组号：

****

操作系统课程设计报告

题 目： GeekOs操作系统的研究与实现

院（系）： 计算机与信息安全学院

专 业： 计算机科学与技术

组 长：

组 员：

指导老师：

职 称：

2021年5月10日

小组分组安排

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 团队成员 | 姓名 | 性别 | 学号 | 分工 | 小组评分 | 工作量（%） |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

报告评分标准及得分

|  |  |  |
| --- | --- | --- |
| 序号 | 评分标准 | 得分 |
| 1 | 报告规范性（15分） |  |
| 2 | 论文工作量（20分） |  |
| 3 | 设计方案及应用能力（40分） |  |
| 3 | 团队组织形式（10分） |  |
| 4 | 实验结果及分析（15分） |  |
| 总分 | |  |

摘 要

GeekOS是一个基于X86架构的PC上运行的微操作系统内核, 系统内核设计简单,却又兼备实用性,它可以运行在真正的 X86 PC 硬件平台.可作为一个课程设计平台。本课程设计是基于GeekOS的project0-project6七个源工程项目进行开发完善，建立与操作系统理论基础知识的联系。

GeekOS提供了操作系统与硬件之间的所有必备接口,实现了系统引导,实模式到保护模式的转换,中断调用及异常处理,基于段式的内存管理,FIFO 进程调度算法以及内核进程,基本的输入输出(键盘作为输入设备,显示器作为输出设备),以及一个用于存放用户程序的只读文件系统 PFAT。在此基础上，我们可以通过project0掌握理解操作系统基础I / O操作, project1和project2掌握用户态和内核态的进程相关生命周期，project3实现多级反馈队列(MLF)和轮询(RR)调度算法以及信号量的相关操作,通过project4实现分段分页内存管理方式，最后project5基于虚拟文件系统实现GOSFS文件系统，project6为文件系统增加访问控制列表，并使用匿名半双工管道实现进程间通信。

本课程设计实现了project0-project6的基本需求，project0-project4实现了包括IO输入输出、解析ELF文件，建立运行用户态进程，调度算法的切换和信号量操作，分页虚拟存储内存等功能。此外，本课程设计还实现了project5的GOSFS文件系统的部分常用文件操作指令功能(mkdir、touch、rm、cp、cat等)，以及project6的管道操作和进程通信功能。

关键词： GeekOS、调度算法、内存管理、进程通信、文件系统GOSFS

目 录

[第一章 引言 1](#_Toc71971001)

[1.1 课程设计开发背景 1](#_Toc71971002)

[1.1.1 GeeKOS课程设计任务概述 1](#_Toc71971003)

[1.1.2 GeeKOS实验环境 1](#_Toc71971004)

[1.2 课程设计具体开发步骤 2](#_Toc71971005)

[第二章 GeekOS设计项目0—GeekOS系统环境调试及编译 3](#_Toc71971006)

[2.1 Project0项目原理分析 3](#_Toc71971007)

[2.1.1 项目0设计目的 3](#_Toc71971008)

[2.2.2 项目0设计原理及分析过程 3](#_Toc71971009)

[2.2 Project0项目运行分析 4](#_Toc71971010)

[第三章 GeekOS设计项目1—内核级线程设计及实现 6](#_Toc71971011)

[3.1 Project1项目原理分析 6](#_Toc71971012)

[3.1.1 项目1设计目的 6](#_Toc71971013)

[3.1.2 项目1设计原理及分析过程 6](#_Toc71971014)

[3.2 Project1项目运行分析 8](#_Toc71971015)

[第四章 GeekOS设计项目2—用户级进程的动态创建与执行 10](#_Toc71971016)

[4.1 Project2项目原理分析 10](#_Toc71971017)

[4.1.1 项目2设计目的 10](#_Toc71971018)

[4.1.2 项目2设计原理及分析过程 10](#_Toc71971019)

[4.2 Project2项目运行分析 13](#_Toc71971020)

[第五章 GeekOS设计项目3—进程调度算法与信号量功能 14](#_Toc71971021)

[5.1 Project3项目原理分析 14](#_Toc71971022)

[5.1.1 项目3设计目的 14](#_Toc71971023)

