备驱动程序，但是与普通的设备驱动程序不一样，用户空间进程无法直接访问，只能通过内核提供的接口进行访问。它包括一个主循环用来接收消息，一个每秒钟激活60次的中断处理程序和一些支持函数。它的主要作用是：

1. 维护日常时间
2. 防止进程运行超时

3. 审计CPU的使用
4. 处理用户发出的alarm系统调用
5. 为系统的某些部分提供看门狗时钟
6. 执行测试统计, 监视并获取统计数据

系统任务代替设备驱动程序和服务器进程执行I/O操作, 即向设备驱动程序和服务器提供一组内核调用。在MINIX 3用户空间的进程发出的系统调用被转换成消息发往服务器进程, 服务器再调用系统任务提供的内核调用来完成用户的请求。

IDLE进程拥有比用户进程还低的优先级, 在系统中没有任何进程运行时被调度。

运行在用户空间的设备驱动程序和服务器被称为系统进程, 其中必不可少的两个服务器进程是: 进程管理器和文件系统服务器。

进程管理器用来处理与进程管理相关的系统调用, 例如fork, exec等, 并且也负责“存储管理器”的工作。

文件系统服务器负责处理一切与文件相关的系统调用。

要将MINIX 3完整地移植到新的体系结构上, 需要移植的进程有下面几个:

- IDLE idle任务
- CLOCK 时钟任务
- SYSTEM 系统任务
- HARDWARE 伪进程, 内核任务
- PM 进程管理器
- FS 文件系统服务器
- RS 重生服务器
- MEMORY 内存驱动程序
- LOG 日志驱动程序
- TTY 终端驱动程序
- INIT 第一个用户进程

3.2 MINIX 3内核代码的组织

第一个被移植的进程应该是内核。MINIX 3的原始内核位于一个平面目录树中, 这种结构没有体系结构子目录。应该对内核代码进行一些组织, 隔离只包含IBM PC兼容机

的头文件，使得这些代码可以在多种体系结构上使用。编译程序指令就用来包含这些代码模块或文件的。尽管一些（旧的）体系结构也支持原始MINIX 3原始内核，但是这个内核是以IBM PC兼容机为目标平台的。

在这篇论文中，对原始的MINIX源代码树做了一些转变。新的内核树被扩张并分为两个部分，根目录是第一部分并且包含所有的机器不相关代码；第二部分从目录`<./kernel/arch/xxx/>`开始，并且包含所有的机器相关代码。ARM体系结构的机器相关代码部分应该放在目录`<./kernel/arch/arm-minix/>`中。

接下来图3.2显示的是MINIX 3的内核树，它是Intel 32位体系结构的。

```
./kernel
|--system
|  |--do_abort.c
|  |--
|--Makefile
|--clock.c
|--config.h
|--const.h
|--debug.c
|--debug.h
|--exception.c
|--glo.h
|--i8259.c
|--ipc.h
|--kernel.h
|--klib.s
|--klib386.s
|--klib88.s
|--main.c
|--mpx.s
|--mpx386.s
|--mpx88.s
|--priv.h
|--proc.c
|--proc.h
|--proctect.c
|--proctect.h
|--proto.h
|--sconst.h
|--start.c
|--system.c
|--system.h
|--table.c
|--Type.h
```

```
--utility.c
```

图3.2: MINIX 3的内核树

我们在内核根目录下新建一个机器相关目录`< arch/arm >`和`< arch/x86 >`，分别用来存放分离后的不同体系结构的相关代码。改变后的内核目录结构如下：

```
./kernel
|--arch
|  |--arm
|  |  |--
|  |  '--x86
|  |--
|--
|--interface.h
|--
|--glo.h
|--
```

图3.3: 重建后的内核树

为了移植现有的内核代码，所有的机器相关部分必须首先被找到。这些工作可以用前面介绍过的方法（2.1.2.2小节“从现存的文件创建可移植代码”）完成，这样就可以对编译时依赖的定义建立对应的文件。下面列出从`clock.c`文件中删除明显的机器相关代码后的缺失符号：

1. function 'outb'
2. 'TIMER_MODE' undeclared
3. 'TIMER0' undeclared
4. 'PPRT_B' undeclared
5. function 'inb'
6. COUNTER_FREQ
7. LATCH_COUNT
8. SQUARE_WAVE
9. TIMER_COUNT
10. TIMER_FREQ

11. CLOCK_ACK_BIT

图3.4: 文件clock.c中的机器相关代码

并不是所有的机器相关代码都能通过编译时列出的“缺少的符号”找出来。这个列表只列出了大部分我们需要的文件，每一个汇编文件默认都是机器相关的。其中6-10行是没有移动的，但是已知是机器相关的符号和定义。

要对文件做出的更改远不止上面列出的这些，还有对现存头文件的更改以及新头文件的建立；符号和定义的移动、不再需要的及位置不对的函数原型都是导致做出更改的原因。1-6行是被编译器所接受的定义，它们只是定义，但是它们放错了位置，因为这些定义被用来初始化时钟硬件和与时钟硬件通信。它们对时钟管理不起什么作用，所以这些行应该被转移到机器相关代码文件中。应该将文件`< clock.c >`被分为两个文件，`< ./kernel/clock.c >`和`< ./kernel/arch/arm/clock.c >`，保持一样的名字把他们的目的联系了起来。

3.3 内核驱动程序模型

我们想将内核分为MIC和MDC两部分。就像早先提到的，我们以将所有的系统部分定义为设备来达到这个目的。然后为每一个设备建立驱动程序，使用驱动程序模型，建立我们需要的MIC/MDC的分离。我们还需要建立内核内部的设备驱动程序，这种方式通常被用在中断控制器硬件上，大多数情况下还包括时钟硬件。时钟驱动程序由时钟任务设置和使用，所以事实上它不是一个在内核内部的驱动程序，但是内核却直接调用时钟驱动程序，用来在关闭时停止时钟。

将CPU看作一个设备的观点是比较新的。对MINIX 3来说，CPU被逻辑地分为两个设备，并且被两个驱动程序驱动，一个内存驱动和一个系统驱动。内核总共使用4个接口用来访问系统硬件。

1. System interface
2. Memory interface
3. Interrupt interface
4. Clock interface

图3.5: 内核接口

图3.5出的接口定义在`< ./kernel/interface.h >`中。移植系统遇到的一个问题是搜索所有系统相关函数，并做一个这些函数的列表，并且决定哪些函数放在接口文件里面，哪些函数放在接口文件外面。最终，文件`< interface.h >`包含需要向新体系结构移

植的函数的列表。每一个体系结构都应该能提供接口所需要的这些功能。通常有这种可能性，一个新的系统可能需要一些函数，而这些函数并不存在，或者函数已经定义了，但是需要更新。到最后，移植到的体系结构越多，接口也就越健壮。

