重庆大学本科学生毕业设计（论文）

基于PC机的多任务

操作系统内核的设计与实现



学 生：耿渊哲

学 号：20092048

指导教师：洪明坚 副教授

专 业：软件工程

重庆大学软件学院

二0一三年六月

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

**Design and Implementation of PC-based Multi-tasking operating system kernel**



**Undergraduate: Geng Yuanzhe**

**Instructor: Associated Prof.Hong Mingjian**

**Major: Software Engineering**

**School of Software Engineering**

**Chongqing University**

**June 2013**

# 摘要

操作系统，作为管理计算机硬件和软件资源的计算机程序，更是软件工业的基础与核心。然而，操作系统的市场却几乎被国外所占据，国内尚未有商业级的操作系统。对操作系统的研发国内也仅仅是对国外优秀系统的模仿上，没有自己的创新。同时，操作系统的教学主要停留在系统基本的工作原理，没有系统的完整实现。

多任务操作系统内核便是针对这一现状，对操作系统进行实践和研究，为众多计算机学习人员打造一个学习和研究的平台。不同于那些庞大复杂的操作系统，本多任务内核，基于操作系统最核心的内容，利用djgpp，bochs，makefile等工具，构建了一个完整清晰的架构。同时，该平台支持各方面的功能扩展，允许学生对平台进行自由的研究和完善。

本文的主要工作：

1. 对内核开发环境进行说明，对系统关键技术进行讲解。
2. 对多任务操作系统内核进行设计，对系统引导和系统初始化进行简单介绍，对任务管理模块进行详细设计。
3. 编码实现多任务操作系统内核，描述任务管理各项功能的实现方法，对核心代码进行详细的解释。
4. 多任务操作系统内核的测试，对系统的总体功能，以及函数的主要功能进行黑盒测试，对关键函数的分支进行白盒测试，最后给出一个综合测试的案例，直观地对对测试结果进行展示，验证系统的健壮与稳定性。

**关键词：**操作系统内核，任务管理，稳定性

# ABSTRACT

Operating System, as computer hardware and software resources management program, is the foundation and core of the software industry. However, the operating system market is almost occupied by foreign and there is no commercial grade operating system in our country. The development of the operating system in China is only an imitation of foreign excellent system and we don’t have our own innovations. At the same time, the operating system teaching still focuses on the basic system working principle, without the complete implementation.

In response to this program, the Multi-tasking operating system kernel appears.

We study and practice the operating system, and create a learning platform for many computer learning and research staff. Unlike those large and complex operation systems, this multitasking kernel, based on the core of the operating system, build a complete and clear structure with tools such as djgpp, bochs and makefile. Meanwhile, the platform supports extensions and students using the platform are allowed to study and improve it freely.

This paper’s major work,

1. Introduce the development environment of kernel, and explain the key technologies of the system.
2. Design the multi-tasking operating system kernel. Introduce the system booting and the system initialization and design the task module in detail.
3. Implement the system kernel, describe the implementation methods and explain the core code.
4. Multi-tasking operation system kernel testing. Test the overall function of the system as well as the function of the main function with black box and the branch of the core functions with while box. At last, demonstrate the result and verify the robustness and stability of the system.

**Key word:** Operating System Kernel, Task Management, Stability

# 目录

[摘要 I](#_Toc358850539)

[ABSTRACT II](#_Toc358850540)

[目录 III](#_Toc358850541)

[1 绪论 1](#_Toc358850542)

[1.1 选题背景及意义 1](#_Toc358850543)

[1.2 国内外研究现状 2](#_Toc358850544)

[1.3 本文的主要内容及结构 3](#_Toc358850545)

[2 系统关键技术 4](#_Toc358850546)

[2.1 系统内核开发环境 4](#_Toc358850547)

[2.2 AT&T汇编和GCC内嵌汇编 4](#_Toc358850548)

[2.2.1 AT&T与Intel汇编语言语法的区别 4](#_Toc358850549)

[2.2.2 GCC内嵌汇编 6](#_Toc358850550)

[2.3 中断处理 7](#_Toc358850551)

[2.3.1 中断机制 7](#_Toc358850552)

[2.3.2 中断描述符表 8](#_Toc358850553)

[2.4 Make程序和Makefile文件 9](#_Toc358850554)

[2.5 Bochs调试简介 11](#_Toc358850555)

[2.6 本章小结 12](#_Toc358850556)

[3 多任务操作系统内核设计 13](#_Toc358850557)

[3.1 系统引导 13](#_Toc358850558)

[3.2 系统的初始化 13](#_Toc358850559)

[3.3 任务管理 14](#_Toc358850560)

[3.3.1 内核线程控制 14](#_Toc358850561)

[3.3.2 线程的运行状态 14](#_Toc358850562)

[3.3.3 任务数据结构 16](#_Toc358850563)

[3.3.4 任务管理接口设计 17](#_Toc358850564)

[3.4 本章小结 20](#_Toc358850565)

[4 多任务操作系统内核实现 21](#_Toc358850566)

[4.1 程序目录 21](#_Toc358850567)

[4.2 任务管理模块的实现 21](#_Toc358850568)

[4.2.1 线程调度接口实现 21](#_Toc358850569)

[4.2.2 定时器线程调度实现 28](#_Toc358850570)

[4.3 本章小结 29](#_Toc358850571)

[5 多任务操作系统内核测试 30](#_Toc358850572)

[5.1 测试计划 30](#_Toc358850573)

[5.1.1 测试目标 30](#_Toc358850574)

[5.1.2 测试的函数 30](#_Toc358850575)

[5.2 测试用例 30](#_Toc358850576)

[5.2.1 系统功能测试 30](#_Toc358850577)

[5.2.2 函数结构测试 34](#_Toc358850578)

[5.3 综合测试 37](#_Toc358850579)

[5.4 本章小结 38](#_Toc358850580)

[6 总结与展望 39](#_Toc358850581)

[6.1 总结 39](#_Toc358850582)

[6.2 展望 39](#_Toc358850583)

[致谢 40](#_Toc358850584)

[参考文献 41](#_Toc358850585)

# 绪论

## 选题背景及意义

在IT产业迅猛发展的今天，计算机已经深入到人们生产和生活之中，小到便捷工作，带来娱乐，大到自动化控制、军事模拟、能源勘探，计算机都发挥着无容置疑的巨大作用。操作系统，位于底层硬件与用户之间，作为管理计算机硬件和软件资源的计算机程序，为用户提供快捷、安全且有效率的操作方式。在计算机科学中，操作系统理论是历史悠久而又活跃的分支，而操作系统的设计与实现则是软件工业的基础与核心。

操作系统从无到有，从人工操作阶段、管理程序阶段、批处理系统、程序系统、再到分时系统、实时系统以及现在的通用操作系统。在短短的几十年内，操作系统取得了长足的发展。如今，世界上已经存在很多成功的作品，如：IMB公司的MS-DOS操作系统，AT&T公司的贝尔实验室开发的Unix操作系统，微软公司的Microsoft Windows系列，以及由Linus最初编并由写全世界的计算机爱好者共同参与开发的开源操作系统Linux等。

作为计算机专业的学生，在操作系统的课程学习中，不仅需要懂得系统的基本工作原理，还需要结合源代码，学习系统的实现方法，因为理论毕竟是理论，其中省略了很多细节，而这些细节问题虽然没有太大的理论含量，但它们是一个系统的必要的组成部分。Linux作为当前世界上用户最多的开源操作系统，且该系统的初衷是用于的学习使用。因此，阅读研究Linux的源代码，同时动手实现一个可运行的多任务操作系统内核成为必然。

多任务操作系统内核是源于Linux操作系统，并进行简化，使用户更容易理解和接受。该内核抛弃和简化了大部分的操作系统功能，而把注意力放在多任务部分，主要进行线程管理和中断调用，使线程的初始化、创建、调度以及中断调用等操作更加清晰明了。通过对该内核的学习，一方面对系统的运作原理有一个非常完整而实际的概念，同时为更深入的学习内核代码打下坚实的基础。本操作系统内核使用GUN gcc 编译后，可运行在Bochs仿真环境下。同时，该系统还可以作为一个多任务系统基础平台供学生或者爱好者进行研究，更可以进行扩展，以实现更加完善和复杂的功能。

## 国内外研究现状

纵观电脑的发展历史，操作系统与电脑硬件的发展息息相关。从最早的批次模式开始时，分时机制随之出现，在多处理器时代来临时，操作系统也随机添加多处理器协调功能，甚至是分布式系统的协调功能。然而就像计算机一样，操作系统的产生和发展，都是在国外进行，而操作系统的市场也均被国外所占据，

从服务器端出类拔萃的Unix操作系统，到开源易用的Linux系统，再到稳定、高性能的Mac OS,以及占据中国95%用户的Windows系统。

此外，对于操作系统的研究，国外也较国内有较大的领先。而且觉大部分国外的公司，对其操作系统的内核代码，采取了保密的做法。微软和苹果公司把其Windows和Mac OS的源代码作为商业机密，而Unix的源代码也仅仅在得到贝尔实验室许可的大学中用于教学而不得公开。直到作为教学使用的Minix操作系统的出现，Linus参考它开发出自己的操作系统Linux，并将其作为开源项目公布于众，操作系统的研究在全世界计算机开好这的推动下快速发展越来越多大中型企业及政府投入更多的资源来开发Linux，越来越多国家逐渐的把政府机构使用的计算机转移到Linux上。

在国内，国防科技大学计算机学院着手研究Unix操作系统，并把银河巨型机操作系统的技术路线从专用的向量巨型机操作系统转变为Unix操作系统。之后随着国际Linux热潮席卷中国，许多大型企业纷纷投身于Linux国产化研发，先后发布了各自的国产Linux操作系统，随后政府“软件正版化”运动的开始以及“核高基”重大专项政策的出台，使国产操作系统企业逐步发展起来。但是我国广泛应用的主流操作系统仍是国外引进直接使用的产品，其安全性难以令人放心。我国在安全操作系统方面已经展开一些工作，但是缺乏理论基础，也就是缺乏对安全模型和安全评估准则的深入研究，使得安全操作系统方案缺乏整体性。同时国内基本都是利用国外的技术甚至是部分源代码，根据市场需要自己组合成的操作系统，这种系统不具有我们的自主版权。以Linux为代表的国际自由软件的发展为我国发展具有自主版权的系统软件提供了良好的机遇，但Linux内核设计缺乏模块化，从而导致基于Linux平台研制的安全操作系统的做法缺乏科学的安全模型支持。

对于操作系统内核研究，国内外教材较多。国外教材如《Unix操作系统设计》UNIX System V内核的工作原理和数据结构， 《Programming the 80386》讲解80x86下保护模式编程方法以及《MINIX作系统设计与实现》提供文件系统方面的诸多知识。国内如赵炯博士在其《Linux内核完全注释》中，针对Linux内核代码0.11，对操作系统内核的工作原理进行深入的剖析和解释，对于初学者是一本不可多得的材料，也是本文主要参照。

## 本文的主要内容及结构

本文主要分为以下几部分：

第一章，绪论。论文本课题的研究背景、意义以及操作系统在国内外的发展和研究现状；

第二章，系统关键技术。对系统开发使用的关键技术进行介绍。主要包括内核开发环境，AT&T汇编与GCC内嵌汇编的语法，中断处理机制，Bochs调试，以及Makefile文件的使用方法；