[5.1.2 项目3设计原理及分析过程 14](#_Toc71971024)

[5.2 Project3项目运行分析 17](#_Toc71971025)

[第六章 GeekOS设计项目4—分页存储管理机制 19](#_Toc71971026)

[6.1 Project4项目原理分析 19](#_Toc71971027)

[6.1.1 项目4设计目的 19](#_Toc71971028)

[6.1.2 项目4设计原理及分析过程 19](#_Toc71971029)

[6.2 Project4项目运行分析 22](#_Toc71971030)

[第七章 GeekOS设计项目5—GOSFS文件系统 24](#_Toc71971031)

[7.1 Project5项目原理分析 24](#_Toc71971032)

[7.1.1 项目5设计目的 24](#_Toc71971033)

[7.1.2 项目5设计原理及分析过程 24](#_Toc71971034)

[7.2 Project5项目运行分析 27](#_Toc71971035)

[第八章 GeekOS设计项目6—进程通信 29](#_Toc71971036)

[8.1 Project6项目原理分析 29](#_Toc71971037)

[8.1.1 项目6设计目的 29](#_Toc71971038)

[8.1.2 项目6设计原理及分析过程 29](#_Toc71971039)

[8.2 Project6项目运行分析 30](#_Toc71971040)

[第九章 课程设计心得体会总结 32](#_Toc71971041)

[参考文献 33](#_Toc71971042)

# 第一章 引言

操作系统是管理[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA" \t "_blank)[硬件](https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6)与[软件](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6)资源的[计算机程序](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A8%8B%E5%BA%8F)。操作系统需要处理如管理与配置[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98)、决定[系统资源](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E8%B5%84%E6%BA%90/974435)供需的优先次序、控制[输入设备](https://baike.baidu.com/item/%E8%BE%93%E5%85%A5%E8%AE%BE%E5%A4%87/10823368)与[输出设备](https://baike.baidu.com/item/%E8%BE%93%E5%87%BA%E8%AE%BE%E5%A4%87/10823333)、操作网络与管理文件系统等基本事务。操作系统也提供一个让用户与系统交互的操作界面。

GeekOS是一个基于x86体系结构的微操作系统内核，同时也是一个用C语言编写的开源操作系统项目。通过本课程的设计，进行一个小型操作系统Geekos的实现。我们需要在Linux环境下扩展它的功能。在每个项目中，我们可以通过阅读分析源代码并根据提示完成相关功能。完成基本功能后，可以逐步实现一个微操作系统，使其在bochs模拟器上正常运行。

# 1.1 课程设计开发背景

# 1.1.1 GeeKOS课程设计任务概述

在本次的操作系统课程设计中，需要在Linux环境下使用一个基于X86架构的PC机上运行的小型操作系统GeekOS来进行操作系统工作原理的理解，以及对其进行功能的扩充来完善一个操作系统。完成GeekOS预留的project0-project6项目，运行Bochs模拟器，查验项目结果。

# 1.1.2 GeeKOS实验环境

Bochs是一个x86硬件平台的开源模拟器。它可以用来模拟各种硬件的配置。包括I/O设备、内存和BIOS。它可以在任何编译运行Bochs的平台上模拟x86硬件。通过改变配置，可以指定使用的CPU以及内存大小等。Bochs可以被编译运用在多种模式下，其中有些仍处于发展中。Bochs的典型应用是提供x86 PC的完整仿真，包括x86处理器、硬件设备和存储器。为了模拟一台计算机执行一个操作系统软件，Bochs 需要编写.bochsrc 配置文件用于描述模拟器的硬件配置，如图1.1所示。

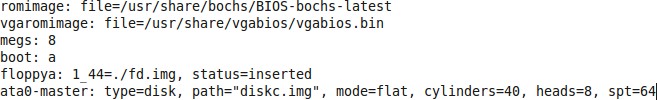


图1.1 .bochsrc 配置文件

Bochs 在启动时会根据.bochsrc配置文件初始化模拟器的硬件配置。在其配置文件中，第一行意思是模拟 bochs 硬件的 BIOS为romimage ；模拟bochs 显示系统的 BIOS为vgaromimage ；megs 设置模拟器内存为 8M；然后boot 设置系统引导方式为磁盘引导；在floppya 设置了模拟器启动的镜像文件；最后一行ata0-master 配置 ata 串口驱动器。