文件< ./kernel/interface.h >中放置图3.5中所列出的4个内核接口。下面是文件中的数据结构的列表：

//内存驱动程序接口

```
typedef struct if_memory_s {
    const char*    info;
    int            (*init)      (void);

    int            (*alloc_segments)    (struct proc* p);

    vir_bytes      (*alloc_remote_segment) (u32_t* selector, segframe_t* s,
        int index, phys_bytes phys, vir_bytes size, int privilege);

    void            (*copy_message) (int source, struct proc* p_src, message*
        m_src, struct proc* p_dst, message* m_dst);

    void            (*phys_copy)      (phys_bytes src, phys_bytes dst,
        phys_bytes count);

    void            (*phys_memset)    (phys_bytes src, u32_t pattern,
        phys_bytes count);

    phys_bytes      (*seg2phys)      (segdesc_t segdesc);

    phys_bytes      (*umap_local)    (struct proc* rp, int seg, vir_bytes
        vir_addr, vir_bytes bytes);

    phys_bytes      (*umap_remote)   (struct proc* rp, int seg, vir_bytes
        vir_addr, vir_bytes bytes);

    void            (*outb)          (volatile u8_t* a, u8_t v);
    void            (*outw)          (volatile u16_t* a, u16_t v);
    void            (*outl)          (volatile u32_t* a, u32_t v);
    u8_t            (*inb)           (volatile u8_t* a);
    u16_t           (*inw)           (volatile u16_t* a);
    u32_t           (*inl)           (volatile u32_t* a);
    void            (*be_outw)       (volatile u16_t* a, u16_t v);
    void            (*be_outl)       (volatile u32_t* a, u32_t v);
    u16_t           (*be_inw)        (volatile u16_t* a);
    u32_t           (*be_inl)        (volatile u32_t* a);
```

```

void      (*le_outw)  (volatile u16_t* a, u16_t v);
void      (*le_outl)  (volatile u32_t* a, u32_t v);
u16_t     (*le_inw)   (volatile u16_t* a);
u32_t     (*le_inl)   (volatile u32_t* a);
void      (*insb)     (volatile u8_t* addr, void* b, size_t c);
void      (*insw)     (volatile u16_t* addr, void* b, size_t c);
void      (*insl)     (volatile u32_t* addr, void* b, size_t c);
void      (*outsb)    (volatile u8_t* addr, void* b, size_t c);
void      (*outsw)    (volatile u16_t* addr, void* b, size_t c);
void      (*outsl)    (volatile u32_t* addr, void* b, size_t c);
} if_memory_t;
//系统驱动程序接口
typedef struct if_system_s {
    const char*   info;
    int           (*init)      (void);
    void          (*syscall)   (void);
    void          (*intr_disable) (void);
    void          (*intr_enable) (void);
    int           (*locked)    (void);
    void          (*shutdown)  (void);
    void          (*reset)     (void);
    void          (*idle)      (void);
    void          (*sputc)     (int c);
    void          (*read_tsc)   (unsigned long* high, unsigned long* low);
    void          (*restart)    (void);
} if_system_t;
//中断驱动程序接口
typedef struct if_interrupt_s {
    const char*   info;
    int           (*init)      (void);
    int           (*get_vector) (void);
    int           (*set_vector) (int source, int priority, int vector);
    int           (*ack_vector) (void);
    int           (*enable)    (int source);
    int           (*disable)   (int source);
} if_interrupt_t;
//时钟驱动程序接口
typedef struct if_clock_s {
    const char *info;
    int          (*init)(void);
    void          (*stop)(void);
    void          (*start)(void);
    u32_t         (*frequency)(void);

```

```
clock_t    (*read)(void);
} if_clock_t;
```

图3.6: 接口文件< ./kernel/interface.h >中定义的内核接口

每一个要移植MINIX 3的体系结构都应该提供这些功能。这些类型定义需要被内核MIC和MDC知道。MIC将使用列出在时钟接口中的函数去初始化和管时硬件。MDC使这些函数发生实际作用。列在这个接口中的每一个函数使用机器相关代码，例如端口号、寄存器地址、或者一个只在目标体系结构上可见的驱动程序。通过内核代码识别系统相关代码或者调用是简单的。文件< ./kernel/glo.h >包含如下的声明：

```
1. extern const if_system_t      System;
2. extern const if_memory_t      Memory;
3. extern const if_clock_t       Clock;
4. extern const if_interrupt_t    Interrupt;
```

图3.7: 内核中硬件访问的入口点

以上这些定义是对接口文件< ./kernel/interface.h >中包含函数列表的数据结构的引用。内核调用都通过这4个数据结构来调用。例如用来初始化时钟的调用如下：

```
Colock.init();
```

最后要说的是，驱动程序模型对原始内核MIC的影响是很小的，它只要求对原始函数的重命名。

3.4 操作系统中的进程调度

中央处理器（CPU）是整个计算机系统的核心资源，在多进程的操作系统中，进程数往往多于处理器数，这将导致各进程互相争夺处理器。进程调度对系统功能的实现及各方面的性能都有着决定性的影响，其实质就是把处理器公平、合理、高效地分配给各个进程。调度是实现多任务并发执行的必要手段，不同的操作系统有着不同的调度目标。在传统的Unix类分时系统中，保证多个进程公平地使用系统资源，提供较好的响应时间是调度的主要目标；而在强实时操作系统中，总是优先级高的任务优先获得处理器的使用权。

当计算机是多道程序设计系统时，通常就会有多个进程竞争CPU。当多个进程处于就绪状态而只有一个CPU时，操作系统就必须决定先运行哪一个进程。在操作系统中做这种决定的部分称为调度器（scheduler），它使用的算法称为调度算法（scheduling algorithm）。

时钟中断给操作系统提供了一个判断当前进程是否运行了足够长时间的机会，根据如何处理时钟中断，可以把调度算法分为两类：非抢占式调度算法（non-preemptive

scheduling algorithm) 和抢占式调度算法 (preemptive scheduling algorithm)。非抢占式调度算法挑选一个进程运行, 直到该进程阻塞或者自愿退出。而抢占式调度算法挑选一个进程运行一段时间, 这段时间的最大值是固定的。在进程运行完这段时间以后, 即使进程不愿意阻塞或结束也会被操作系统强制挂起, 然后再挑选另外一个进程运行。抢占式调度器在这个时间间隔的最后需要一个时钟中断来获得对CPU的控制权。如果系统中没有时钟, 那么就不能实现抢占式调度算法。

在以下几种情况下将会发生进程的调度:

1. 当一个进程退出时
2. 当一个进程在I/O或信号量上阻塞时
3. 当一个新进程创建时
4. 当一个I/O发生时
5. 当一个时钟中断发生时

内核在这些时候负责进程在运行态、阻塞态和就绪态之间的转换。

3.4.1 常见的进程调度算法

一般操作系统的进程调度算法和嵌入式操作系统的进程调度算法在设计目标上有本质的不同。一般的进程调度算法的目标是尽量公平, 响应时间短, 周转时间小, 响应时间短, CPU利用率高等。而嵌入式系统则一般要求实时性, 将对实时任务的最先处理为第一目标。下面是一些调度算法的目标:

所有系统:

公平——给每个进程公平的CPU份额

策略强制执行——执行所规定的策略

平衡——保持系统所有的部分都忙碌

批处理系统:

吞吐量——最大化每小时作业数

周转时间——最小化从提交到完成的时间间隔

CPU利用率——保持CPU时钟忙碌

交互式系统:

响应时间——快速响应请求

均衡性——满足所有用户的请求

实时系统:

满足截止时间——避免丢失数据

可预测性——在多媒体系统中避免丢失

图3.8: 不同系统中调度算法的一些目标

下面来看一些常用的进程调度算法:

- 时间片轮转调度

每一个进程被分配一定长度的时间片来运行, 如果在时间片结束后进程还在运行, CPU将被分配给别的进程, 此进程挂起。在这种算法的设计中, 时间片的大小是一个决定性的因素, 一般比较合理的安排是20-50ms。

- 优先级调度

每一个进程被赋予一个优先级, 调度器总是优先调度优先级最高的就绪进程。优先级也不是一直不变的, 可以根据情况动态地改变。

- 最短进程优先

总是优先运行最短的那个进程。但是在大多数情况下进程的长短是无法预测的, 有一种方法是根据进程过去的行为进行推测, 并执行估计运行时间最短的那一个。

- 保证调度算法

这种调度算法向所有的用户进行保证, 能获得多少CPU时间, 然后去实现它。例如在单用户操作系统中的 n 个进程, 每个进程将获得 $1/n$ 的CPU时间。

- 彩票调度算法

基本思想是为每一个进程发放系统各种资源的彩票, 然后调度器在需要决策时就抽取一张彩票, 拥有彩票的进程将被调度运行。需要更多机会运行的进程就应该分配更多的彩票, 以增加运行时间。

- 公平分享调度

这种调度算法是从进程拥有者的角度来考虑问题的, 每一个进程的所有者将分得一部分CPU时间, 然后这个用户的所属进程再分享这部分时间。

3.4.2 实时任务调度

在实时系统中，时间是一个很关键的因素。实时系统通常分为硬实时（hard real time）和软实时（soft real time）系统。实时调度算法可以是静态的也可以是动态的。静态调度算法在系统启动之前完成所有的调度决策，而动态调度算法是在运行时做决定的。实时系统要响应的事件可以分为周期性和非周期性两类。常见的实时调度算法有：

- 发生率单调算法

该算法实现为每个任务分配一个与任务发生频率成正比的优先级，调度时总是调度优先级最高的任务。必要时可以剥夺当前运行的任务。

- 最早截止时间优先算法

该算法将待调度的周期性任务按截止时间排列，截止时间靠前的任务先运行。

- 最小裕度优先算法

如果一个周期性任务的运行时间是100ms，而它必须在200ms内完成，则该任务的裕度为100ms。该算法中裕度小的任务优先运行。

- 弹簧调度算法

该算法用于任务负载动态变化的软实时环境。比如在图像获取，声音采样，数据压缩等系统中，任务多数是周期性的，但是他们的执行速率并不像工业控制那样严格。此算法的基本思想是把每个任务看作一根给定弹性系数和长度限制的弹簧。铜鼓和一个由若干段不同弹簧组成的弹簧系统作类比，找出了在一个任务负载加重，超过了系统的可调度条件后，调整系统内其他任务的负载，使得系统可以继续运行下去的一种调度算法。

3.5 MINIX 3进程调度概述

MINIX 3使用一种多级调度算法。进程被赋予一个与图3.9所示结构相关的初始优先级。

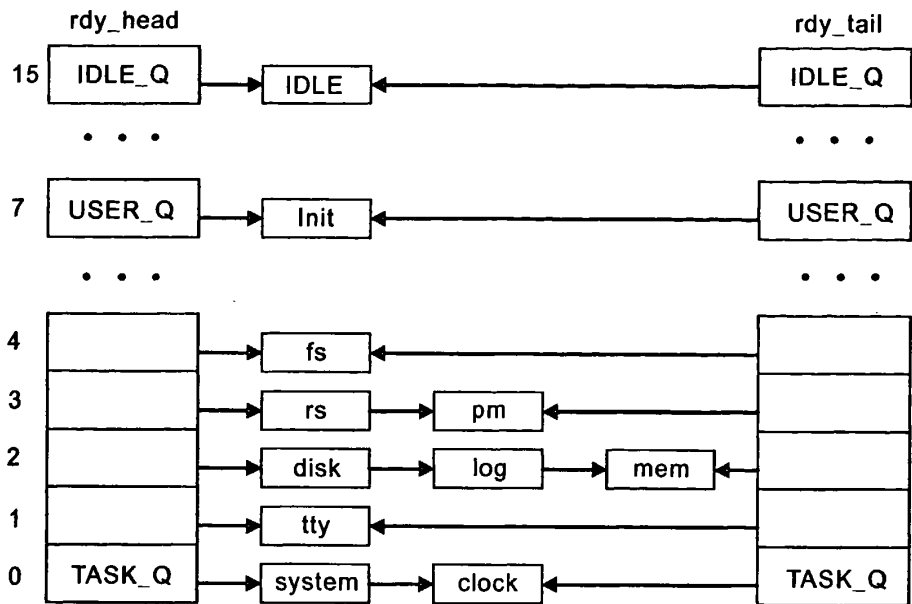


图3.9: 调度器维护的16个队列，每个队列只有一个优先级

在图3.9中，第一层的时钟和系统任务获得最高的优先级。第二层的设备驱动程序获得低一些的优先级，它们的优先级不完全相同。第三层的服务器进程的优先级比驱动程序低一些，但其中一些优先级比另一些要低。用户进程启动时优先级比所有系统进程优先级都低，并且初始值相等，通过nice命令可以提高或降低某个进程的优先级。

调度器维护16个可运行进程队列，但是任一时刻并不一定所有的队列都在使用。图3.9展示了当内核完成初始化并开始运行时，优先级队列和进程的情况。rdy_head数组中的每一项对应一个队列，这一项指向对应队列的头进程。同样，rdy_tail数组中的每一项对应队列尾的进程。

每个队列内部都采用时间片轮转调度算法。如果每一个运行的进程用完了它的时间片，则它被转移到队列尾部并分配一个新的时间片。然而当一个阻塞的进程被唤醒时，如果在阻塞前有没有用完的时间片，则它将被放到队首。它并不会得到一个新的时间片，而只是得到阻塞前所剩余的时间片。数组rdy_tail的存在使得向队列尾部加入一个进程变得简单。当一个运行的进程被阻塞或者被一个信号杀死时，进程将被移出调度队列。队列中仅有可运行进程。

有了上面描述的队列结构以后，调度算法就非常简单：找到最高非空优先级队列，选取队列首部的进程。进程IDLE总是处于就绪态，并且位于最低优先级队列中。如果所有的高优先级队列都空，则IDLE进程将运行。

3.6 MINIX 3进程调度部分代码分析及移植

我们知道，尽管不是所有的嵌入式系统都需要具有实时能力，但实时性确实是嵌入式系统最重要的特点之一。所以本文在对MINIX 3的进程调度部分的修改中，加入对实时任务的调度。

在这里我们根据任务的实时程度提供了三种调度策略：

1. SCHED_OTHER

SCHED_OTHER是面向普通非实时进程的时间片轮转策略。这种策略也就是MINIX 3中原有的调度策略，针对MINIX 3内核任务、系统进程、以及用户进程。采用该策略时，系统为处于运行状态的每个进程分配一个时间片。当时间片用完时，进程调度程序再选择下一个优先级相对较高的进程，并授予CPU使用权。