第三章，系统设计。主要对任务数据结构和任务的接口进行详细设计。并详细说明各个接口的功能。

第四章，系统实现。编码实现任务管理的各个接口，并对核心代码进行阐释。

第五章，系统测试。针对系统的功能性和分支进行测试。验证系统的健壮性与安全性。

第六章，展望与总结。对整个开发工作的总结还是以及对未来的展望。

# 系统关键技术

## 系统内核开发环境

本多任务操作系统内核使用GNU C语言和汇编作为编程语言，采用djgpp作为开发平台，使用make工具以及makefile文件对项目文件进行编译和链接，最终运行在仿真平台bochs下。

djgpp(DJ’s GNU Programming Platform)是基于386IMB兼容机的开发套件，兼容DOS操作系统，支持多种GNU开发工具接口，它包含了C的编译器。Make工具是来决定项目中那些文件需要重新编译，以及重编译的命令。而makefile文件则是make工具的配置文件，告诉make如何来编译和连接程序。Bochs是一个开放源代码的PC仿真软件，它可以模拟x86的外围环境，同时还仿真了所有的指令，并且含有标准PC机外设所有的设备模块，使在其中执行的软件“认为”自己真实地运行在一台PC中。

## AT&T汇编和GCC内嵌汇编

内核中的汇编语言程序均使用GAS(GNU AS汇编器)来编译，C语言则用gcc (GNU Compiler Collection)来编译。不论是GAS汇编，还是GCC中的C内嵌汇编，都遵从AT&T汇编语法，下面先介绍AT&T汇编语言语法与Intel汇编语法的差别，然后介绍GCC内嵌汇编语法。

### AT&T与Intel汇编语言语法的区别

AT&T和Intel汇编语言的语法区别主要体现在操作数前缀、赋值方向、间接寻址语法、操作码后缀上，而就具体的指令而言，在统一平台上，两种汇编语言是一致的。下面仅列出这两种汇编语言在语法上的不同点。Intel汇编语言语法可参考i386手册。

1. 操作数前缀

在Intel的汇编语言语法中，寄存器和立即数都没有前缀。但是在AT&T中，寄存器前冠以“%”，而立即数前冠以“$”。在Intel的语法中，十六进制和二进制立即数后缀分别冠以“h”和“b”，而在AT&T中，十六进制立即数前冠以“0x”，表2.1给出几个相应的例子。

表2.1 Intel与AT&T汇编语言的指令操作数前缀

|  |  |
| --- | --- |
| Intel语法 | AT&T语法 |
| mov ax, 8 | movl $8,%eax |
| mov ebx,0ffffh | movl $0xffff, %ebx |
| int 80h | int $0x80 |

1. 源/目的操作数顺序

如表2.1中所看到的，Intel汇编语言的指令与AT&T的指令操作数的方向上正好相反：在Intel语法中，第一个操作数是目的操作数，第二个操作数是源操作数。而在AT&T中，第一个是源操作数，第二个是目的操作数。如表2.2所示：

表2.2 Intel与AT&T汇编语言的指令操作数赋值方向

|  |  |
| --- | --- |
| Intel语法 | AT&T语法 |
| MOV EAX,8 | movl $8,%eax |

上表所举例子中，我们可以看到，Intel规定目的操作数eax在前而源操作数8在后，AT&T则正好相反。

1. 寻址方式

Intel语法的间接内存引用形式：section: [base + index\*scale + disp]

对应于如下AT&T语法形式：section: disp(base, index, scale)

其中base和index是可选的32位基寄存器和索引寄存器，disp是可选的偏移值。Scale是比例因子，取值范围是1、2、4、8。Scale乘上索引index用来计算操作数地址。如果没有指定scale,则scale取默认值1。Seg为内存操作数指定可选的段寄存器，并且会覆盖操作数使用的当前默认段寄存器。表2.3是几个AT&T语法形式的内存引用例子：

表2.3 AT&T汇编语言间接寻址示例

|  |  |
| --- | --- |
| AT&T汇编语句 | 解释 |
| movl var, %eax | 把内存地址var处的内容放入寄存器%eax中 |
| movl %cs:var, %eax | 把代码段中内存地址var处的内容放入%eax中 |
| movl $var, %eax | 把var的地址放入%eax中 |
| movl array(%esi), %eax | 把array+%esi确定的内存地址处的内容放入%eax中 |
| movl (%ebx, %esi, 4), %eax | 把%ebx+%esi\*4确定的内存地址处的内容放入%eax中 |
| movl array(%ebx, %esi, 4), %eax | 把array+%ebx+%esi\*4确定的内存地址的内容放入%eax |
| movl -4(%ebp), %eax | 把%ebp-4内存地址处的内容放入%eax中，默认段%ss |
| movl foo(, %eax, 4), %eax | 把内存地址foo+%eax\*4处内容放入%eax中，默认段%ds |

1. 标识长度的操作码前缀和后缀

在AT&T汇编中远程跳转指令和子过程调用指令的操作码使用前缀“l”，分别为ljmp，lcall，与之相应的返回指令为lret。如表2.4所示：

表2.4 Intel与AT&T汇编语言的操作码前缀区别

|  |  |
| --- | --- |
| Intel 语法 | AT&T语法 |
| CALL SECTION:OFFSET | lcall $section:$offset |
| JMP FAR SECTION:OFFSET | ljmp $section:$offset |
| RET FAR STACK\_ADJUST | lret $stack\_adjust |

在AT&T的操作码后面有时还会有一个后缀，其含义就是指出操作码的大小。“l”表示长整形(32位)，“w”表示字(16位)，“b”表示字节(8位)。而在Intel的语法中，则要在内存单元操作数的前面加上dword ptr、word ptr和byte ptr，“dword”对应“long”。表2.5给出几个相应的例子。

表2.5 Intel与AT&T汇编语言的操作码后缀区别

|  |  |
| --- | --- |
| Intel语法 | AT&T语法 |
| Mov al, bl | Movl %bl, %al |
| Mov ax, bx | Movw %bx, %ax |
| Mov eax, ebx | Movl %ebx, %eax |
| Mov eax, dword ptr[ebx] | Movl (%ebx), %eax |

### GCC内嵌汇编

GCC提供了内嵌汇编的功能，可以在C代码中直接内嵌汇编语言语句，大大方便了程序设计。这里对其基本格式进行简单的描述，详细的说明可参见GNU gcc手册。嵌入汇编的基本格式为：

|  |
| --- |
| asm(“汇编语句”  :输出寄存器  :输入寄存器  :会被修改的寄存器); |

其中“汇编语句”是程序员写汇编指令的地方；“输出寄存器”表示当这段嵌入汇编执行之后，哪些寄存器用于存放输出数据。这是寄存器会分别对应一个C语言表达式或一个内存地址；“输入寄存器”表示在开始执行汇编代码时，这里指定的一些寄存器中应存放的输入值，他们也分别对应着一个C变量或常数值。

表2.6中是一些可能会用到的寄存器加载代码及其具体的含义。

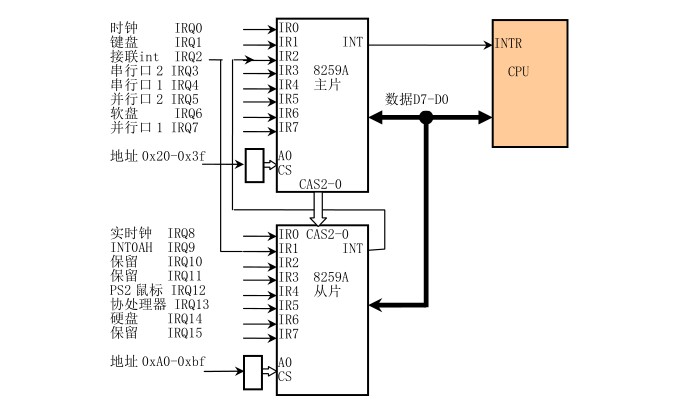
表2.6 常用寄存器加载代码说明

|  |  |  |  |
| --- | --- | --- | --- |
| 代码 | 说明 | 代码 | 说明 |
| a | 使用寄存器eax | m  o | 使用内存地址 |
| b | 使用寄存器ebx | 使用内存地址并可以加偏移值 |
| c | 使用寄存器ecx | i | 使用常数0-31 |
| d | 使用寄存器edx | j | 使用常数0-63 |
| S | 使用esi | K | 使用常数0-255 |
| D | 使用edi | L | 使用常数0-65535 |
| q | 使用动态分配字节可寻址寄存器（eax、ebx、ecx、edx） | M | 使用常数0-3 |
| r | 使用任意动态分配的寄存器 | N | 使用1字节常数（0-255） |
| g | 使用通用有效的地址即可 | O | 使用常数0-31 |
| A | 使用eax与edx联合（64位） | = | 输出操作数。输出值将替换前值 |
| + | 标识操作数可读可写 | & | 早期会变的操作数 |

## 中断处理

### 中断机制

在使用80x86组成的PC机中，采用了两片8259A可编程中断控制芯片。每片可以管理8个中断源，共可管理15级中断向量。其级联示意图见图2.1。主8259A芯片的端口基地址是0x20，从芯片是0xA0。其中主片的INT引脚连接到CPU上，而从片的INT引脚用于和主片相连。在总线控制器控制下，8259A芯片可以处于变成状态和操作状态。在编程状态下，CPU使用IN或OUT指令对8259A芯片进行初始化编程的。当初始化完成，芯片即进入操作状态，此时芯片可以随时响应外部设备发出的中断请求 (IRQ0~IRQ15)。通过中断判优选择，芯片将选中当中最高优先级的中断请求作为中断服务对象，并通知CPU外中断请求的到来，CPU响应后，芯片从数据总线D7~D0将编程设定的当前服务对象的中断号送出，CPU由此获取对应的中断向量值，并执行中断服务程序。

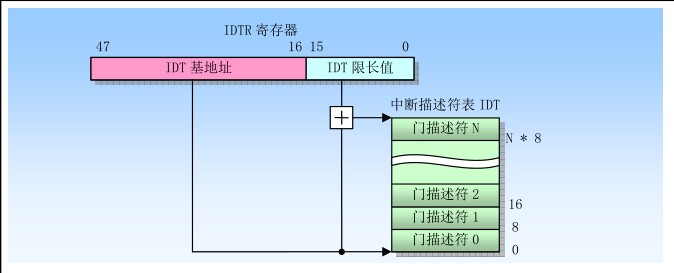
图2.1 PC/AT微机级联式8259控制系统

在操作系统内核中，每个中断是由0~255之间的一个数字来标识，称为中断向量号。中断int0~int31(0x00~0x1f)由Intel公司固定设定或保留用，属于软件中断，也称之为异常。因为这些中断是在CPU执行指令时探测到异常情况而引起的，如除0异常和缺页异常等。中断int32~int255(0x20~0xff)可以由用户自己设定。在本系统中，则将int32~int47(0x20~0x2f)对应于8259A中断控制器芯片发出的硬件中断请求号IRQ0~IRQ15。