Make 工具会根据 Makefile 文件执行对项目代码文件的编译操作。 它能够根据makefile文件的规则自动完成相应的编译工作，同时只会对上次修改过的文件进行编译，减少重复编译的工作量。

# 1.2 课程设计具体开发步骤

1.下载并安装VMware虚拟机，在VWware虚拟机上安装linux操作系统。

1. 在终端输入 sudo apt-get install build-essential //下载安装 build-essential包。
2. 在终端执行sudo apt-get install nasm //下载安装NASM包
3. 在终端执行 sudo apt-get install bochs

在终端执行 sudo apt-get install bochs-x

在终端执行 sudo apt-get install bochs-sdl

安装Bochs

2.开始一个GeekOS项目，第一步是添加相应的代码，接着对Geek0S源文件进行编译和链接。

修改Makefile文件：

CC\_GENERAL\_OPTS := $(GENERAL\_OPTS) -Werror 这一行改成

CC\_GENERAL\_OPTS := $(GENERAL\_OPTS) -O0

在gcc 后面加上-fno-stack-protector

3.在Linux下利用make命令编译系统编译成功后生成fd. img软盘映射文件和hd. img硬盘映射文件。  
 $cd /projectx/build  
 $make depend  
 $make  
4.编写每个项目相应的Bochs的配置文件，运行Bochs模拟器，执行GeekOS内核。

5. 输入bochs命令后，会出现一些提示。如果编译成功，且bochs的配置文件也没问题，会看到一个模拟VGA的文本窗口，然后GeekOS就能会行程序在文本窗口输出相应的信息。

# 第二章 GeekOS设计项目0—GeekOS系统环境调试及编译

# 2.1 Project0项目原理分析

# 2.1.1 项目0设计目的

熟悉GeekOS的项目编译、调试和运行环境，掌握GeekOS运行工作过程。

(1)搭建GeekOS的编译和调试平台，掌握GeekOS的内核进程工作原理。

(2)熟悉键盘操作函数，编程实现一个内核进程。该进程的功能是：接收键盘输入的字符并显示到屏幕上，当输入ctrl+d时，结束进程的运行。

# 2.2.2 项目0设计原理及分析过程

Project0项目主要实现能够接收键盘输入的字符并显示，最后输入ctrl+d时结束运行的内核进程。整个实现流程主要分为内核线程的建立和键盘IO处理两部分内容：

（1）建立内核进程：

经过阅读分析源代码，GeekOS在kthread.c定义有内核线程的创建、内存空间的分配、初始化等内核线程操作函数， 在kthread.h定义有内核线程结构struct Kernel\_Thread，其中包含esp内核堆栈指针、numTichs字段是存放计时器、priority进程优先级等成员。

而Start\_Kernel\_Thread（src/geekos/kthread.c）是建立内核进程的核心函数，其需参数startFunc函数指针，指向内核线程入口的函数体；arg是传递给入口函数的参数；priority是线程的优先级，detached表示线程是否为子线程的标志，返回值是返回一个生产的线程指针。

Start\_Kernel\_Thread函数内部首先调用Create\_Thread()创建一个线程，接着调用Alloc\_Page()分配内存空间，进行Init\_Thread()初始化线程操作。下一步调用Add\_To\_Back\_Of\_All\_Thread\_List()和Setup\_Kernel\_Thread() 配置内核线程，Make\_Runnable\_Atomic()设置线程运行的原子性操作，最后通过Disable\_Interrupts()和Make\_Runnable()禁止中断和使能中断，过程如图2.1所示。

Start\_Kernel\_Thread

Create\_Thread

Setup\_Kernel\_Thread

Make\_Runnable\_Atomic

Alloc\_Page

Init\_Thread

Add\_To\_Back\_Of\_All\_Thread\_List

图2.1 GeekOS系统系统内核线程创建

（2）键盘IO处理

分析源代码可发现，Main函数中调用Init\_Keyboard函数进行键盘处理初始化，Init\_Keyboard 主要功能是设置初始状态键码缓冲区，并为键盘中断设置处理函数。任何操作都会引发中断处理函数，其关键在于根据是否按下Shift键，寻找对应按键值。如果需要获得键盘输入只要调用函数Wait\_For\_Key()，调用该函数后会阻塞进入按键操作的等待队列，直到按键操作结束，进程才被唤醒。