2. SCHED_FIFO

SCHED_FIFO策略适用于对响应时间要求比较高，运行所需时间比较短的实时任务。采用该策略时，各实时任务按其进入可运行队列的顺序依次获得CPU。除了因等待某个事件主动放弃CPU，或者出现优先级更高的进程而剥夺其CPU之外，该进程将一直占用CPU运行。

3. SCHED_RR

SCHED_RR策略适用于对响应时间要求比较高，运行所需时间比较长的实时任务。采用该策略时，各实时任务按时间片轮流使用CPU。当一个运行进程的时间片用完后，进程调度程序停止其运行并将其置于可运行队列的末尾。

为了实现对实时任务的调度，需要在MINIX 3原有16个进程优先级的基础上再加入16个实时任务的对应的优先级，共计32个优先级。如图3.10所示：

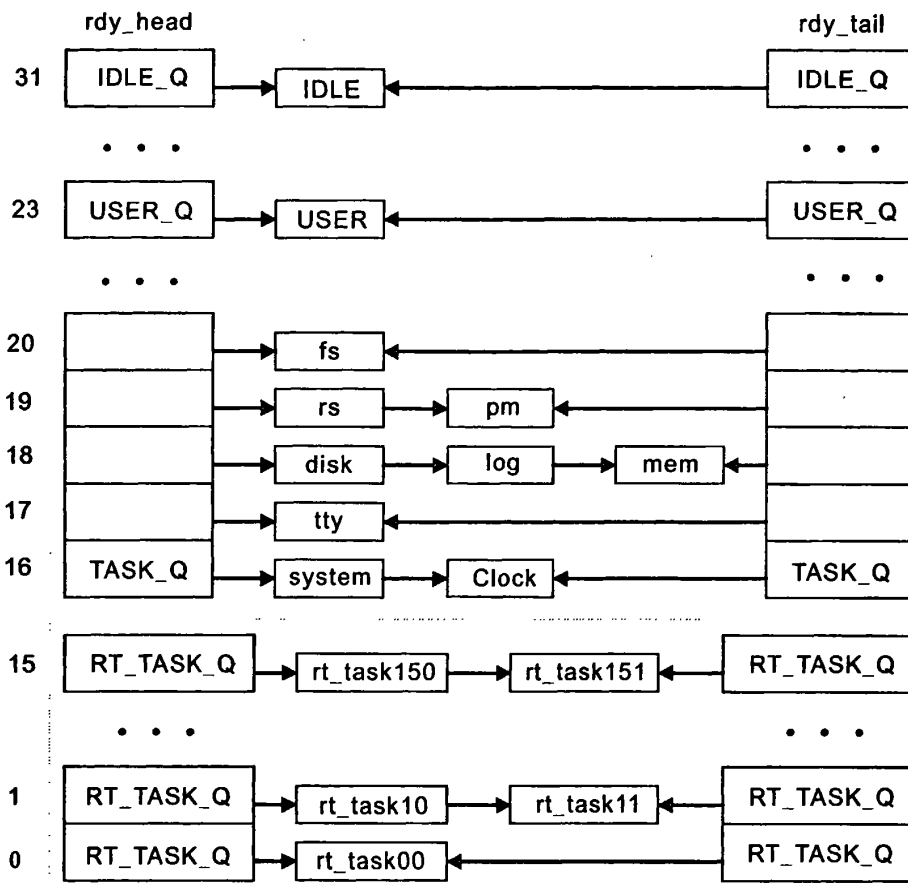


图3.10: arm-minix中的进程优先级队列，共计32个队列，其中实时任务使用前16个队列

在上图中，为arm-linux中的可运行进程的优先级队列。其中新加入了前0-15个优先级队列用来调度实时进程，剩下的16-31个优先级队列用来调度MINIX 3系统中的所有进程和普通用户进程。其中有两类进程的CPU是不可抢占的，其中一种是用SCHED_FIFO策略调用的比较短的实时任务，另一种MINIX操作系统的内核空间进程。内核的不可抢占性也就决定了这种进程调度是软实时的，而并非硬实时。

有了以上的描述，那么可以得出新的进程调度算法：当需要进程调度的时候，先看系统中有无就绪的实时任务，如果有的话就挑选优先级最高的就绪实时任务。如果该任务是SCHED_FIFO类型的，那么就一直运行直到自行退出或自行阻塞，或者有优先级更高的实时任务剥夺其CPU。若该任务是SCHED_RR类型的，那么对该任务实行时间片轮转调度，运行完分配的时间片之后被放进相应优先级队列的末端。如果没有就绪的实时任务，那么就按照MINIX 3原来的调度算法对非实时进程进行调度。

下面来看两个主要与进程调度相关的文件：proc.h和proc.c，它们分别定义了与进程调度相关的类型和函数。其中调度器pick_proc就定义在proc.c文件中。要对MINIX 3的进程调度进行移植，必须对这两个文件中的数据结构和函数进行修改。

接下来我们来看一看修改的具体过程（相关代码见附录）。首先讨论文件proc.h，文件proc.h主要定义了进程表项的数据结构struct proc。在MINIX 3的目录< ./kernel/ >下，头文件proc.h定义了一个对进程调度非常重要的数据结构proc，也就是内核进程表的表项。proc包括了所有的与进程有关的数据，包括寄存器、标志位、调度优先级、内存映射、记账、消息发送相关信息，等等。很多汇编代码的例程的引用域都位于这个结构中。

结构体proc在起始位置包含了另一个数据结构：struct stackframe_s，用来在进程上下文切换的时候保存当前进程的寄存器值，它实际上起到了一个堆栈的作用，它的地址保存在TSS（Task State Segment，任务状态段）中。那么当然的，结构体stackframe_s属于MDC，所以需要重新定义对应ARM体系结构寄存器的这个结构体。重新定义后的stackframe_s位于arm-minix/arch/arm-linux/atypes.h文件中，如图3.11。

```
struct stackframe_s {
    u32_t cpsr;
    u32_t r4;
    u32_t r5;
    u32_t r6;
    u32_t r7;
    u32_t r8;
    u32_t r9;
    u32_t sl;
    u32_t fp;
    u32_t pc;
};
```

图3.11：重新定义后的stackframe_s结构

结构体proc中还包含了用于描述程序段的结构体数组，由于Intel 32体系结构使用的是全局描述符表和局部描述符表的保护模式，在原MINIX 3版本中用于保存当前程序段地址的结构体也属于MDC。由于ARM体系结构的保护模式的实现原理的不同，在这里我们去除与Intel 32保护模式相关的部分代码：

```
#include "protect.h"
reg_t p_ldt_sel;
struct segdesc_s p_ldt[2+NR_REMOTE_SEGS];
```

图3.12：结构体proc中的MDC

取而代之，我们重新定义用于描述程序段地址的结构体segdesc_t。最终，proc包含一个segdesc_t的数组segframe_t来存储所有的程序段地址。此数据结构的定义也放在文件arm-minix/arch/arm-linux/atypes.h中，如图3.13。

```
typedef unsigned int segdesc_t;

typedef struct segframe_s {
    segdesc_t sr[NR_SEGR];
} segframe_t;
```