### 中断描述符表

中断描述符表IDT将每个异常或中断向量分别与他们的处理过程联系起来。与GDT和LDT表类似，IDT也是由8字节长描述符组成的一个数组。与GDT不同的是，表中第一项可以包含描述符。为了构成IDT表中的一个索引值，处理器把异常或中断的向量号乘以8。因为最多只有256个中断或异常向量，所以IDT无需包含多余256个描述符。IDT中可以含有少于256个描述符，因为只有可能发生的异常或中断才需要描述符。

IDT表可以驻留在线性地址空间的任何地方，处理器使用IDTR寄存器来定位IDT表的位置。这个寄存器中含有IDT表32位的基地址和16位的长度(限长)值，见图2.2所示。IDT表基地址应该对其在8字节边界上以提高处理器的访问效率。限长值是以字节为单位的IDT表的长度。

图2.2 中断描述符表IDT和寄存器IDTR

指令LIDT和SIDT分别用于加载和保存IDTR寄存器的内容。LIDT指令把内存中的限长值和基地址操作数加载到IDTR寄存器中。SIDT指令用于把IDTR中的基地址和限长内容复制到内存中。

如果中断或异常向量引用的描述符超过了IDT的界限，处理器会产生一个一般保护性异常。

## Make程序和Makefile文件

Make工具是来决定一个大型项目中的哪部分需要重新编译，并确保重编译的命令。而makefile文件则是make工具的配置文件，它告诉make工具做什么，通常告诉make如何来编译和连接程序。下面仅对makefile文件做简单介绍，详细说明参照GNU make manual。

当make工具重编译程序时，每个改变的C源文件都必须重新编译。如果一个头文件改变，那么每个包含这个头文件的C源文件为了保证安全也需要重新编译。每个编译操作都产生一个与源文件相对应的目标文件。如果任何一个源文件被重编译，所有不管是最新生成还是之前编译产生的目标文件都必须被连接到一起来产生最终的可执行程序。

简单的makefile含有一些规则，这些规则具有如下形式：

|  |
| --- |
| target(目标)… : prerequisites(先决条件)…  recipe(行为)  …  … |

“目标”通常是一个程序生成的文件的名字，例如可执行文件或者目标文件。目标文件也可以是要采取的活动的名字，比如“clean”。“先决条件”是一个或多个文件名，作为创建目标的输入。“行为”则是make要执行的操作。一个规则可以有很多命令，每个命令自成一行。在每个“行为”行开始之前加一个制表符，这是很多人容易忽略的地方。

这里讨论一个简单的makefile文件，它描述如何编译和连接一个包含3个C源文件和3个头文件的文本编辑器程序。

|  |
| --- |
| edit: main.o command.o display.o  cc -o edit main.o command.o display.o  main.o: main.c defs.h  cc -c main.c  command.o: command.c defs.h command.h  cc -c command.c  display.o: display.c defs.h buffer.h  cc -c display.c  clean:  rm edit main.o command.o display.o |

为了让这个makefile创建这个名为edit的可执行文件，输入：

|  |
| --- |
| make |

为了使用这个makefile删除可以执行文件以及所有的目标文件，输入：

|  |
| --- |
| make clean |

在这个makefile例子中，目标包括可执行文件“edit”，以及目标文件“main.o”、“command.o”和“display.o”。先决条件是“main.c”，“defs.h”等文件。事实上，每个“.o”既是“目标”又是“先决条件”。行为包括“cc –c main.c”和“cc –c command.c”等。

当目标是一个文件时，在先决条件改变时它需要被重新编译或重新连接。另外任何先决条件，如果他们自己是自动生成的话，他们要最早被更新。在这个例子中，edit依赖于3个目标文件，目标文件main.o依赖于源文件main.c和头文件defs.h。

行为在目标和先决条件之后，来说明如何更新目标文件。行为的每行之前要都有一个制表符。目标“clean”并不是一个文件，而仅仅是一个活动的名字。因为通常并不想在规则里使用这个活动，“clean”并不是先决条件也没有任何的先决条件。因此如果用户不专门说明，make程序是不会使用这个活动的。

## Bochs调试简介

运行操作系统内核最佳的方法是使用PC仿真软件。Bochs属于其中的一种，并且是开放源代码软件。Bochs可以仿真x86的硬件环境及其外围设备，因此可以移植到很多操作系统上或者不同体系结构的平台上。

Bochs是一个能完全仿真Intel x86计算机的程序。它可以被配置成仿真386、486、Pentium或以上的新型CPU。在执行的整个过程，bochs仿真了所有的指令，并且含有标准PC机外设所有的设备模块。由于bochs仿真了整个PC环境，因此在其中执行的软件会“认为”它是在一个真实的机器上运行。

为了在bochs中运行一个操作系统，最少需要一下一些资源或信息：bochs执行文件、bios映像文件、vga bios映像文件、至少一个引导启动磁盘映像文件。但是我们在使用过程中往往需要为运行系统预先设置一些参数。这些参数可以在命令行上传递给bochs执行程序，但通常我们都使用一个配置文件（文件后缀为.bxrc，例如Sample.bxrc）。为专门的一个应用来设置运行参数。

Bochs使用配置文件中的信息来寻找所使用的磁盘映像文件、运行环境外围设备的配置以及其它一些模拟机器的设置信息。每个被仿真的系统都需要设置一个响应的配置文件。在bochs安装的主目录下有一个名称为“bochsrc-sample.txt”的样板配置文件，其中列出了所有可用的参数设置，并带有详细的说明。

bochs具有非常强大的操作系统内核调试功能。具体的bochs下载、安装和调试命令见参考文献《Linux内核完全注释》。表2.7给出常用指令及其作用：

表2.7 bochs调试命令及作用

|  |  |  |
| --- | --- | --- |
| 行为 | 指令 | 举例 |
| 在某物理地址设置断点 | b addr | b 0x30400 |
| 显示当前所有断点信息 | info break | info break |
| 继续执行，直到遇上断点 | c | c |
| 单步执行 | s | s |
| 单步执行(遇到函数则跳过) | n | n |
| 查看寄存器信息 | info cpu  r  fp  sreg  creg | info cpu  r  fp  sreg  creg |
| 查看堆栈 | print-stack | print-stack |
| 查看内存物理地址内容 | xp /nuf addr | xp /40bx 0x9013e |

## 本章小结

本章详细介绍了内核的编译语言与环境。先给出了内核开发环境的简单介绍，比较了AT&T汇编会Intel汇编的区别，再介绍了GCC内嵌汇编知识，然后介绍内核编译环境makefile程序，以及内核调试工具bochs。其中还穿插了中断处理的相关知识，这些都是操作系统内核必要的知识储备。了解了这些，我们可以开始进行系统内核的设计和实现工作。

# 多任务操作系统内核设计

## 系统引导

当PC启动时，Intel系列的CPU首先进入的是实模式，CS寄存器的值为0xffff，IP寄存器的值是0x0000。因此CPU开始的时候是执行位于地址0Xffff0处的代码，也就是ROM-BIOS起始位置的代码。BIOS先进行一系列的系统自检，然后初始化位于地址0的中断向量表。最后BIOS将启动盘的第一个扇区(512字节)装入到绝对地址0x07C00，并跳转到这里开始执行此处的代码。这里说的启动盘既可以是硬盘也可以是软盘。这就是计算机启动过程的一个最简单的描述。由于本系统内核是基于导师所搭建的基础平台之上，系统引导部分是由导师所完成，并非本文所述重点，这里不做过多讲解，详情可参照《Linux内核完全注释》以及本人导师所搭建平台的文件djgpp/work/ bootsect /bootsect.s。

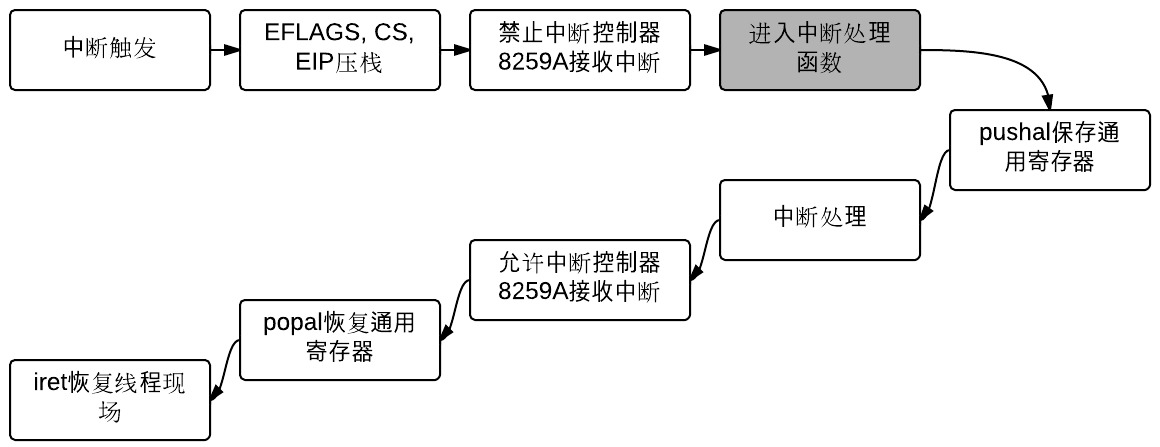
## 系统的初始化

系统初始化主要完成中断描述符表(Interrupt Descript Table，IDT)的创建和加载，定时器芯片(Programmable Interval Timer，PIT)8253与中断控制器芯片(Programmable Interrupt Controller，PIC)8259A的初始化设置，并设置全局描述符表(Global Descript Table，GDT)和局部描述符表(Local Descript Table，LDT)最后再跳转到main.c文件中的xmain()函数运行。这块内容也主要由导师完成，本人只在中断部分添加上自己的任务调度。下面只简单介绍与本多任务内核相关的中断初始化部分。详细可以参照文件djgpp/work/crt0.S。

中断的初始化部分，主要分两个阶段。

1. 中断描述符表的创建和加载。先预留足够的空间存放IDT表，然后将中断处理函数的地址对应Intel规定的中断(或异常)序列存放到IDT表中，最后把IDT的基址和限长作为LIDT的参数加载到IDTR中
2. 中断处理函数的编写。中断触发时，系统会先禁止中断控制器8259A接收其他中断，然后CPU会自动将标志寄存器EFLAGS，代码段寄存器CS，指令指针寄存器EIP的值压栈。为了保存中断现场，中断处理函数首先要将通用寄存器的值压栈，然后完成处理事宜，再允许中断控制器接收中断，并恢复现场。

中断处理的流程如图3.1：

图3.1 中断处理流程图

## 任务管理

程序是一个可执行的文件，进程(process)是一个执行中的程序实例，而线程(thread)作为更是作为CPU调度的基本单位，为了更方便资源共享和上下文切换，本多任务操作系统内核采用线程管理来实现任务调度。任务管理是多任务内核的核心部分，下面给出该模块的详细设计。

### 内核线程控制

在多任务操作系统内核中，系统最多可有64个线程同时存在。除了第一个线程是使用schedule\_init()初始化，其他全部是用全部是使用create\_thread()创建。被创建的线程称为子线程(child thread)，创建者为父线程(parent thread)。内核程序使用线程标识号(thread ID, tid)来标识每个线程。

