重庆大学本科学生毕业设计（论文）

基于ARM的EPOS操作系统的移植



学 生：冯志敏

学 号：20121955

指导教师：洪明坚

专 业：软件工程专业

重庆大学软件学院

二O一六年六月

**Graduation Design (Thesis) of Chongqing University**

**Transplantation of EPOS operating system based on ARM**



**Undergraduate: Feng Zhimin**

**Instructor: Associated Prof.Hong Mingjian**

**Major: Software Engineering**

**School of Software Engineering**

**Chongqing University**

**June 2016**

# **摘 要**

随着开源硬件在嵌入式领域的广泛使用，越来越多基于ARM体系结构的开发板产品也随之诞生，其中比较流行的一款开源硬件—Raspberry PI(树莓派)，其体积仅信用卡大小，搭载ARM架构处理器，运算性能和智能手机相仿。本文将EPOS操作移植到Raspberry PI开发板上。

本文是将X86体系结构下的EPOS操作系统移植到以Broadcom BCM2835 ARMv6 为处理器的开源硬件Raspberry PI B+上，主要是将EPOS由X86系统结构修改成ARM体系结构，大致修改内容包括引导项(Bootloader)、内存管理单元(MMU)、中断处理、系统定时器、多线程的切换以及PI开发板上的UART驱动。

本文实现的功能为：将EPOS操作系统编译后生成kernel.img文件，将kernel.img文件拷贝到FAT文件格式的SD卡中，将SD卡插入Raspberry PI B+卡槽内，开启电源后，通过UART串口向PC机发送一条消息，同时初始化多任务。

关键词：嵌入式， Bootloader，UART，MMU，Raspberry PI B+

# ABSTRACT

With the widely use of the open source hardware in the embedded field, more and more based on ARM architecture development board products emerge, which more popular a open source hardware - Raspberry PI (raspberry pi), the volume of only credit card size, equipped with ARM processor architecture, the computational performance and smart phones similar。In this paper, the EPOS operation is transplanted to the PI Raspberry development board.

In this paper, the x86 architecture of EPOS operating system transplantation by Broadcom BCM2835 armv6 processors of open source hardware raspberry PI B +, mainly is the EPOS by x86 system structure modified to arm architecture, roughly modified content includes a guide (bootloader), memory management unit (MMU), interrupt processing, system timer, multi thread switching and PI development board UART driver.

In this paper, the realization of the function is: EPOS operating system compiler generated after kernel.img file, kernel.img files are copied to the SD card fat file format in, SD card into the raspberry PI B + card slot, after power is turned on, through the UART serial port to the PC sends a message. Meanwhile, the initial of multi task.

**Key words：**Embedded, Bootloader, UART, MMU, Raspberry PI B+

# 目录

[摘要 I](#_Toc358850539)

[ABSTRACT II](#_Toc358850540)

[目录 III](#_Toc358850541)

[1 绪论 7](#_Toc358850542)

[1.1 选题背景及意义 7](#_Toc358850543)

[1.2 开发板介绍 **错误!未定义书签。**](#_Toc358850545)

[1.3 开发环境搭建 **错误!未定义书签。**](#_Toc358850545)

[2 系统关键技术 **错误!未定义书签。**](#_Toc358850546)

[2.1 ARM体系结构介绍 **错误!未定义书签。**](#_Toc358850547)

[2.2 Bootloader设计 **错误!未定义书签。**](#_Toc358850548)

[2.2.1 Raspberry PI启动流程介绍 **错误!未定义书签。**](#_Toc358850549)

[2.2.2 EPOS初始化以及加载kernel **错误!未定义书签。**](#_Toc358850550)

[2.3 ARM虚拟内存 **错误!未定义书签。**](#_Toc358850551)

[2.3.1 MMU工作原理 **错误!未定义书签。**](#_Toc358850552)

[2.3.2 CP15协处理器 **错误!未定义书签。**](#_Toc358850553)

[2.3.3 EPOS恒等映射 **错误!未定义书签。**](#_Toc358850553)

[2.3.4 EPOS pagefault处理 **错误!未定义书签。**](#_Toc358850553)

[2.4 ARM中断处理 **错误!未定义书签。**](#_Toc358850551)

[2.4.1 中断机制 **错误!未定义书签。**](#_Toc358850552)