图3.13: 结构体segdesc_t的定义

接下来看proc.h中其它与进程调度相关的部分。在文件的后半部分有一些宏定义，它们是一些与优先级有关的定义，如内核任务的优先级，系统进程的优先级，用户进程的默认优先级及上下边界值等等。

```
#define TASK_Q          0
#define MAX_USER_Q      0
#define USER_Q          7
#define MIN_USER_Q      15
#define IDLE_Q          15
```

图3.14: MINIX 3内核进程及系统进程的优先级

由于实时任务的优先级要高于MINIX系统中所有的其它进程，所以要对图3.12中的定义做出修改，将它们的优先级逐个降低16，将前16个优先级留给实时任务。做出的修改和添加的新定义如下：

```
#define NR_SCHED_QUEUES    32 #define MAX_RT_TASK_Q      0
#define MIN_RT_TASK_Q      15 #define TASK_Q            16
#define MAX_USER_Q         16 #define USER_Q            23
#define MIN_USER_Q         30 #define IDLE_Q             31
```

图3.15: 对优先级宏定义的修改

最后还有一点要说明，因为对实时任务和MINIX进程采用了不同的调度策略，所以要告诉内核这个进程采用了什么调度策略，就要在进程表项中添加一个变量“unsigned long policy”，用来说明此进程的调度策略。还要在proc.h文件中添加对应这个变量的宏定义，分别对应三种不同的调度策略：SCHED_OTHER、SCHED_OTHER、SCHED_RR。

至此，我们就完成了对文件proc.h的修改。下一步，继续来看进程调度器的定义文件proc.c。在文件proc.c中，与进程调度相关的函数主要有四个，它们

是dequeue、enqueue、sched和pick_proc。从本质上讲，pick_proc就是调度器。下面分别来讨论这四个函数。

首先来看函数enqueue。函数void enqueue(register struct proc *rp)以一个进程指针作为参数，首先这个进程是可运行的，它的目的就是把这个进程加入到某一个可运行进程队列中。注意在这个函数中有两个变量，int q和int front，正是这两个参数实现了进程调度策略（mechanism）和机制（policy）的分离。在函数enqueue中，以变量q和front作为参数调用函数sched，q和front的值由函数sched赋予并返回给函数enqueue。变量q的取值决定这个进程应该属于哪个队列，变量front决定此进程是放在队首还是队尾，这就是策略部分。然后函数enqueue根据他们的值，作出相应的动作，将进程放在正确的位置，这是机制。我们对函数enqueue作出一些修改，让它在开始对进程的类型做出一个判断，如果是MINIX进程，就调用sched函数；如果是实时任务，就调用rt_sched函数：

```
if ( rp->policy == SCHED_OTHER )
    sched(rp, &q, &front);
else rt_sched( rp, &q, &front );
```

图3.16: 对实时任务和普通MINIX 3进程分别调度

函数enqueue接下来就会根据变量q和front的值将此进程插入相应队列。如果将要插入的q号调度队列是一个空队列，那么就按照变量q的值新建一个q号调度队列，并将队列头指针rdy_head[q]和队列尾指针rdy_tail[q]都指向当前进程。如果q号调度队列不为空，那么就根据变量front的值决定将进程插入队列头部还是尾部。如果front值为1，那么就插入q号队列头部，将指针rdy_head[q]指向当前进程；如过front值为0，那么就插入q号队列尾部，将指针rdy_tail[q]指向当前进程。函数的最后将调用pick_proc()来决定下一个将要运行的进程。

函数denqueue的作用与enqueue相反，将一个进程转换为非就绪态。一个将要阻塞的进程必然处于运行态，那么它就位于相应调度队列的首端。但是还有一种情况，可能会给一个不处于运行态的进程发了信号，那么就要遍历队列来找到这个进程，并将其移出队列。

在函数denqueue的开始部分，首先要判断当前进程是否是内核空间进程，如果是的话，就要对内核任务栈进行一个检查。如果正常，就继续往下执行；如果出错，就发出警告信息，终止系统的运行。

然后，就要在目标进程所在队列中寻找将要移出队列的那个目标进程了，它最可能在队列头部，但也有可能在队列中的任何地方。目标进程表项中的p_priority标志记录了此进程在第几个调度队列，由此可以决定要搜寻目标进程的队列。最后，用一个for循环遍历此队列，找到目标进程，并将它移出队列。

函数sched决定了进程调度的策略，当一个进程需要被插入调度队列的时候调用这个函数，决定插入位置。它可能会更新进程的优先级。相应地，我们新增一个函数rt_sched，给出实时任务的调度策略，决定将实时任务插入队列时的位置。

在函数`sched`的开始定义了三个变量：`static struct proc *prev_ptr`、`int time_left`和`int penalty`。`prev_ptr`是一个指向上一个用完时间片的进程的指针，一定要被声明为静态变量。`time_left`是一个整形变量，表示进程是否用完所分配的时间片，它的值由进程的标志`p_ticks_left`是否大于零来决定。如果大于零，表示有时间片剩余，`time_left`赋值为1；如果等于零，表示已经用完了时间片，`time_left`赋值为0。`penalty`也为整形变量，表示进程优先级的升降。

下一步用一个`if`语句判断进程是否有残存的时间片，如果没有，就为进程重新分配一个时间量程，这个量程的大小由进程的标志`p_quantum_size`决定。注意，在这里还要用另一个`if`语句进行一个判断，如果上一个用完时间片的进程是当前进程的话，为了避免可能的无限循环，就将当前进程的优先级降低1，即赋予`penalty`一个正值：1；如果上一个用完时间片的进程不是当前进程的话，就将当前进程的优先级升高1，即赋予`penalty`一个负值：-1。然后将指针`prev_ptr`指向当前进程。

接下来的`if`语句更新进程的优先级，也就是改变`p_priority`的值，使更新后的`p_priority`的值等于原来的`p_priority`的值加上`penalty`的值。在这里内核空间进程的优先级是不会改变的。并且还要对进程的优先级是否越界做一个检查，将进程的优先级放在上下边界之内，例如用户空间中的普通进程最高优先级为16，最低为30。最后将这个进程放在相应调度队列的尾部。

如果进程有时间片残留，那么直接将进程放在相应调度队列的队首，使其可以立即运行。

新增的调度函数`rt_sched`基本上与`sched`相同，区别就是它对实时任务进行调度：实时任务没有内核空间 and 用户空间之分，并且优先级越界的判断也是在实时任务的范围之内进行的。

最后一个函数`pick_proc`决定哪一个进程是下一个将要运行的进程，这个进程由指针`next_ptr`指向。并且如果这个进程是可计费的，那么指针`bill_ptr`指向这个进程，计算这个进程的CPU时间。这个函数的主体只有一个`for`循环，它从0级调度队列头部开始依次遍历32个调度队列，运行遇到的第一个进程。

到这里，我们就完成了对MINIX 3进程调度部分向ARM软实时进程调度的修改。

第4章 结论

本文重点讨论了MINIX 3操作系统移植的一般方法和过程，建立可移植代码的方法，从总体上讲述了如何移植一个操作系统，移植一个操作系统的步骤等。分析了MINIX 3操作系统的内核结构，代码组织，本人阅读了MINIX操作系统内核部分代码和之外的相当一部分代码，收获颇丰。