该函数在keyboard.c里面定义，其作用是循环等待一个键盘事件，然后返回一个16位的Keycode类型的值，这个值也就是我们所按下的键的键值。Read\_Key(Keycode\* keycode) )函数可以处理队列键盘按键,可以保存到队列中并输出。键盘大部分键的键值定义在keyboard.h和keyboard.c中，用16位的Keycode数据类型来定义，低10位用来表示键盘值,通过s\_scanTableNoShift和s\_scanTableWithShift这两个数组来转换相应的键盘码为所表示字符的ASCII码。

Wait\_For\_Key()和Read\_Key()函数都是和键盘输入相关的函数，最后通过内置Print()函数进行显示到屏幕上，完整流程如图2.2所示。

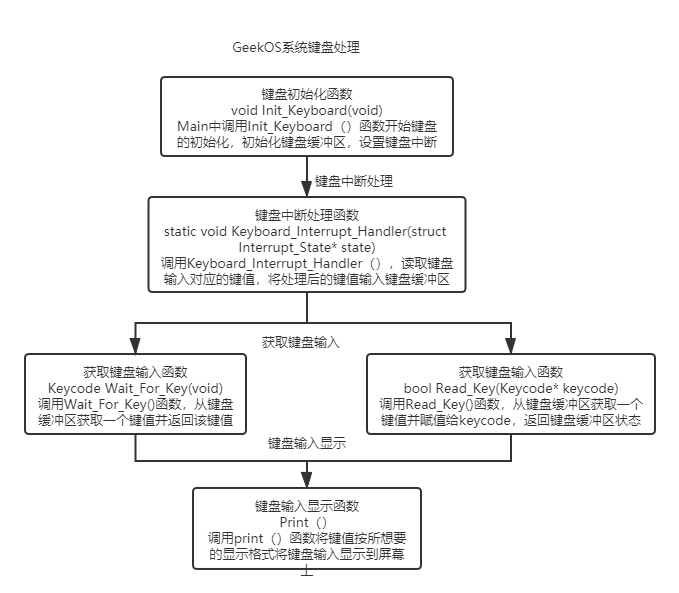


图2.2 GeekOS系统键盘处理过程

# 2.2 Project0项目运行分析

在main.c设计IO\_Process()函数，接收键盘输入的字符并显示，当输入ctrl+d时，结束进程运行的函数。接着主函数中调用thread = Start\_Kernel\_Thread(&IO\_ Process(),0,PRIORITY\_NORMAL,true)。

IO\_Process()函数如下：

void IO\_Process(){

    Print("To Exit hit Ctrl + d.\n");

    Keycode keycode;

    while(1)

    {

        if(Read\_Key(&keycode))

        {

            // Print("%x",keycode & 0xff);

            if(!((keycode & KEY\_SPECIAL\_FLAG) || (keycode & KEY\_RELEASE\_FLAG)))// 不是特殊键或者弹起

            {

                int asciiCode = keycode & 0xff;//d

                if((keycode & KEY\_CTRL\_FLAG)==KEY\_CTRL\_FLAG && asciiCode=='d')//ctrl+d

                {

                    Set\_Current\_Attr(ATTRIB(BLACK, GREEN|BRIGHT));

                    Print("\nOS has received Ctrl + d, See you next time~\n");

                    Exit(1);

                }else

                {

                    Set\_Current\_Attr(ATTRIB(BLACK, GREEN|BRIGHT));

                    Print("%c", (asciiCode=='\r') ? '\n' : asciiCode);