对于只有一个CPU的系统中，在某一时刻只能有一个线程在运行，内核通过调度程序分时调度执行各个线程。利用分时技术，把CPU的运行时间分成一个个规定长度的时间片(time slice)，让每个线程在一个时间片内运行。当线程的时间片用完时系统就利用调度程序切换到另一个线程去运行，由于时间片很短，所以表面看起来好像所有的线程都在同时运行着。

### 线程的运行状态

一个线程在执行的过程中，总是在改变其状态。线程的状态保存在线程数据结构的state字段中。线程可以处于以下的状态之一：

新建状态(TASK\_NEW)。在create\_thread()的过程中，当线程已经申请到了任务结构和栈的内存，但是还未初始化任务结构，不能被调度程序调度，这个状态称为TASK\_NEW，新建状态。

运行状态(TASK\_RUNNING)。当线程正在被CPU执行，或者线程已经准备就绪等待CPU调度，则该线程处于运行状态。这两种状态在内核中表示相同（意味着调度程序可以调度该线程），都称为TASK\_RUNNING。

等待状态(TASK\_WAITING)。当线程因为需要某一资源才可以运行时，该线程加入到该资源的等待队列中，同时该线程处于等待状态，即TAKS\_WAITING，只有当内核显示地调用wake\_up()函数时，该线程才能再次转变为TASK\_RUNNING(就绪态)，被调度程序调度。

僵死状态(TASK\_ZOMBIE)。线程函数结束前不可以使用return返回，而是要调用thread\_exit()函数来退出，调用了thread\_exit()的线程，会处于TASK\_ZOMBIE的状态，这时候线程已经停止，但其父线程还未询问其状态。这个状态的线程已经不可能运行了，但是其数据结构和申请的栈仍然保留，待父线程查看后释放。

线程的状态转换如图3.2所示：

图3.2 线程转换状态图

调度

运行态

就绪态

中断

线程退出

等待资源

唤醒

初始化

当一个线程创建，线程申请了任务结构和当一个线程的运行时间片用完，中断程序则会调用调度程序切换到另一个线程(也可能会切换向本线程，这时不做任何操作)，当前线程从运行态(TASK\_RUNNING)转变为就绪态(TASK\_RUNNING)，而切换到的线程则从就绪态变为运行态。如果线程在运行态执行的时候需要等待系统的某个资源，该线程会调用sleep\_on()自愿放弃CPU的使用权，而让调度程序去执行其他线程，此时该线程变为TASK\_WAITING状态，并加入到该资源的等待队列。当系统资源可用，会调用wake\_up()唤醒等待队列，该队列的线程从TASK\_WAITING转变为就绪态等待CPU调度。执行完毕的线程会调用thread\_exit()来退出，线程状态变为TASK\_ZOMBIE。内核中运行的线程不能被其他线程抢占，而且一个线程无法改变另一个线程的状态。为了避免在线程切换的时候造成数据错误，内核在执行切换代码时会禁止一切中断。

### 任务数据结构

一个线程通常包括以下内容：一个指向当前被执行指令的指令指针，一个栈，一个寄存器值的集合，定义了一部分正在执行线程的处理器状态的值，一个私有的数据区。这些数据被保存在一个名为task\_struct的数据结构中，同时在task\_struct中还保存着线程的tid，状态，时间片数，优先级等信息。所有的task\_struct指针作为线程任务表task的一项保存在内存中，供调度程序进行管理和使用。详细参见文件sched.h。

当一个线程在执行时，CPU的所有寄存器中的值、线程的状态以及堆栈中的内容被称为线程上下文(context)。当内核从一个线程切换(switch to)到另一个线程中，需要保存当前线程的所有信息，即保存上下文，以便当内核再次被当切换到该线程的时候，能够恢复当前线程的状态继续执行下去。

任务数据结构设计如下：

|  |
| --- |
| struct task\_struct  {  struct context\_struct context;  long state;  long counter;  long priority;  long tid, father;  }; |

state为线程状态。

counter为时间片计数，起始值等于priority。

priority为线程的优先级，越大代表线程会更多的被调度。

tid为线程的线程号，其值也是该线程在任务数组中的索引。

father为父线程的线程号。

context为线程上下文结构。

|  |
| --- |
| struct context\_struct  {  long eflags;  long eip;  long eax, ecx, edx, ebx;  long esp, ebp, esi, edi;  long cs, ds, ss, es, fs, gs;  }; |

eflags保存标志寄存器的值，eip保存指令寄存器的值

eax，ecx…edi保存通用寄存器的值，他们的位置顺序遵从汇编指令pushal的压栈顺序。

cs，ds…gs代表段寄存器的值，在本操作系统内核中，该值在系统初始化后便固定了，不需改变，因此只是作为扩展，本内核并没有其保存和加载的过程。

任务数据结构这样设置后，在内嵌汇编中可以直接使用任务指针current加偏移量，便可得到所有寄存器值的位置，这点在任务切换中有用到。

### 任务管理接口设计

任务管理的接口包含了提供给内核调用的全局变量和接口函数。

全局变量如下所示：

|  |
| --- |
| extern BOOL reschedule;  extern struct task\_struct \*current;  extern struct task\_struct \*task[NR\_TASKS]; |

reschedule用来标记定时器中断是否要进行任务调度。

current是指向当前正在执行任务的指针。

task是存放所有任务指针的任务数组。

接口函数如下：

|  |
| --- |
| extern int schedule\_init();  extern int create\_thread(void \*(\*proc)(void \*));  extern void switch\_to(int n);  extern void schedule();  extern int sleep\_on(struct task\_struct\*\* p);  extern void wake\_up(struct task\_struct \*\*p);  extern void thread\_exit(); |

1. 调度器的初始化与线程的创建

调度器初始化函数schedule\_init()负责任务0的创建与初始化工作，任务0作为空闲任务，在无其他线程需要运行时被调度。schedule\_init()先申请任务数据结构和栈的内存，并把任务指针存储到任务数组的首项task[0]。然后设置线程的优先级，时间片，tid和父线程，初始化寄存器值，最后把线程状态设置为TASK\_RUNNING，允许调度程序调度。

线程创建函数create\_thread()与调度器初始化函数很类似，它负责除了任务0之外的所有线程的创建与初始化。与任务0直接存储到任务数组的首项不同，create\_thread()先在任务数组中找出一个还没被任何线程使用的空项。如果系统已经有64个线程在运行，则create\_thread()返回错误。为了防止未处理完成的新建线程被调度函数执行，create\_thread()在初始化数据之前还需将线程的状态设置为TASK\_NEW，直到线程初始化完成才改为TAKS\_RUNNING。

1. 线程调度

线程调度是任务管理的核心部分，包括调度程序schedule()，线程切换函数switch\_to()。

1. 调度程序

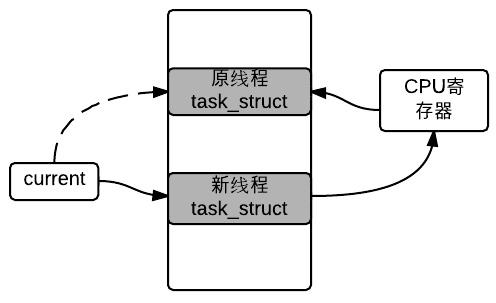
为了能让线程有效的使用系统资源，又能使线程有较快的响应时间，就需要对线程的切换调度采用一定的策略。本系统采用了基于优先级排队的调度策略。

线程调度函数schedule()根据调度算法，从任务数组task中找出要跳转的目标任务，然后使用switch\_to()跳转到该任务。如果没有可调度的任务，则跳转到任务0空闲运行。

1. 线程切换

执行实际的线程切换任务由switch\_to()函数完成。在进行切换之前switch\_to()首先检查要切换到的线程是否为当前线程，如果是则什么都不做，直接退出，否则就先关闭中断，将当前CPU寄存器的值保存到全局变量current指针指向的线程任务结构中，然后再将current设置为新任务的指针，最后把新任务的寄存器值加载到CPU中，实现跳转。

线程切换如图3.3：

图3.3 线程切换示意图

1. 线程调度的时机

在多任务系统内核中，调度程序schedule()会在两种情况下被调用。

1. 线程主动调用

当线程调用sleep\_on()等待资源，进入休眠状态，或者调用thread\_exit()退出运行进入僵死状态。线程都会主动放弃CPU的使用权，并调用schedule()，让调度程序重新选择线程执行。这时候调度函数保存的是线程的现场，当该线程再次运行的时候会恢复该现场，继续运行。

1. 中断调用

所谓中断，即定时器中断。中断调用，则指系统每隔几次定时器中断(一个时间片)，调用一次调度函数。这是操作系统任务调度的主要方式，频率很高。因为调度函数会直接切换到目标任务运行，为了可以打断它，不能在中断处理函数中调用，而需要在中断处理函数结束，打开8259A中断芯片后调用。这时候，调度函数保存的现场是中断现场，而线程的现场则被中断调用保存，当该任务再次被调度运行的时候，调度函数恢复中断现场，中断再返回到线程函数中继续运行。

1. 线程的休眠与唤醒

线程休眠函数sleep\_on()的主要功能是当一个线程所请求的资源不可用的时候，将该线程置于等待队列，并切换到其他函数运行。该线程得到请求的资源则切换回来继续运行。当一个线程调用sleep\_on()的时候，系统把等待队列的头指针设置为当前线程，通过局部变量来引用原队列头，然后改变当前线程状态设置为TASK\_WAITING，并调用schedule()函数，切换到另一个线程运行。当请求的资源可用，系统调用wake\_up()函数，把队列头指针指向的线程状态设置为就绪态，即把最后调用sleep\_on()的线程状态设置为TASK\_RUNNING，允许该线程被调度。当该线程被调度，继续运行sleep\_on()的后续代码——修改局部变量引用的线程(即原队列头指针指向的线程)状态为就绪态，这样递归下去，使得所有处于等待状态的线程均被唤醒。

1. 线程退出

当一个线程结束运行，那个内核则需要释放该线程所占用的系统资源。主要包括该线程申请的内存等。

当一个线程调用thread\_exit()，则会将该线程的状态设置为TASK\_ZMOBIE,然后调用schedule()，让调度程序重新选取一个线程开始执行。最后，父线程要检测处于僵死状态的子线程，查看其状态并可以手动释放其资源。

## 本章小结

本章主要介绍多任务系统的内核设计，包括系统的引导，初始化，任务管理，内存管理等模块。本章的核心是任务管理模块，给出了内核最为重要的数据结构以及接口的设计，并对接口所完成的功能加以阐释。

然而，正如Linux系统的创始人Linus所说，要理解一个系统的真正的运行机制，一定要阅读其源代码。下一章将介绍本系统的代码实现，并对关键代码进行分析。

# 多任务操作系统内核实现

## 程序目录

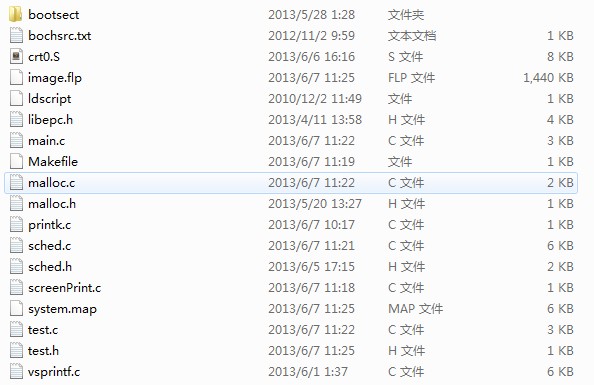
多任务操作系统内核主要由引导文件bootsect.S，初始化文件crt0.S，任务管理文件sched.c，内存管理文件malloc.c，以及屏幕打印文件printk.c、vsprintf.c与screenPrint.c组成。图4.1为程序的目录结构：