最后修改了MINIX 3内核中进程调度部分的源代码，使其符合软实时操作系统进程调度的特征，并能进行进程的软实时调度。但是软实时的进程调度并不能满足对一些实时响应要求很高的实时任务的调度。只能实现软实时进程调度，这是由MINIX 3内核的特性决定的，例如：

1. MINIX 3的内核是非抢占式的
2. MINIX 3的进程调度策略不是完全抢占式的
3. MINIX 3采用的时钟中断的精度不高
4. MINIX 3系统的一些额外操作会延迟实时进程的执行

要在MINIX 3中实现硬实时的进程调度，一种办法是在MINIX 3原有内核和中断之间加入一个新的实时内核，用它来截获MINIX 3内核的关中断请求，从而实现MINIX 3内核任务的可抢占性。第二种方法是对普通的MINIX 3的内核的数据结构、调度函数、中断方式等进行修改使其能够处理实时进程。不管采用上述哪一种方式，都需要对MINIX 3的内核做出很大的改动，第一种方式甚至要创建一个新的内核，难度和工作量都是相当可观的，不过这也可能是MINIX 3今后的一种研究方向。

在MINIX 3内核向ARM体系结构移植方面做的工作还很少，只对内核做了比较小的一部分工作，还仅仅是开了一个头，很多东西只是停留在理论的层面上。不懂的问题也很多，只有待日后研究解决。

参考文献

- [1] Andrew S.Tanenbaum and Albert S.Woodhull, 操作系统设计与实现(第二版), 电子工业出版社, 1998.08
- [2] Andrew S.Tanenbaum and Albert S.Woodhull, 操作系统设计与实现(第三版), 电子工业出版社, 2007.03
- [3] Andrew S.Tanenbaum, Modern Operating Systems(Second Edition), Prentice Hall, 2002.01
- [4] Thomas E.Anderson, Brian N.Bershad, Edward D.Lazowska, Henry M.Levy, Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, ACM Trans. on Computer Systems vol. 10, pp. 53-79, 1992.02
- [5] M. J.BACH, The Design of the UNIX Operating System, Upper Saddle River, NJ: Prentice Hall, 1987
- [6] L. F.BIC, A. C.SHAW, Operating System Principles, Upper Saddle River, NJ: Prentice Hall, 2003
- [7] P.BRINCH HANSEN, Classic Operating Systems, New York: Springer-Verlag, 2001
- [8] V. G.CERF, Spam, Spim, and Spit, Commun. of the ACM vol. 48, pp.39-43, 2005.04
- [9] A.CHOU, J.-F.YANG, B.CHELF, S.HALLEM, An Empirical Study of Operating System Errors, Proc. 18th Symp. on Oper. Syst. Prin., ACM, pp.73-88, 2001
- [10] J.CORBET, A.RUBINI, G.KROAH-HARTMAN, Linux Device Drivers(Third Edition), Sebastopol, CA: O'Reilly, 2005
- [11] H. M.DEITEL, P. J.DEITEL, D. R.CHOFFNES, Operating Systems(Third Edition), Upper Saddle River, NJ: Prentice-Hall, 2004
- [12] E. W.DIJKSTRA, My Recollections of Operating System Design, Operating Systems Review vol. 39, pp. 4-40, 2005.04
- [13] C.DODGE, C.IRVINE, T.and NGUYEN, A Study of Initialization in Linux and OpenBSD, Operating Systems Review vol. 39, pp. 79-93, 2005.04
- [14] D. R.ENGLER, M. F.KAASHOEK, J. Jr.O'TOOLE, Exokernel: An Operating System Architecture for Application-Level Resource Management, Proc. 15th Symp. on Oper. Syst. Prin., ACM, pp. 251-266, 1995.
- [15] M. K.MCKUSICK, G. V.NEVILLE-NEIL, The Design and Implementation of the FreeBSD Operating System, Addison-Wesley: Boston, 2005
- [16] 康一梅, 嵌入式软件设计, 机械工业出版社, 2007.06
- [17] 周立功, ARM嵌入式系统基础教程, 北京航空航天大学出版社, 2005.01

- [18] 赵星寒, 刘涛, 从51到ARM-32位嵌入式系统入门, 北京航空航天大学出版社, 2005.10
- [19] Andrew Sloss, Dominic Symes, Chris Wright, ARM System Developer's Guide: Designing and Optimizing System Software, Elsevier, 2005.05
- [20] 赵炯, Linux内核完全剖析, 机械工业出版社, 2006.01
- [21] 沈美明, 温冬婵, x86汇编语言程序设计, 清华大学出版社, 2001.09
- [22] W.Richard Stevens, Advanced Programming in the UNIX Environment, Addison Wesley/Pearson, 2000.02
- [23] Eric S.Raymond, The Art of Unix Programming, 电子工业出版社, 2006.03
- [24] DANIEL P. BOVET, MARCO CESATI, UNDERSTANDING THE LINUX KERNEL, O'Reilly Media, Inc., 2007.09
- [25] Jean J. Labrosse, MicroC/OS-II The Real-Time Kernel(Second Edition), CMP, 2003.05
- [26] Karim Yaghmour, Building Embedded Linux Systems, O'Reilly, 2004.12
- [27] 吴景峰, 嵌入式操作系统进程调度研究, 电脑知识与技术, 2006.10
- [28] 毛耀, 杨颂华, MINIX中的Intel x86汇编语言, 西南民族大学学报, pp. 615-620, 2003.10
- [29] 朱晖, 嵌入式实时操作系统RT-MINIX的设计, 西南交通大学研究生学位论文, 2003.3
- [30] 秦蔚, ARM平台下Linux内核移植技术的分析研究与应用, 昆明理工大学硕士学位论文, 2004.06
- [31] 项功宏, Minix的研究及向uCLinux的移植, 浙江大学硕士学位论文, 2005.03
- [32] 范艳开, 基于ARM的嵌入式Linux操作系统的移植, 西北工业大学硕士学位论文, 2005.03
- [33] 刘勇, 嵌入式Linux开发技术研究, 西南交通大学硕士学位论文, 2004.05
- [34] 褚亚铭, 一个教学用微内核操作系统的设计与实现, 苏州大学硕士学位论文, 2005.04
- [35] Ingmar A. Alting, MinixPPC A port of the MINIX OS to the PowerPC platform, Masters thesis Computer Science vrije Universiteit amsterdam, 2006.09
- [36] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, Andrew S. Tanenbaum, Construction of a highly Dependable Operating System, Proc. 6th European Dependable Comp. Conf., 2006.10
- [37] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, Andrew S. Tanenbaum, Reorganizing UNIX for Reliability, Proc. 11th ACSAC, 2006.09