[2.4.2 中断向量表 **错误!未定义书签。**](#_Toc358850553)

[2.4.3 Raspberry PI定时器 **错误!未定义书签。**](#_Toc358850553)

[2.5 多任务处理 **错误!未定义书签。**](#_Toc358850551)

[2.5.1 多任务工作原理 **错误!未定义书签。**](#_Toc358850552)

[2.5.2 多任务切换 **错误!未定义书签。**](#_Toc358850553)

[3 模块详细设计 **错误!未定义书签。**](#_Toc358850557)

[3.1 系统引导及其初始化 **错误!未定义书签。**](#_Toc358850558)

[3.2 UART串口驱动设计 **错误!未定义书签。**](#_Toc358850559)

[3.3 虚拟内存映射 **错误!未定义书签。**](#_Toc358850560)

[3.3.1 内存映射 **错误!未定义书签。**](#_Toc358850561)

[3.3.2 激活MMU **错误!未定义书签。**](#_Toc358850562)

[3.3.3 EPOS恒等映射 **错误!未定义书签。**](#_Toc358850563)

[3.4 EPOS中断处理 **错误!未定义书签。**](#_Toc358850560)

[3.4.1 初始化中断向量表 **错误!未定义书签。**](#_Toc358850561)

[3.4.2 中断现场保护 **错误!未定义书签。**](#_Toc358850562)

[3.3.3 定时器中断处理 **错误!未定义书签。**](#_Toc358850563)

[3.5 多任务 **错误!未定义书签。**](#_Toc358850560)

[3.5.1 创建多任务 **错误!未定义书签。**](#_Toc358850561)

[3.5.2 任务切换 **错误!未定义书签。**](#_Toc358850562)

[4 效果展示 **错误!未定义书签。**](#_Toc358850566)

[4.1 源码编译操作 **错误!未定义书签。**](#_Toc358850567)

[4.2 系统运行结果 **错误!未定义书签。**](#_Toc358850568)

[致谢 **错误!未定义书签。**](#_Toc358850584)

[参考文献 **错误!未定义书签。**](#_Toc358850585)

# **绪论**

## 选题背景及意义

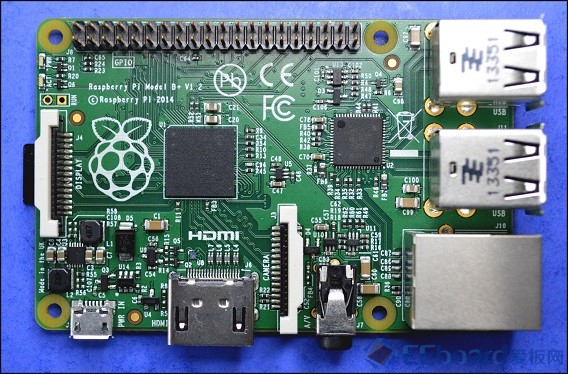
对于计算机专业的学生来说，操作系统是一门非常重要的主干课程，是计算机专业学生必须掌握的一门专业基础课程，对学生以后的在职业道路上有很大的帮助。在操作系统的课程学习中，不仅需要懂得一个操作系统的基本工作原理，还需自己亲自去做实验，学习操作系统的实现方法，这样有助于学生更加深刻的理解操作系统原理。本课题选择将一个轻量级的操作系统EPOS移植到开源硬件Raspberry PI上运行，一方面可以帮助学生理解操作系统的工作原理，同时学生可以直接将操作系统的理论知识转化为代码，并且可以在Raspberry PI上验证自己实验的正确性。另一方面，Raspberry PI在嵌入式领域非常受欢迎，我们将EPOS移植到Raspberry PI上，在嵌入式领域上我们可以使用EPOS来完成部分项目的需求。

## 开发板介绍

树莓派(Raspberry PI)是由英国的树莓派基金会开发的一款信用卡大小的卡片式电脑，其官方提供的系统是基于Linux的Raspbian操作系统。树莓派简称RPI是为学生计算机编程教育而设计，意在提升学校在CS（Computer Science）上的教学，让计算机变得有趣。目前树莓派以及有多个版本，本课题所使用的是Raspberry PI B+开发板，其采用Broadcom BCM2835 700MHz ARM1176JZFS 处理器，带FPU和VideoCore IV双核GPU。