图4-1 项目目录结构

其中bootsect.S存储在文件夹bootsect中，bochsrc.txt对bochs以及其加载镜像进行设置。image.flp为该系统内核生成的软盘镜像，Makefile是内核代码的编译文件，system.map则是编译内核时生成的内核内存存贮位置信息文件，该文件主要用在内核的调试过程中。

## 任务管理模块的实现

### 任务管理接口实现

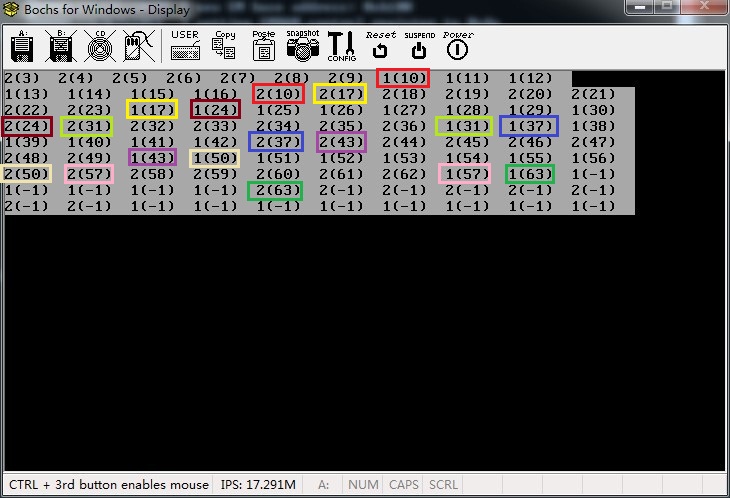
1. 调度器初始化与线程创建函数的实现

线程创建函数create\_thread()的实现比较简单，只有一点需要注意，线程创建函数需要调用find\_empty\_thread()来扫描任务数组找出空位，由于线程创建函数可能在多线程中被多个线程同时调用，而扫描任务数组的操作处于临界区内，因此需要使用CLI和STI，开关中断将这块代码保护起来。

下面是线程创建函数的伪代码：

|  |
| --- |
| CLI; //关中断  int pos =find\_empty\_thread(); //扫描任务数组查找空项  struct task\_struct \*tcb = malloc(); //为任务数据结构申请内存  char\* stack = malloc(); //为线程栈申请内存  task[pos] = tcb; //将任务指针放入任务数组  STI; //开中断  … //初始化任务等 |

如果不使用CLI和STI保护临界区则可能出现多个线程，找到任务数组的同一位置作为其任务指针的存放处。图4.2说明了这一情况：

图4.2 未保护临界区的函数创建图

上图的运行结果是，两个线程不间断的调用create\_thread()时出现的情况。

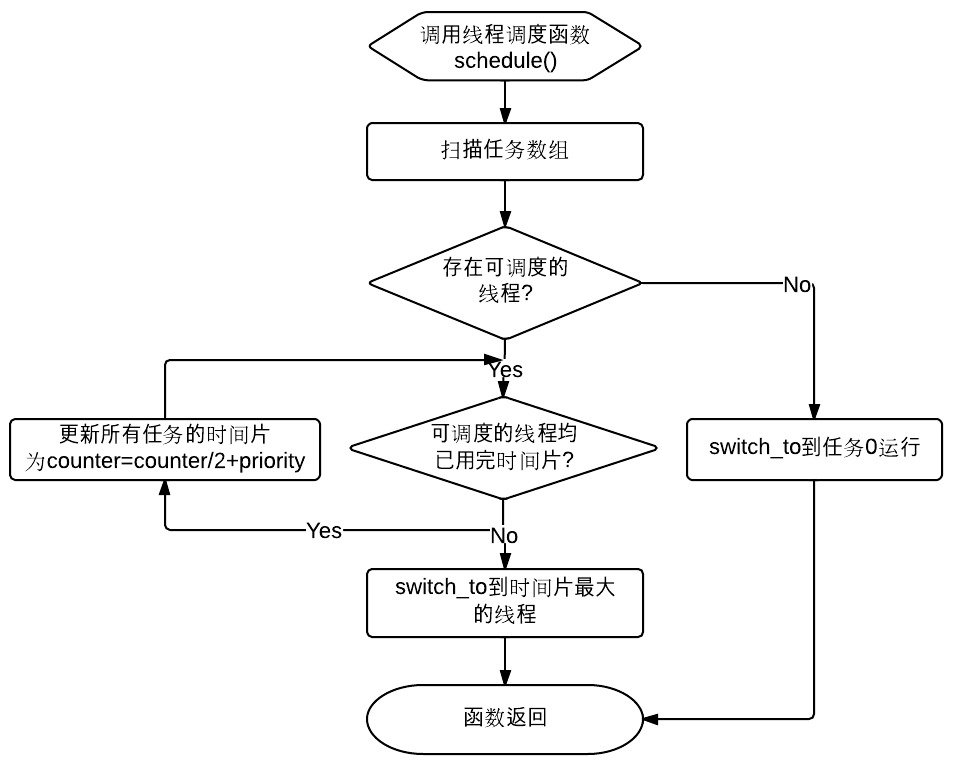
括号外的数字代表线程号，括号内的数字代表该线程使用create\_thread()申请到的任务数组中的位置。这里为了是现象更明显，在create\_thread()的find\_empty\_thread()使用了sleep()进行延时。我们发现，当线程2刚执行完find\_empty\_thread()找到位置10时，还未将找到的位置“占领”，便切换到线程1，线程1扫描任务数组，也找到了位置10，所以最后出现的结果是两个线程创建出的线程都把10作为其存放任务指针的位置，发生了错误。其他诸如位置17，24等都出现了该问题。这点可以使用开关中断保护来解决，解决的结果见该函数的测试部分。

调度器初始化较为简单，因为它只会被调用一次。不会出现线程创建函数中的问题。这里不做过多介绍，具体可按照代码。

1. 调度函数的实现

调度函数schedule()函数首先扫描任务数组。如果此时不存在可调度(处于TASK\_RUNNING)的任务，则直接切换到任务0运行。如果存在可调度的线程，该函数则检查这些线程的时间片数，若都为0，则系统根据优先级对所有的线程(包括处于休眠状态的线程)重新计算时间片；若存在线程时间片不为0，该函数则找出时间片值最大的线程，然后调用switch\_to()切换到目标线程。

线程调度的流程如图4.3：

图4.3 线程调度流程图

1. 任务切换函数的实现

任务切换函数switch\_to(int n)的作用是把当前正在执行的函数切换到任务数组的第n项，即task[n]。该线束是线程调度的核心函数 ，主要负责把当前线程的上下文保存到current指向的任务结构，并把task[n]中的上下文加载到当前CPU的寄存器中，来实现跳转到目标任务的目的。

跳转工作分为四步：

1. 该函数先判断要跳转的任务是否是当前任务，若是则什么都不做函数返 回，若不是则执行调转操作。
2. 将当前线程的上下文保存到current指向的任务结构中。

|  |
| --- |
| \_\_asm\_\_("cli \n\t"  "pushal \n\t"  "movl %0, %%eax \n\t"  "popl 36(%%eax)\n\t "  "popl 32(%%eax)\n\t "  "popl 28(%%eax)\n\t "  "popl 24(%%eax)\n\t "  "popl 20(%%eax)\n\t "  "popl 16(%%eax)\n\t "  "popl 12(%%eax)\n\t "  "popl 8(%%eax)\n\t "  "pushfl \n\t"  "popl (%%eax)\n\t"  "leal 1f, %%ecx\n\t"  "movl %%ecx, 4(%%eax)"  ::"m"(current)  ); |

这段代码使用C内嵌汇编，先把当前线程的数据结构地址current作为输入列表传入，并使用cli关闭中断(任务切换的时候禁止中断)，然后立刻使用pushal将所有通用寄存器压栈，保证栈中是未被修改的寄存器值。由于内嵌汇编不允许“36(%0)”这样的内存引用，所以先把current的值传入寄存器eax(通用寄存器已经保存在栈中，因此可以使用)，再使用popl命令将栈中的通用寄存器值弹出到当前线程的数据结构中。pushal的压栈顺序是，eax, ecx, edx, edx, 原始esp, ebp, esi, edi。所弹栈的顺序正好相反，从地址current+36的edi开始(类型long的变量占4个字节，edi前有9个变量)弹出，到current+8处的eax。上述流程除了cli会将标志寄存器的中断允许标志位IF复位(这并不影响线程的运作)，其他操作不会改变标志寄存器的值，所以使用pushfl指令将标志寄存器的值压栈，并将值弹到eflags对应的地址处，便将标志寄存器保存。最后leal取“标号1”的逻辑地址存入寄存器ecx中，然后把它保存到eip中。

“标号1”位于switch\_to()函数的末尾。于是，当该线程跳转完毕，再次被调度时，CPU通用和标志寄存器中加载的值是switch\_to函数刚进入时的值，而eip的值则为函数的末尾，当该线程开始运行，switch\_to()立刻返回。线程函数便恢复到没调用switch\_to()的状态继续向下运行，如此说来，线程现场的保存成功。

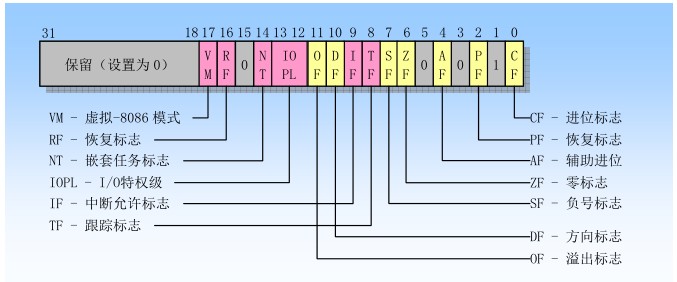
1. 将要跳转的线程task[n]指针赋值给current。
2. 将线程task[n]的上下文加载到CPU中

|  |
| --- |
| \_\_asm\_\_("movl %0, %%eax \n\t"  "movl 24(%%eax), %%esp \n\t"  "movl (%%eax), %%ecx \n\t"  "or $0x02, %%ch\n\t"  "pushl %%ecx \n\t"  "pushl %%cs \n\t"  "pushl 4(%%eax) \n\t"  "pushl 8(%%eax) \n\t"  "pushl 12(%%eax) \n\t"  "pushl 16(%%eax) \n\t"  "pushl 20(%%eax) \n\t"  "pushl 24(%%eax) \n\t"  "pushl 28(%%eax) \n\t"  "pushl 32(%%eax) \n\t"  "pushl 36(%%eax) \n\t"  "popal \n\t"  "iret \n\t"  "1:"  ::"m"(current)  ); |