- [38] Jorrit N.Herder, Herbert Bos, Ben Gras, Philip Homburg, Andrew S.Tanenbaum, Minix 3:A Highly Reliable,Self-Repairing Operating System, Oper. Sys. Rev., 2006.07
- [39] Jorrit N.Herder, Herbert Bos, Andrew S.Tanenbaum, A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers, Technical Report IR-CS-018, 2006.01
- [40] Jorrit N.Herder, Herbert Bos, Andrew S.Tanenbaum, Can We Make Operating Systems Reliable and Secure?, IEEE Computer, 2006.05
- [41] Ivan Kelly, Porting MINIX to Xen, Submitted in part requirment for final year project course CS4618 to University of Limerick,Ireland, 2006,05
- [42] Jorrit N.Herder, Herbert Bos, Ben Gras, Philip Homburg, Andrew S.Tanenbaum, Modular System Programming in MINIX 3, USENIX ;login, 2006.04
- [43] Jorrit N.Herder, Towards a True Microkernel Operating System, Masters thesis Computer Science vrije Universiteit amsterdam, 2005.02
- [44] Rogier Meurs, Building Performance Measurement Tools for the MINIX 3 Operating System, Masters thesis Computer Science vrije Universiteit amsterdam, 2006.08

致 谢

本文是在周宇斌导师的严格要求和悉心指导之下完成的。周宇斌老师学识渊博，知识涉猎多个学科领域，他对知识的大局观以及治学态度都让我们获益匪浅。正是他对操作系统的热情和对嵌入式系统的远见卓识让我对操作系统和嵌入式产生了浓厚的兴趣，他也是我操作系统的启蒙导师，使我走上了操作系统的研究道路，这也是我论文选题的基础。这三年来，无论是在学习和生活上，还是在为人处事，做人的道理上，他都是我效仿的榜样。在此向他表示衷心的感谢和由衷的敬意。

同时还要感谢同学何昕、张志勇、荣杰林、马双双，他们在学业和生活上对我的帮助也很大，我忘不了和他们一起讨论问题，学习知识的情景。三年来，我们同甘共苦，互相帮助，共同进步，一起走到了现在。

无私的父母一直都是我最大的支持，没有他们就没有今天的我。

最后，衷心感谢数学与统计学院各位老师和领导三年来对我的帮助和关怀，感谢所有帮助过我的老师和同学，谢谢你们。

附录

```

+++++
arm-minix/arch/arm-linux/atypes.h
+++++
typedef unsigned int segdesc_t;

struct stackframe_s {
    u32_t cpsr;
    u32_t r4;
    u32_t r5;
    u32_t r6;
    u32_t r7;
    u32_t r8;
    u32_t r9;
    u32_t sl;
    u32_t fp;
    u32_t pc;
};

typedef struct segframe_s {
    segdesc_t sr[NR_SEGR];
} segframe_t;

+++++
arm-minix/kernel/proc.h
+++++
#ifndef PROC_H
#define PROC_H

#include <minix/com.h>
#include "const.h"
#include "priv.h"

struct proc {
    struct stackframe_s p_reg;        /*进程的寄存器保存在栈框中*/
    segframe_t p_seg;                /*段描述符*/

    proc_nr_t p_nr;                  /*第几个进程*/
    struct priv *p_priv;              /*系统特权结构体*/
    char p_rts_flags;                 /*消息状态，接收、发送等*/

    char p_priority;                  /*当前的调度优先级*/

```

```

unsigned long policy;           /* 此进程的调度策略 */
char p_max_priority;           /* 最大调度优先级 */
char p_ticks_left;             /* 剩下的时钟嘀嗒数 */
char p_quantum_size;           /* 时钟量程的大小 */

struct mem_map p_memmap[NR_LOCAL_SEGS];
                                /* 内存映射 */

clock_t p_user_time;           /* 用户记帐时间 */
clock_t p_sys_time;           /* 系统记帐时间 */

struct proc *p_nextready;      /* 下一个可运行进程指针 */
struct proc *p_caller_q;
                                /* 希望对其发送消息的进程队列的头指针 */
struct proc *p_q_link;
                                /* 指向下一个希望被发送进程的指针 */
message *p_messbuf;           /* 指向发送了的消息的指针 */
proc_nr_t p_getfrom;
                                /* 希望接从其接受消息的进程的进程号 */
proc_nr_t p_sendto;
                                /* 希望发送消息的进程的进程号 */

sigset_t p_pending;           /* 挂起的内核信号的位图 */

char p_name[P_NAME_LEN];
                                /* 此进程的名字，末尾有字符\0 */
};

/* 运行时标志位。一个进程时可运行的当且仅当 p_rts_flags == 0. */
#define SLOT_FREE 0x01          /* 次进程表项为空 */
#define NO_MAP 0x02             /* 防止还没有内存映射的子进程运行 */
#define SENDING 0x04            /* 试图被阻塞SEND */
#define RECEIVING 0x08          /* 试图被阻塞RECEIVE */
#define SIGNED 0x10             /* 当新的内核信号到达时置位 */
#define SIG_PENDING 0x20
                                /* 当信号没有到达时挂起 */
#define P_STOP 0x40             /* 当进程被跟踪时置位 */
#define NO_PRIV 0x80            /* 禁止系统进程运行 */

/* 进程的调度策略 */
#define SCHED_OTHER 0           /* 用于调度非实时进程 */
#define SCHED_FIFO 1           /* 用于调度短实时任务 */
#define SCHED_RR 2             /* 用于调度长实时任务 */

```

```

/* 进程调度优先级的相关定义 */
#define NR_SCHED_QUEUES      32    /* 最小优先级 + 1 */
#define MAX_RT_TASK_Q        0     /* 实时任务的最高优先级 */
#define MIN_RT_TASK_Q        15    /* 实时任务的最小优先级 */
#define TASK_Q                16
                                /* 非实时任务的最高优先级, 内核任务专用 */
#define MAX_USER_Q           16    /* 用户进程的最低优先级 */
#define USER_Q               23    /* 用户进程的默认优先级 */
#define MIN_USER_Q           30    /* 用户进程的最小优先级 */
#define IDLE_Q               31    /* 最低优先级, 只属于进程IDLE */

/* 进程表的地址 */
#define BEG_PROC_ADDR (&proc[0])
#define BEG_USER_ADDR (&proc[NR_TASKS])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])

#define NIL_PROC              ((struct proc *) 0)
#define NIL_SYS_PROC          ((struct proc *) 1)
#define cproc_addr(n)         (&(proc + NR_TASKS)[(n)])
#define proc_addr(n)          (pproc_addr + NR_TASKS)[(n)]
#define proc_nr(p)            ((p)->p_nr)

#define isokprocn(n)           ((unsigned) ((n) + NR_TASKS) < NR_PROCS +
                                NR_TASKS)
#define isemptyn(n)           isemptyp(proc_addr(n))
#define isemptyp(p)           ((p)->p_rts_flags == SLOT_FREE)
#define iskernelp(p)          iskerneln((p)->p_nr)
#define iskerneln(n)          ((n) < 0)
#define isuserp(p)            isusern((p)->p_nr)
#define isusern(n)            ((n) >= 0)

/* 以下定义进程表和指向进程表项的指针。这些指针使得访问进程表更快速, 因为现在一个进程
   表项由数组中的索引找到。当要访问进程表的元素时, pproc_addr 它的地址可以用乘以
   sizeof(struct proc) 得到。 */

EXTERN struct proc proc[NR_TASKS + NR_PROCS];    /* 进程表 */
EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
EXTERN struct proc *rdy_head[NR_SCHED_QUEUES];
                                /* 指向就绪进程队列头部 */
EXTERN struct proc *rdy_tail[NR_SCHED_QUEUES];
                                /* 指向就绪进程队列尾部 */