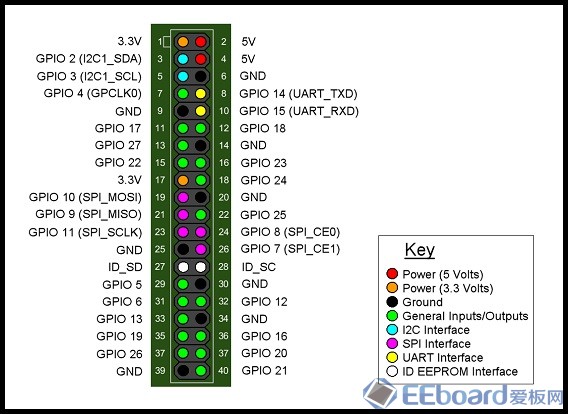
Raspberry PI开发板与普通ARM开发板的略有不同，常见的ARM开发板是上电后直接初始化CPU，树莓派不同的是上电后先驱动GPU，当GPU初始化成功后才会跳转到物理内存为0x8000处去驱动CPU，接下来对arm模式设置并且初始化各种模式下的堆栈，设置栈指针等等操作。

RPI B+提供4个USB2.0接口、512MB SDRAM、 10/100 BaseT RJ45以太网插座、 HDMI 1.3和1.4视频/音频插座、MicroSD 卡插座以及40个外接引脚。下面是Raspberry PI B+的外观图：见图1-2-1(树莓派B+外观图)



**图1-2-1 树莓派B+外观图**

下面是Raspberry PI B+的40个外接引脚功能图：见图1-2-2(引脚图)，本课题中使用的UART的引脚是GPIO14和GPIO15。



**图 1-2-2 RPI B+引脚图**

## 开发环境搭建

本课题使用的开发平台是Ubuntu 14.04 64位，采用GNU 推出的的ARM交叉编译工具arm-none-eabi-gcc，可用于交叉编译ARM MCU芯片，如ARM7、ARM9、Cortex-M/R芯片程序，采用GNU make来构建和管理源码。

EPOS的交叉编译环境如图1-3-1所示，首先通过make将源码编译生成kernel.img文件，然后将kernel.img文件拷贝到FAT32格式的SD卡中，将SD卡插入树莓派卡槽，用串口线将PC与树莓派连接起来(注意：我们使用串口转USB线，将USB接口插入PC机，将串口线与树莓派的外接引脚相连)，开启电源，此时我们就可以在PC机通过minicom软件将串口的数据打印出来。

**图 1-3-1 交叉编译环境**

串口

RPI B+开发板

Ubuntu 主机

# **系统关键技术**

1. ARM体系结构介绍

计算机指令集可分为精简指令集和复杂指令集两种。精简指令集计算机(RISC:Reduced Instruction Set Computing RISC) 是一种执行较少类型计算机指令的微处理器，复杂指令集计算机(Complex Instruction Set Computing ，CISC)以微程序技术为基础的、具有较复杂指令系统的计算机。X86是采用的CISC指令集，而ARM采用RISC指令集。

目前EPOS操作系统是运行在X86体系结构下的，因为X86结构和ARM结构完全不同，所以本课题主要完成的任务是将EPOS所有与X86相关的底层代码都要修改为符合ARM的要求，主要包括启动流程、MMU、中断处理以及多任务的切换。

ARM体系的CPU有两种工作状态：

1、ARM状态：处理器执行32位的字对齐的ARM指令；

2、Thumb状态：处理器执行16位的、半字对齐的Thumb指令；

在程序运行的过程中，可以在两种状态之间进行相应的转换。处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。CPU上电后默认处于ARM状态。

ARM体系的CPU有以下7种工作模式，分别是：

1、用户模式(USR)：用于正常执行程序；

2、快速中断模式(FIQ)：用于高速数据传输；

3、外部中断模式(IRQ)：用于通常的中断处理；

4、管理模式(SVC)：操作系统使用的保护模式；

5、数据访问终止模式(ABT)：当数据或指令预取终止时进入该模式，可用于虚拟存储以及存储保护；

6、系统模式(SYS)：运行具有特权的操作系统任务；

7、未定义指令中止模式(UND)：当未定义的指令执行时进入该模式，可用于支持硬件；

在以上7中模式中，除了用户模式(USR)外，其它6中模式都属于特权模式，在这6中特权模式下，除了系统模式(SYS)外，其它5中模式都属于异常模式，每种异常模式都拥有私有的堆栈指针寄存器R13、返回地址寄存器(R14)以及模式备份寄存器SPSR，程序只有在对于的异常模式下才有权访问该模式下的3个私有寄存器。一般我们只用到用户模式(USR)和系统模式(SYS)两种，其它模式是在处理异常的时候用到。