与上下文保存的代码类似，加载代码先把current(此时已经被赋值为task[n])作为输入传入，由于对内存间接寻址需要用到寄存器eax，所以我们这里采取类似中断调用的方式，先把eflags，cs，eip压栈，然后把通用寄存器压栈，最后使用popal把栈中的通用寄存器值弹入CPU，再用iret，把eip，cs，eflags的值分别赋值给指令寄存器，代码段寄存器和标志寄存器。

在这里使用到了两个通用寄存器，eax用来保存任务结构的地址，ecx用来保存eflags的值。这里先把把寄存器ch(ecx的8-15位)与0x02取或，这样相当于把中断允许标志位IF设为1，然后把ecx作为eflags的值压栈。这样当调用iret时，切换到目标任务的时候中断是打开的，这样定时器中断才能每个一个时间片便中断线程，进行调度。

图4.4是标志寄存器中各标识位代表的意义：

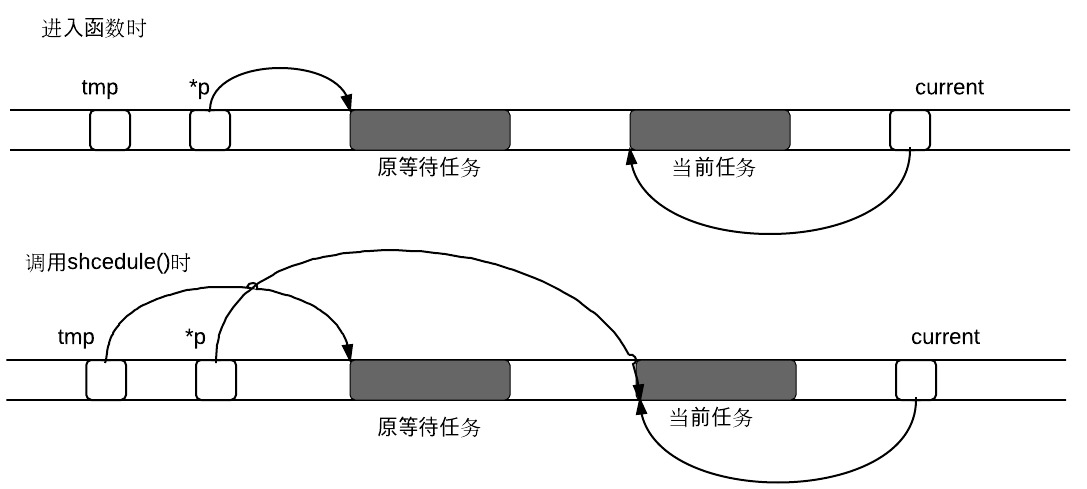
图4.4 标志寄存器EFLAGS中的系统标志

1. 线程休眠与唤醒函数的实现

线程休眠函数sleep\_on()将当前线程放入休眠队列并重新进行任务调度。代码如下：

|  |
| --- |
| int sleep\_on(struct task\_struct\*\* p)  {  struct task\_struct \*tmp;  if(!p)  return -1;  if(current == FIRST\_TASK)  return -2;  tmp = \*p;  \*p = current;  current->state = TASK\_WAITING;  schedule();  if(tmp)  tmp->state = TASK\_RUNNING;  return 0;  } |

当刚进入该函数的时候，队列头指针\*p指向已经在队列中等待的任务结构。当然，在系统刚开始运行的时候，等待队列上无等待任务，这时候\*p为NULL。通过指针操作，在调度程序schedule()执行之前，tmp指向了原等待任务，队列头指针\*p指向了当前任务。如图4.5所示：

图4.5 sleep\_on()函数中指针变化示意图

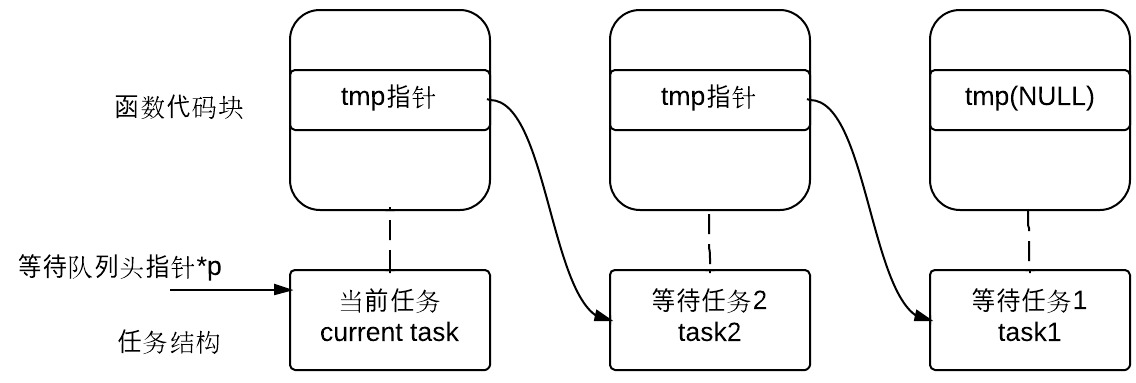
通过临时指针tmp，在几个线程为等待同一资源而多次调用该函数时，程序就隐式地构筑处一个等待队列。从图4.6中，我们可以更容易的理解sleep\_on()函数的等待队列形成的过程。图中给出了当向队列头部插入第三个任务时的情况。

图4.6 sleep\_on函数的隐式任务等待队列

在插入等待队列后，sleep\_on()函数就会调用schedule()函数去执行别的线程。当线程被唤醒而重新执行时就会执行后续的语句，把比它早进入等待队列的一个线程唤醒。

唤醒操作函数wake\_up()把正在等待可用资源的指定任务置位就绪状态，当唤醒的线程被调度，处于等待队列的前一个任务便被置位就绪状态。所以wake\_up()的作用实际上是唤醒指定任务以及在该任务之前进入等待队列的任务。但在有些情况下，唤醒的并非队列的头指针，也可能是中间的某个任务，因此还需要把被唤醒的任务结构的指针质空。这样在其后进入睡眠的线程被唤醒时，就不会再次修改该任务的状态。wake\_up()函数的代码如下：

|  |
| --- |
| void wake\_up(struct task\_struct \*\*p)  {  if (p && \*p) {  (\*\*p).state = TASK\_RUNNING;  \*p = NULL;  }  } |

### 定时器线程调度实现

关于线程调度的时机，在上一章我们知道，线程的调度主要位于定时器中断中。定时器中断每隔一个时间片调度一次schedule()，进行任务调度和切换。其实现代码如下：

|  |
| --- |
| #define hwint\_master(irq) \  pushal; \  pushl $irq; \  movl (\_int\_vector + 4 \* irq), %eax; \  call \*%eax; \  popl %eax; \  movb $0x20, %al; \  outb %al, $0x20; \  cmpl $0, \_reschedule; \  je 1f; \  call \_schedule; \  1:; \  popal; \  iret |

当定时器中断发生，标识寄存器，代码段寄存器，指令指针寄存器压栈，然后hwint\_master(0)被调用，pushal把所有的通用寄存器压栈，此时线程上下文已被保存。接下来的工作便是将irq(中断号)压栈作为中断处理函数的参数，中断处理函数的地址也是用irq来寻址，call跳转到中断处理函数。当中断处理函数返回，则允许8259A芯片处理中断，然后根据变量reschedule来判断是否进行任务调度。之后则弹出通用寄存器以及使用iret恢复线程现场。继续原来的线程运行。

## 本章小结

本章介绍了多任务操作系统内核的代码实现，主要针对任务管理模块的实现进行分析。其中包括线程调度接口以及定时器线程调度的实现。详细的讲解了线程的创建函数create\_thread()，跳转函数switch\_to()，以及休眠函数sleep\_on()和唤醒函数wake\_up()，并对函数内较为复杂的代码采用图示的方式进行解释。

# 多任务操作系统内核测试

对于一个操作系统来说，安全性、稳定性至关重要。为了确保和证明本多任务操作系统内核的正确运行。本章给出系统总体功能以及关键函数的测试用例，以及测试结果。

## 测试计划

### 测试目标

1. 测试系统总体是否正确运行，是否能正确完成任务调度。
2. 测试关键函数的运行。
3. 测试关键函数的内部逻辑。

### 测试的函数

1. create\_thread();
2. switch\_to();
3. schedule();
4. sleep\_on()与wake\_up();

其他函数由于比较简单，不作测试。

## 测试用例

### 系统功能测试

表5.1 系统功能测试用例1

|  |  |
| --- | --- |
| 测试用例 | 系统功能测试用例1（黑盒） |
| 用例描述 | 总体功能测试，使用create\_thread创建两个线程分别输出1和2，各输出100次后线程退出。 |
| 输入数据 | 两个线程  Thread1:输出1  Thread2:输出2 |
| 预测结果 | 屏幕交替输出1和2，输出的次数大致相同，当输出够100个1和2的时候，开始输出0(0为任务0的输出) |

得测试结果如图5.1所示：

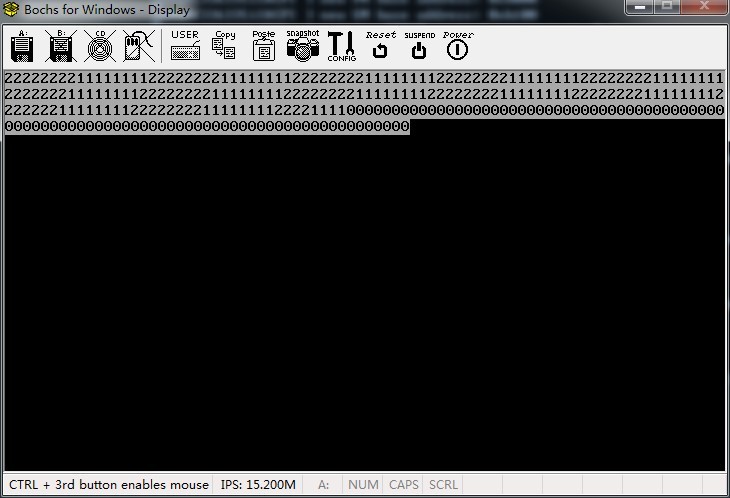
图5.1 系统功能测试用例1结果图

表5.2 系统功能测试用例2

|  |  |
| --- | --- |
| 测试用例 | 系统功能测试用例2(黑盒) |
| 用例描述 | 功能测试(线程创建create\_thread()) |
| 输入数据 | 使用create\_thread()创建一个线程，输出“T” |
| 预测结果 | 屏幕先输出100个T，开始输出0(0为任务0的输出) |