```

```
#endif /* PROC.H */

+++++
arm-minix/kernel/proc.c
+++++
/*=====
enqueue
=====*/
PRIVATE void enqueue(rp)
register struct proc *rp;      /*此进程现在是可运行的 */
{
    /* 将一个进程加入到一个可运行进程的队列中。这个函数负责将一个进程插入到一个调度队列
       中。机制在这里实现。实际的调度策略由函数sched()和pick_proc()定义。*/
    int q;                      /* 将要用到的调度队列 */
    int front;                  /* 加到队首还是队尾 */

    /* 决定将进程插入到哪里 */
    if ( rp->policy == SCHED.OTHER )
        sched(rp, &q, &front);
    else rt_sched( rp, &q, &front );

    /* 现在, 将进程插入到队列 */
    if (rdy_head[q] == NIL_PROC) {          /* 加入到空队列中 */
        rdy_head[q] = rdy_tail[q] = rp;    /* 建立一个新队列 */
        rp->p_nextready = NIL_PROC;        /* 标记一个新的队尾 */
    }
    else if (front) {                      /* 插入到队列头 */
        rp->p_nextready = rdy_head[q];      /* 队列的头部 */
        rdy_head[q] = rp;                  /* 设置一个新的队列头*/
    }
    else {                                 /* 插入到队尾 */
        rdy_tail[q]->p_nextready = rp;      /* 队列尾部 */
        rdy_tail[q] = rp;                  /* 设置一个新的队尾 */
        rp->p_nextready = NIL_PROC;        /* 标记一个新的队尾 */
    }

    /* 现在选择一个新的进程去运行 */
    pick_proc();
}

/*=====
denqueue
=====*/
```

```

PRIVATE void dequeue(rp)
register struct proc *rp; /*这个进程不再是可运行的了 */
{
    /*一个进程必须被从调度队列中移出，例如，它阻塞了。如果当前的活动进程被移出，由函数pick_proc()一个新进程运行。*/
    register int q = rp->p_priority; /* 将要用到的队列 */
    register struct proc **xpp; /* 迭代所有的队列 */
    register struct proc *prev_xp;

    /* 检查内核任务栈是否正常 */
    if (iskernelp(rp)) {
        if (*priv(rp)->s_stack_guard != STACK_GUARD)
            panic("stack_overrun_by_task", proc_nr(rp));
    }

    /* 现在要确定进程不在它的就绪队列中了。如果找到进程，移出队列。一个进程可以被转为非就绪态，即使它没有在运行，因为可以发送一个信号杀死它。*/
    prev_xp = NIL_PROC;
    for (xpp = &rdy_head[q]; *xpp != NIL_PROC; xpp = &(*xpp)->p_nextready) {
        if (*xpp == rp) { /* 寻找要移出的进程 */
            *xpp = (*xpp)->p_nextready; /* 指向队列中下一个进程 */
            if (rp == rdy_tail[q]) /* 从队列尾部移出 */
                rdy_tail[q] = prev_xp; /* 设置一个新的队尾 */
            if (rp == proc_ptr || rp == next_ptr) /* 如果移出的是活动进程 */
                pick_proc(); /* 选择一个新的进程来运行 */
            break;
        }
        prev_xp = *xpp; /* 保存前一个进程 */
    }
}

/*=====
                                sched
=====*/
PRIVATE void sched(rp, queue, front)
register struct proc *rp; /* 将要被调度的进程 */
int *queue; /* 返回将要被插入的队列 */
int *front; /* 返回队首或队尾 */
{
    /*这个函数决定了调度策略。它在一个进程必须被加入到一个调度队列的时候被调用，决定将这个进程插入到哪里。进程的优先级可能被更新。*/

```

```

static struct proc *prev_ptr = NIL_PROC;
/* 前一个用完时间片的进程 */
int time_left = (rp->p_ticks_left > 0);
/* 进程消费的整个量程 */
int penalty = 0;
/* 优先级的改变 */

/* 检查进程是否还有残留时间。如果没有的话分配一个新的时间量程并且提升优先级。一个
   队列中的进程使用多时间量程，如果一个高优先级的进程（系统服务器和驱动程序）进入
   了可能的无限循环，它们将被降低优先级。CPU*/
if ( ! time_left ) {
    /* 消费的时间量程 */
    rp->p_ticks_left = rp->p_quantum_size;
    /* 获得一个新时间量程 */
    if (prev_ptr == rp) penalty ++;
    /* 捕获无限循环 */
    else penalty --;
    /* 增加进程的优先级 */
    prev_ptr = rp;
    /* 为下一次运行保存进程指针 */
}

/* 决定这个进程新的优先级。优先级边界由队列和这个进程的最大优先级决定。内核任务和
   进程永远不改变优先级。IDLEidle*/
if (penalty != 0 && ! iskernelp(rp)) {
    rp->p_priority += penalty;
    /* 更新罚金 */
    if (rp->p_priority < rp->p_max_priority) /* 检查上边界 */
        rp->p_priority = rp->p_max_priority;
    else if (rp->p_priority > IDLE.Q-1) /* 检查下边界 */
        rp->p_priority = IDLE.Q-1;
}

/* 如果有时间残留，进程被插入到相应队列的最前面，所以它可以立即运行。插入队列总是
   由进程当前的优先级决定。*/
*queue = rp->p_priority;
*front = time_left;
}

/*=====
rt_sched
=====*/
/*实时任务的调度策略*/
PRIVATE void rt_sched ( rp, queue, front )
register struct proc *rp;
int *queue;
int *front;
{
    static struct proc *prev_ptr = NIL_PROC;

```

```

int time_left = ( rp->p_ticks_left >0 );
int penalty = 0;

if ( ! time_left ) {
    rp->p_ticks_left = rp->p_quantum_size;
    if ( prev_ptr == rp ) penalty ++;
    else penalty --;
    prev_ptr = rp;
}

if ( penalty != 0 ) {
    rp->p_priority += penalty;
    if ( rp->p_priority < rp->p_max_priority )
        rp->p_priority = rp->p_max_priority;
    else if ( rp->p_priority > rp->MIN_RT_TASK_Q )
        rp->p_priority = MIN_RT_TASK_Q;
}

*queue = rp->p_priority;
*front = time_left;
}

/*=====
                                pick_proc
=====*/
PRIVATE void pick_proc()
{
    /* 现在决定运行哪个进程。下一个要运行的进程由指针"next_ptr"指出。当一个可计帐的
       进程被选择, 指针"bill_ptr"指向它, 这样时钟任务就可以知道谁为系统时间付
       账。"*/
    register struct proc *rp;          /* 将要运行的进程 */
    int q;                             /* 迭代所有队列 */

    /* 检查每一个就绪进程的调度队列。队列的数量定义在文件proc.中, 并且优先级在表中定
       义。最低级队列包含进程, 它总是就绪的himagIdle*/
    for (q=0; q < NR_SCHED_QUEUES; q++) {
        if ( (rp = rdy_head[q]) != NIL_PROC ) {
            next_ptr = rp;              /* 下一次运行进程"rp" */
            if ( priv(rp)->s.flags & BILLABLE )
                bill_ptr = rp;          /* 为系统时间付账 */
            return;
        }
    }
}

```

}