ARM有两种方法进行工作模式切换：

1、被动切换：在arm运行的时候产生一些异常或者中断来自动进行模式切换。

2、主动切换：通过软件改变，即软件设置寄存器来经行arm的模式切换，

应为arm的工作模式都是可以通过相应寄存器的赋值来切换的。

ARM体系的CPU有以下7种异常状态，见表 2-1 (ARM异常状态)

所谓的异常是指终止了正常执行的程序去执行一些特殊的工作，例如：芯片复位、取指失败等等。我们通常说的中断也是一种异常，这里的中断包括外部硬件产生的外部中断和由芯片内部硬件产生的中断，由中断产生的异常和其它异常，从处理方法的角度看并没有任何区别。

|  |  |  |
| --- | --- | --- |
| **异常类型** | **所属模式** | **说明** |
| 芯片复位 | SVC | 由芯片复位触发 |
| 未定义指令 | UND | 指令不能被识别 |
| 软中断 | SVC | 调用SWI指令触发 |
| 预取异常 | ABT | 没有权限访问寄存器 |
| 数据异常 | ABT | 没有权限访问寄存器 |
| 中断 | IRQ | 由硬件进行中断请求触发 |
| 快速中断 | FIQ | 由硬件进行快速中断请求触发 |

**表 2-1 ARM异常状态**

**ARM寄存器**

ARM一共有37个通用寄存器，这不意味着在程序中我们可以同时使用这37个寄存器中的任何一个，因为这些寄存器是按照模式进行分组的，每个模式下都有属于自己的私有寄存器，程序只有在对应模式下才能够使用对应模式的私有寄存器。共有寄存器只有一组，意思是在所有模式下访问的共有寄存器都是同一个寄存器，系统中的R0~R7、R15以及CPSR各寄存器只有唯一一个，例如：在中断模式和用户用户下访问R0寄存器，其实是在访问同一个寄存器。具体寄存器分组见表 2-2 ARM寄存器：

ABT

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_abt |
| R14\_abt |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_abt |

UND

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_und |
| R14\_und |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_und |

IRQ

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_irq |
| R14\_irq |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_irq |

SVC

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13\_svc |
| R14\_svc |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_svc |

FIQ

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8\_fiq |
| R9\_fiq |
| R10\_fiq |
| R11\_fiq |
| R12\_fiq |
| R13\_fiq |
| R14\_fiq |
| R15(PC) |

|  |
| --- |
| CPSR |
| SPSC\_fiq |

USR /SYS

|  |
| --- |
| R0 |
| R1 |
| R2 |
| R3 |
| R4 |
| R5 |
| R6 |
| R7 |
| R8 |
| R9 |
| R10 |
| R11 |
| R12 |
| R13(SP) |
| R14(LR) |
| R15(PC) |

|  |
| --- |
| CPSR |
|  |

**表 2-2 ARM寄存器**

注：R13寄存器别名为SP，该寄存器是栈指针寄存器。R14寄存器别名为LR，该寄存器是链接寄存器。R15寄存器别名为PC，该寄存器是程序计数器。CPSR寄存器全名是：Current Program Status Register，当前程序状态寄存器。每一种运行模式下又都有一个专用的物理状态寄存器，称为SPSR(Saved Program Status Register，备份的程序状态寄存器)，当异常发生时，SPSR用于保存CPSR的当前值，从异常退出时则可由SPSR来恢复CPSR。由于用户模式和系统模式不属于异常模式，它们没有SPSR，当在这两种模式下访问SPSR，结果是未知的。

1. Bootloader设计
2. Raspberry PI启动流程介绍

树莓派的启动方式和普通的arm不一样。主要是由树莓派的硬件决定的。树莓派芯片上有一个arm11的CPU，还有一个GPU用来处理图形。树莓派的启动是GPU和CPU协同工作的：