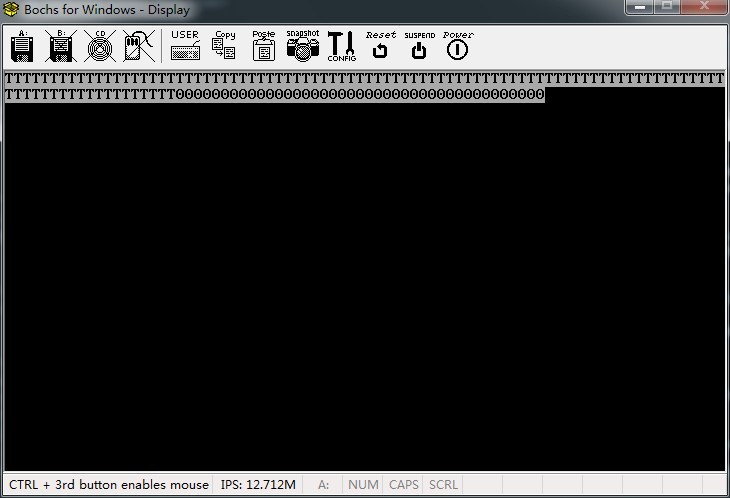
测试结果图5.2所示：

图5.2 系统功能用例2结果图

表5.3系统功能测试用例3

|  |  |
| --- | --- |
| 测试用例 | 系统功能测试用例3(黑盒) |
| 用例描述 | 功能测试(线程切换switch\_to()) |
| 输入数据 | 关闭定时器的任务调度，创建两个函数thread1(输出1)和thread2(输出2)，使用switch\_to()切换到thread2，运行一段代码然后切换到任务1，最后再切换到任务2。 |
| 预测结果 | 屏幕先输出2，然后输出1，再输出2。 |

测试结果如图5.3：

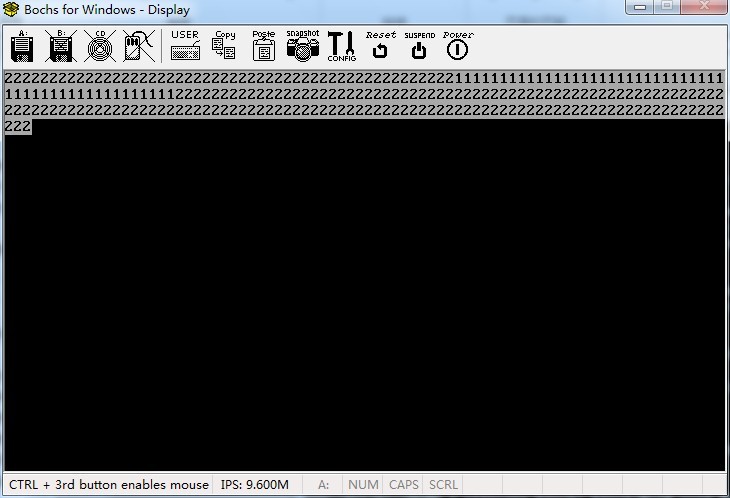
图5.3 系统功能测试用例3结果图

表5.4系统功能测试用例4

|  |  |
| --- | --- |
| 测试用例 | 系统功能测试用例4(黑盒) |
| 用例描述 | 功能测试(线程调度schedule()) |
| 输入数据 | 关闭定时器的任务调度，创建两个函数thread1(输出1)和thread2(输出2)，手动调用schedule()，并在thread1和thread2中分别调用schedule() |
| 预测结果 | 屏幕先输出2，然后输出1，再输出2。 |

测试结果图5.4：

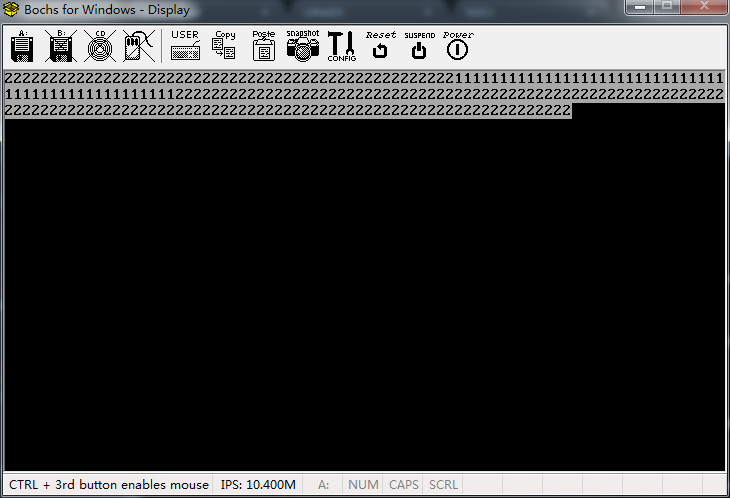
图5.4 系统功能测试用例4输出

表5.5 系统功能测试用例5

|  |  |
| --- | --- |
| 测试用例 | 系统功能测试用例4(黑盒) |
| 用例描述 | 功能测试(线程休眠sleep\_on()与唤起wake\_up()) |
| 输入数据 | 创建两个函数thread1(输出1)和thread2(输出2)，并在定时器中断中使用schedule()进行调度。在thread2中先使用sleep\_on()加入等待队列，在thread1中输出50次后，唤醒该队列。 |
| 预测结果 | 屏幕先输出“1” 50次，然后输出1大约相同的次数(根据调度算法)最后输出1，2交替输出 |

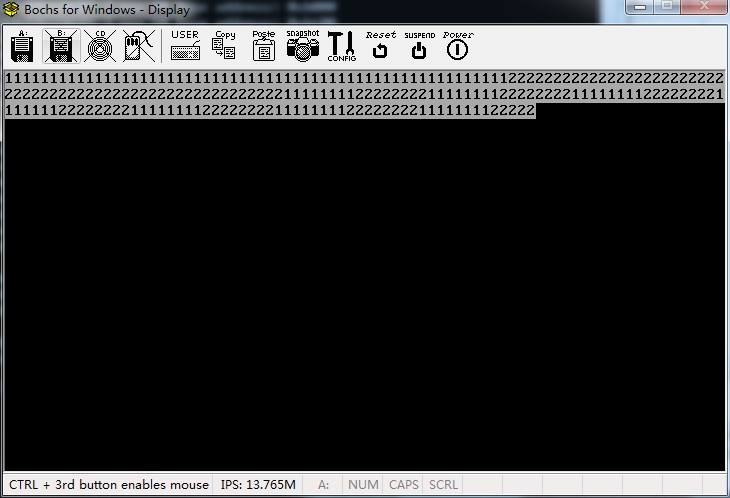
测试结果如图5.5所示：

图5.5 系统功能测试用例5结果图

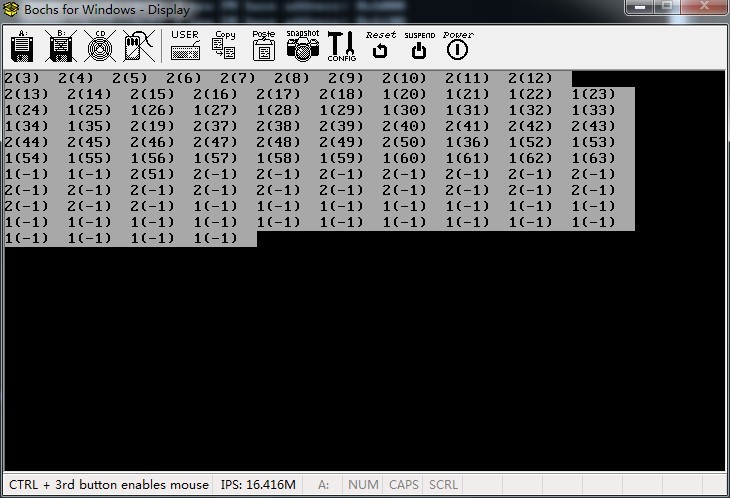
### 函数结构测试

1. 函数create\_thread()分支测试

表5.6 create\_thread()分支测试用例

|  |  |
| --- | --- |
| 测试用例 | create\_thread()分支测试(白盒) |
| 用例描述 | 分支测试针对create\_thread()的临界区保护进行测试，先在find\_empty\_thread()后用sleep()进行延时，然后创建两个线程不间断的调用create\_thread()创建新线程， |
| 输入数据 | 两个线程，各自分别创建线程并显示号 |
| 预测结果 | 线程号从3-63输出，但不一定会按顺序，但绝对不会重复。 |

测试结果如图5.6所示：

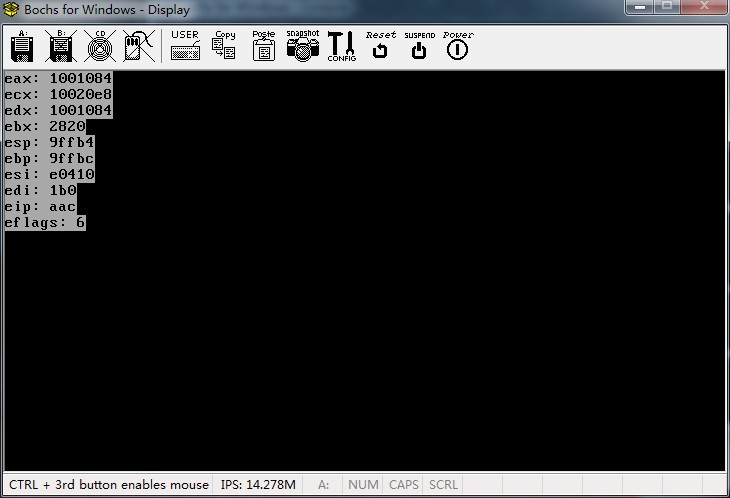
图5.6 createThread分支测试结果图

1. 函数switch\_to()分支测试

表5.7 switch\_to()分值测试用例1

|  |  |
| --- | --- |
| 测试用例 | switch\_to()分支测试用例1(白盒) |
| 用例描述 | 分支测试用例1针对上下文保存部分，把刚进入函数时CPU寄存器的值(从bochs调试中获得)与现场保存后任务结构中的值对比来验证保存成功与否 |
| 输入数据 | 线程执行任务跳转 |
| 预测结果 | CPU寄存器的值与任务结构中的相等 |

任务结构中的值如图5.8所示：

图5.8 switch\_to()上下文保存分支测试任务结构寄存值

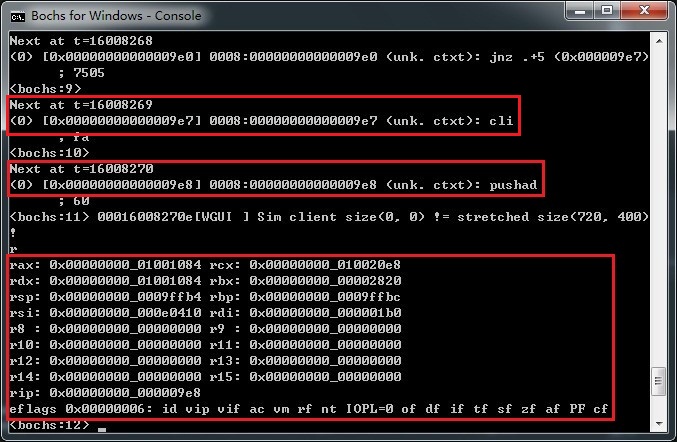
使用bochs调试得到cli后的寄存器值如图5.9所示：

图5.9 switch\_to()上下文保存分支测试bochs调试图1

eip的值不相等，因为eip保存的是“标号1”的地址。在"leal 1f, %%ecx\n\t"后ecx保存的便是正确的地址，如图5.10所示：

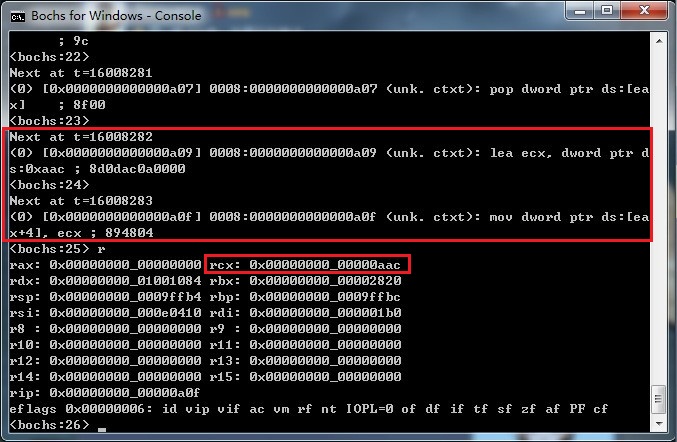
图5.10 switch\_to()上下文保存分支测试bochs调试图2

表5.8 switch\_to()分值测试用例2

|  |  |
| --- | --- |
| 测试用例 | switch\_to()分支测试用例2(白盒) |
| 用例描述 | 分支测试用例2针对上下文加载部分，把跳转刚完成时CPU寄存器的值(从bochs调试中获得)与加载的任务结构中的值对比来验证加载成功与否 |
| 输入数据 | 线程执行任务跳转 |
| 预测结果 | CPU寄存器的值与任务结构中的相等 |