首先： 上电，上电后首先运行的是GPU里面的一个小核心，这个小核心的功能就是用FAT32挂载SD卡。

挂载SD卡后，调用我们放在sd卡boot分区里面的bootcode.bin到内存运行，这个就是我们通常说的bootloader， 再调用start.elf到内存，然后再启动GPU,GPU运行start.elf的程序，开始一系列的配置。再调用kernel.img即系统内核到内存。

最后，GPU将控制权交给CPU，也就是完成系统内核的启动。

注：本课题中所使用的bootcode.bin、start.elf是从树莓派官网下载，kernel.img文件则是将我们的系统源码编译后生成的镜像文件。CPU开始运行的地址不是在0x0处，而是CPU从物理地址的0x8000处开始运行。

1. EPOS初始化以及加载kernel

EPOS中，首先是定义7中异常的服务函数，也就是定义中断向量表，然后EPOS对7中模式进行初始化，首先通过设置CPSR寄存器，将当前模式修改成对应模式，为该模式进行设置栈指针。最后切换当前模式为SVC模式，使用bl指令跳转到C语言函数进行其他操作，到此完成了EPOS的启动。

1. ARM虚拟内存
2. MMU工作原理

MMU是Memory Management Unit的缩写，MMU负责的是虚拟地址与物理地址的转换，提供硬件机制的内存访问授权。图2-3-1-1 展示了MMU的作用，图中是将虚拟内存0x00008000映射到物理内存0xC0000000处。

虚拟内存 物理内存

**图 2-3-1-1 MMU作用**

0x00008000

0xC0000000

MMU

MMU中一个重要的概念就是页表(page table)，页表就是存储在内存中的一张表，在表中记录着将虚拟内存转化为物理内存的关键信息，MMU正是通过查询表的数据，实现了将虚拟地址转化为物理地址。也就是说，MMU每次工作的时候都要去查这张表，从表中找出与虚拟地址相对应的物理地址，然后再进行数据存取操作。页表的作用如图 2-3-1-2所示：

虚拟内存 物理内存

**图 2-3-1-2 页表作用**

0xC0000000

MMU

0x00008000

TLB

页表

条目被称为页表项(page table entry)，一条页表项负责记录一段虚拟地址空间到物理地址空间的映射，这里所说的虚拟地址空间被称为页，一个页对应页表的中的一项。页的大小是可以选择的，通常ARM支持页大小为1K、4K、16K和1M，分别叫做微页、小页、大页和段页。本课题所采用的小页映射，所以接下来我们主要讨论小页映射。

对于4K大小的页，MMU采用二级查表的方法，首先由虚拟地址索引出第一张表的某些字段，然后更具这些字段索引出第二张表，最后更具第二张表的内容确定出该虚拟地址对应的物理地址是多少，这里的第一张表我们称为页目录，第二张表我们称为小页表。采用二级页表查询物理地址的目的是减少页表自身占用的内存空间，但缺点是进一步减少了内存的寻址效率。

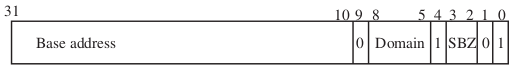
下面介绍二级页表结构和小页表结构以及虚拟地址结构图，图2-3-1-3为二级页表结构，图2-3-1-4为小页表结构，图2-3-1-5位虚拟地址结构图，二级页表的高22位存放的是小页表的基地址，而小页表项的高20位存放的是物理地址的基地址。虚拟地址的高12位(20~31位)存放的是一级页表的偏移量，第12~19位存放的是小页表项的偏移量，低12位(0~11位)存放的是物理地址的偏移量。接下来我们用实际的案例来理解MMU工作原理见图2-3-1-6。

首先，我们要让MMU知道页目录在物理内存中的基地址，因为页目录是我们通过程序确定的，存储在什么位置自然是知道的，然后CPU需要寻址的时候，MMU就可以自动的将虚拟地址映射为物理地址，具体步骤如下：

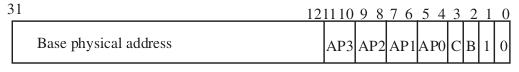
1、MMU取出虚拟地址的高12位作为页目录的偏移量，结合页目录的基地址取出对应的页目录项，具体操作是将这12位取出，然后左移两位和页目录基地址相加操作，这样就得到页目录项的地址了。例如：页目录的基地址是0x20000000，假如虚拟地址是0x00101001，则高12位是0x001，那么左移两位是0x004，那么得到的页目录项的地址是0x20000000+0x004=0x20000004，接下来从该地址读取的数据是该虚拟地址对应的小页表的基地址。

2、找到虚拟地址对应的小页表的基地址后，MMU取出虚拟地址的12~19位，结合小页表的基地址取出对应的小页表项，具体操作是将这8位取出，然后左移两位和小页表的基地址进行相加操作，这样就得到了小页表项的地址了。例如：由1取出的数据是0x30000001，那么我们知道小页表的基地址是0x30000000，此时虚拟地址的中间8位是0x01左移两位后是0x04，那么小页表项的地址是0x30000000+0x04=0x30000004，从该地址读取的数据是该虚拟地址对应的物理地址的基地址。

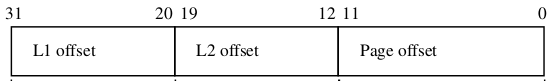
3、从小页表项中读取的32位数据就是包含物理地址的基地址，接下来，将虚拟地址的低12位与物理地址的基地址进行相加操作，最后得到的就是该虚拟地址对应的物理地址。例如：由2中读取的数据是0xC0000001，虚拟地址的低12位是0x001，那么得到的物理地址是0xC0000000+0x001=0xC0000001，那么虚拟地址0x00101001经过MMU查找后对应的物理地址是0xC0000001。



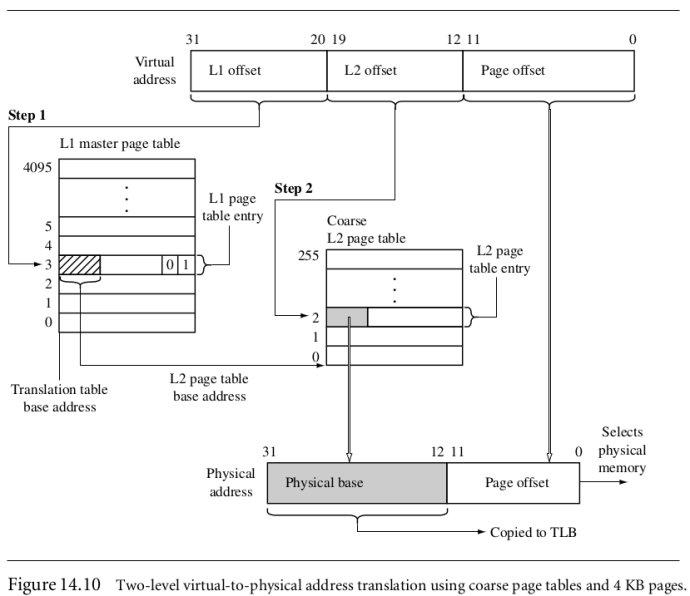
**图 2-3-1-3 二级页表结构**



**图 2-3-1-4 页表项结构**



**图 2-3-1-5 虚拟地址结构**



**图 2-3-1-6 MMU工作原理**

注意：虚拟地址的高12位表示页目录项的偏移量，所以一共有2^12=4096项页目录项，每项占4个字节，所以页目录总共占内存16K，其中每一项对应着一张小页表的基地址，虚拟地址的中间8位表示小页表项的偏移量，所以一张小页表包括2^8=256项小页表项，一项占4个字节，所以一张小页表占1K大小。

1. CP15协处理器

MMU的配置都是通过CP15协处理器来完成的，ARM支持16个协处理器，当然并不是所有的ARM体系结构都支持。CP15是负责系统控制的，系统中的cache、write buffer、MMU等等都是通过CP15协处理器来控制的。协处理器的访问不能使用常规的指令。

ARM寄存器到协处理器寄存器的数据传送指令如下：

MCR{<cond>} p15，0，<Rd>,<CRn>,<CRm>{,<opcode\_2>}

协处理器寄存器到ARM寄存器的数据传送指令如下：

MRC{<cond>} p15，0，<Rd>,<CRn>,<CRm>{,<opcode\_2>}

其中，<cond>为指令执行的条件码。当<cond>忽略时指令为无条件执行。

<Rd>作为源寄存器的ARM寄存器，其值将被传送到协处理器寄存器中。

<CRn>作为目标寄存器的协处理器寄存器，其编号可能是C0，C1，…，C15。

<CRm>和<opcode\_2>两者组合决定对协处理器寄存器进行所需要的操作，如果没有指定，则将为<CRm>为C0，opcode\_2为0，否则可能导致不可预知的结果。