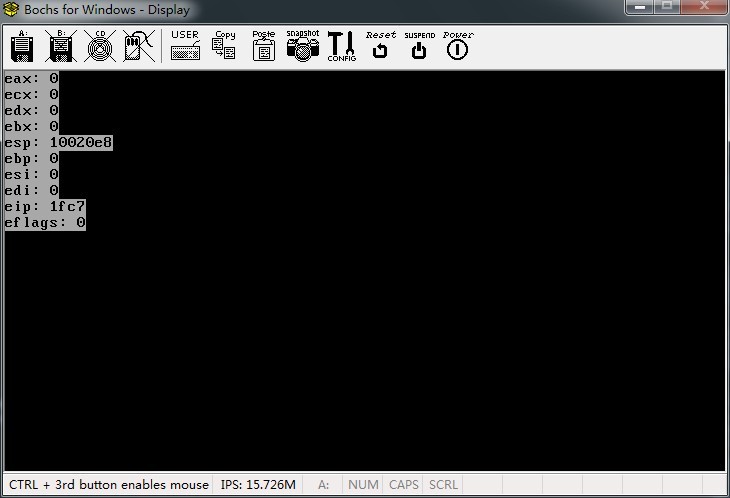
任务结构中寄存器值如图5.11所示：

图5.11 switch\_to()上下文加载分支测试任务结构寄存器值

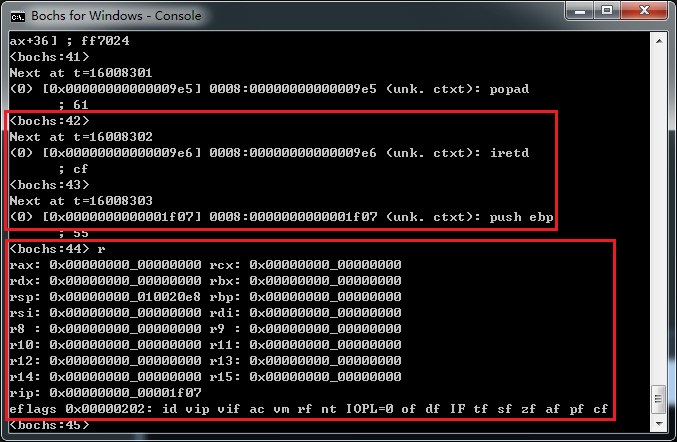
Iret后bochs调试得到寄存器值如图5.12所示：

图5.12 switch\_to()上下文加载分支测试bochs调试图

比照两图可以看出除了标志寄存器eflags的值不同，其他都相同。而eflags变为0x202的原因在于，打开中断(0x200)以及标志寄存器的第1位(从0数)固定为1(见图4.3)。因此可知寄存器加载正确。

## 综合测试

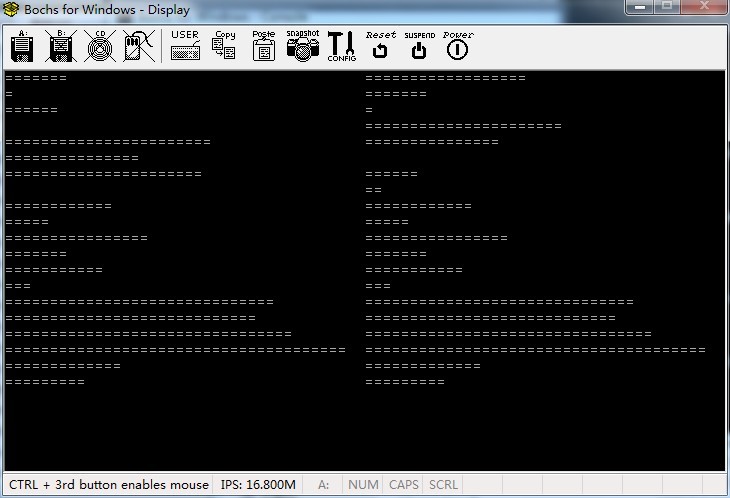
综合测试给出一个较为复杂的测试案例：创建两个线程，分别对一个数组进行快速排序和冒牌排序。为了测试结果的直观性，这里把屏幕分为两个部分来演示排序流程。图5.13为排序过程中的截图：

图5.13 综合测试排序过程图

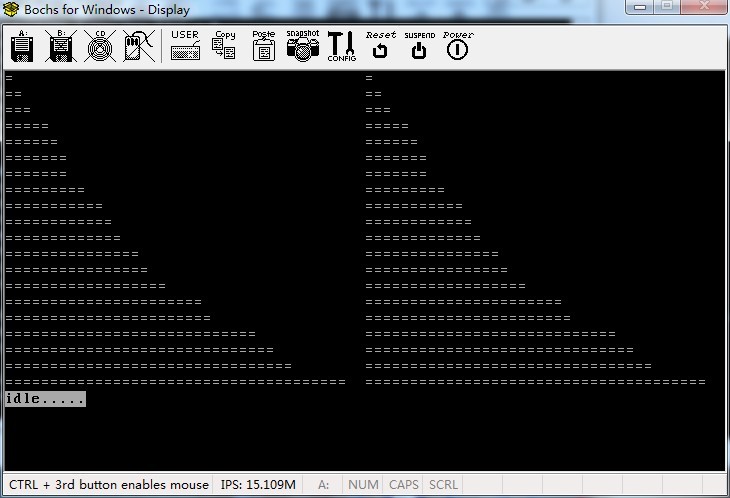
图5.14为排序结束的截图：

图5.14 综合测试结果图

## 本章小结

本章使用了黑盒和白盒技术，对系统的总体功能，以及各个函数功能性进行了黑盒测试，并针对系统关键函数create\_thread()与switch\_to()的内部结构进行了白盒测试。最后更是给出一个较为复杂的测试案例，直观的展示了系统的运行成果。一切测试结果证明，本操作系统内核运行良好，具有较强的安全性和健壮性。

# 总结与展望

## 总结

本次项目主要针对国内操作系统发展缓慢，操作系统研究困难，理论脱离实践的现状，致力于打造一个让广大计算机爱好者研究和实践的平台。主要完成了操作系统中最为关键的任务管理模块。首先系统的底层引导和初始化进行介绍，然后完成了任务管理模块的设计，再通过编码将功能实现。为了项目的良好展示，又在项目实现模块添加了一个较为复杂展示模块。对于操作系统来说，系统的稳定性和安全性至关重要，为了验证本系统的健壮，在项目的最后对项目的功能和关键函数的内部逻辑进行了测试，并通过测试结果证明系统运行良好。

本文所完成的工作归纳如下：

1. 对内核开发环境，以及关键技术进行介绍。
2. 系统设计，对系统底层环境进行介绍，对系统核心模块任务管理进行详细设计。
3. 系统实现，编码实现系统功能，对核心代码进行解释。
4. 系统测试，对系统总体功能以及函数的功能性进行黑盒测试，对关键函数及其核心代码的逻辑性进行白盒测试，并通过一个综合测试的案例直观展示测试结果，验证系统的安全性和健壮性。

## 展望

本系统已基本完成多任务操作系统内核的核心工作——任务管理，但是还有一些诸如系统调用，线程通信等工作亟待完成。同时对于一个完整操作系统来说，内存管理，文件管理，设备管理等都是必须的，因此多任务内核的后续开发，还有很多药做。另一方面，由于时间比较紧迫，代码结构并没有经过严密的组织，函数接口也有待完善。

# 致谢

在毕业论文定稿之际，回顾这几个月的所有工作及努力，在这段时间遇到过许多的困难，在此对所有给予我帮助和关心的良师益友致以最诚挚的谢意！

在此，我要特别感谢我的指导教师——洪明坚老师。在操作系统内核的开发，入门很难！尤其是底层的知识，很多都难以捉摸，在近半年的毕业设计中，洪老师系统设计和实现过程中给我提供了大量的指导，每当遇到困难的时候，洪老师的意见给了我很大的启发和帮助。

其次，我要感谢和我一起并肩走过这半年进行毕业设计的各位同学，我们互相帮助，互相激励，对每一个问题互相讨论，这些点点滴滴都使我受益匪浅，也是你们让我度过了美好的大学四年。

最后，衷心感谢在百忙之中抽出宝贵的时间对本论文进行评审的各位专家！

# 参考文献

[1]Manual P P F D. volume 2, programmer’s reference manual[J]. Intel Corporation, 1996: 242691-001.

[2] Corportation I. IA-32 Intel Architecture software developer’s manual[J]. Intel Corportation, 2001.

[3] Ritchie D M, Kernighan B W, Lesk M E. The C programming language[M]. Bell Laboratories, 1975.

[4] Stallman R M. Using and porting the gnu compiler collection[J]. Free Software Foundation, 1989, 59: 02111-1307.

[5] Rusling D A. The linux kernel[J]. 1999.

[6] Coleman C L. Using Inline Assembly with gcc[J]. 1988.

[7] Crawford J H, Gelsinger P P. Programming the 80386 (1987)[J]. SYBEX, San Francisco, USA.

[8] Tanenbaum A S, Woodhull A S, Tanenbaum A S, et al. Operating systems: design and implementation[M]. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[9] 赵炯. Linux 内核完全注释[M]. 机械工业出版社, 2004.

[10] Bach M J, 陈葆珏. UNIX 操作系统设计[J]. 2000.

[11] Bovet D P, Cesati M, 陈莉君. 深入理解 Linux 内核[M]. 中国电力出版社, 2001.

[12] Stevens W R. UNIX 环境高级编程: 英文版[M]. 机械工业出版社, 2002.

[13] Plauger P J. The standard C library[M]. Prentice Hall PTR, 1991.

[14]布赖恩特, Bryant R E, 奥哈洛伦O', 等. 深入理解计算机系统[M]. 中国电力出版社, 2004.

[15] Culley P R, McGraw M, Walker K N, et al. Programmable interrupt controller: U.S. Patent 5,101,497[P]. 1992-3-31.

[16] Tanenbaum A S, Woodhull A S, 陈渝, 等. 操作系统设计与实现: 第三版[M]. 电子工业出版社, 2007.

[17] Elsner D, Fenlason J. Using as: the GNU Assembler[J]. 2000.