1. EPOS恒等映射

EPOS操作系统中将虚拟内存空间0~3G划分给用户模式使用，将3G~4G空间划分给内核使用。由于ARM启动的时候是从地址空间0x0处开始执行，要想我们将kernel定位到3G以上，那么我们采用恒等映射来完成，具体操作如下：

首先，我们将kernel的物理地址映射为两份，一份映射的虚拟地址和物理地址一致，另一份是将kernel的物理地址映射到3G以上。

上电后，CPU从物理地址0x0处开始执行程序，当激活MMU后，CPU运行在虚拟地址空间上，此时的PC寄存器任然指向3G以下的虚拟空间运行，此时，我们通过修改PC寄存器将CPU跳转到3G以上去运行。

最后，我们将3G以下的kernel虚拟地址空间清除掉，这样就保证虚拟地址控制3G以下留给用户使用，3G以上供内核使用。

图2-3-3-1显示了EPOS恒等映射。

虚拟内存

物理内存

**图 2-3-3-1 恒等映射**

4G

Kernel

3G

MMU

Kernel

Kernel

Kernel

0x0

1. EPOS page fault处理

产生Page fault的原因是程序访问一个虚拟地址的时候，经过MMU转化后没有找到对应的物理地址，则此时会产生一个Page fault错误，简单的说就是由于缺页找到Page fault错误。

产生Page fault的方式有两种，一种是通过预取指令异常产生，当程序执行在某一地址加载一个指令的时候，无法获取该指令，则此时向系统发出Page fault错误。另一种是通过数据异常产生，当程序向一个地址写或者读取数据的时候，如果没有权限进行写或者读操作，那么此时向操作系统发出一个Page fault错误。

处理Page fault的大致方法：首先获取触发Page fault的地址，然后获取错误代码，同时保存上下文，进入对应的异常处理函数后，通过调用do\_page\_fault(由操作系统实现处理page fault的函数)函数来进行操作。

ARM获取Page fault地址和错误代码是通过访问IFAR、DFAR、IFSR和DFSR寄存器来获取。IFAR全称Instruction Fault Address Register，该寄存器中存放的是指令预取中止异常时的地址，IFSR(Instruction Fault Status Register)则存放着对应的错误代码值。DFAR全称Data Fault Address Register，该寄存器中存放的是数据中止异常时的地址，DFSR(Data Fault Status Register)则存放着对应的错误代码值。下面是对以上四个寄存器的操作：

MRC p15, 0, <Rt>, c6, c0, 2 ; 读取 IFAR 到 Rt

MCR p15, 0, <Rt>, c6, c0, 2 ; 将 Rt 写入 IFAR

MRC p15, 0, <Rt>, c5, c0, 1 ; 读取 IFSR 到 Rt

MCR p15, 0, <Rt>, c5, c0, 1 ; 将 Rt 写入 IFSR

MRC p15, 0, <Rt>, c6, c0, 0 ; 读取 DFAR 到 Rt

MCR p15, 0, <Rt>, c6, c0, 0 ; 将 Rt 写入 DFAR

MRC p15, 0, <Rt>, c5, c0, 0 ; 读取 DFSR 到 Rt

MCR p15, 0, <Rt>, c5, c0, 0 ; 将 Rt 写入 DFSR

1. ARM中断处理
2. 中断机制
3. 中断向量表
4. Raspberry PI 定时器
5. 多任务处理
6. 多任务工作原理
7. 多任务切换

# **模块详细设计**

1. 系统引导及其初始化
2. UART串口驱动设计
3. 虚拟内存映射
4. 内存映射
5. 激活MMU
6. EPOS恒等映射
7. 虚拟内存映射
8. 初始化中断向量表
9. 中断现场保护
10. 定时器中断处理
11. EPOS中断处理
12. 创建多任务
13. 任务切换

# **效果展示**

1. 源码编译操作
2. 系统运行结果