                }

            }

        }

    }

}

程序运行结果：

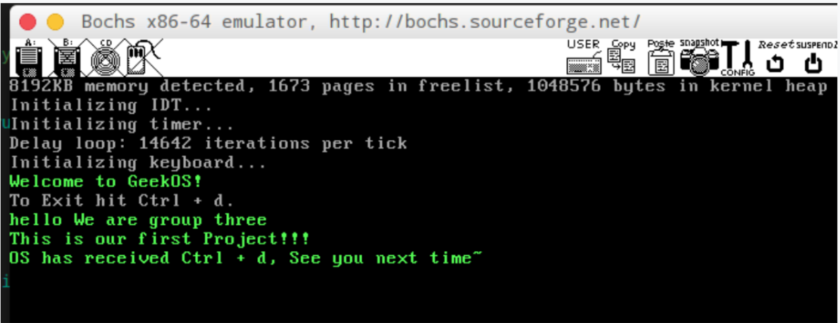


图2.3 程序运行结果

从运行结果图2.3可以看出，当程序开始运行，此时可以从键盘向系统输入一些字符并在屏幕上显示出来，为了演示效果，我们对输入的内容进行高亮显示。

输入完毕之后，按下“Ctrl+d”组合键时，根据程序调用内核线程函数Start\_Kernel\_Thread，判断(key&KEY\_CTRL\_FLAG)==KEY\_CTRL\_FLAG&&asciiCode=='d'，此时Ctrl信号有效且ascii为d，所以程序打印成功退出的信息然后就会退出。

# 第三章 GeekOS设计项目1—内核级线程设计及实现

# 3.1 Project1项目原理分析

# 3.1.1 项目1设计目的

熟悉ELF文件格式，了解GeekOS系统如何将ELF格式的可执行程序装入到内存，建立内核进程并运行的实现技术。修改/geekos/elf.c文件：在函数Parse\_ELF\_Executable( )中添加代码，分析ELF格式的可执行文件（包括分析得出ELF文件头、程序头，获取可执行文件长度，代码段、数据段等信息），并填充Exe\_Format数据结构中的域值。

# 3.1.2 项目1设计原理及分析过程

项目1关键在于完善Parse\_ELF\_Executable( )函数解析ELF文件，ELF是Unix系统实验室作为应用程序二进制接口而开发和发布的，EIF文件格式如表3-1所示。

表3-1 ELF目标文件格式

|  |  |
| --- | --- |
| 连接程序视图 | 执行程序视图 |
| ELF 头部 | ELF 头部 |
| 程序头部表（可选） | 程序头部表 |
| 节区1 | 段 1 |
| ... |
| 节区 n | 段 2 |
| ... |
| 节区头部表 | 节区头部表（可选） |

下面是在本项目中使用到的相关结构体，其余的节区头部表等结构体在本次实验中并未使用到。

ELF文件头部结构体如下：

typedef struct {

unsigned char ident[16];

unsigned short type;

unsigned short machine;

unsigned int version;

unsigned int entry;

unsigned int phoff;

unsigned int sphoff;

unsigned int flags;

unsigned short ehsize;

unsigned short phentsize;

unsigned short phnum;

unsigned short shentsize;

unsigned short shnum;

unsigned short shstrndx;

} elfHeader;

程序头部表结构体如下：

typedef struct {

unsigned int type;

unsigned int offset;

unsigned int vaddr;

unsigned int paddr;

unsigned int fileSize;

unsigned int memSize;

unsigned int flags;

unsigned int alignment;

} programHeader;

分析程序可知，位于user目录下的用户程序在系统的编译阶段完成编译和链接，形成可执行文件，可执行文件保存在PFAT文件系统中。本项目主要实现的是，系统启动后，从PFAT文件中把可执行文件装入内存，建立进程并运行得到相应的输出。具体大致流程为：通过Spawner函数中的Read\_Fully函数先将ELF文件读入内存缓冲区，成功读入到内存缓冲区后。使用我们填充后的Parse\_ELF\_Executable( )来去解析我们之前读入的ELF文件。获取ELF文件头,程序头,可执行文件长度,代码段,数据段等信息,并填充Exe\_Format数据结构的各值，然后传入spawn program函数中，去执行ELF文件。通过结构体中存放的信息来存入计算用户进程所需的内存，分配对应的内存空间，并全部初始化为零，具体两者映射关系如图3.1所示，图中红框部分为Exe\_Format文件镜像，可以看到，实际内存分区和ELF文件存在偏移映射。

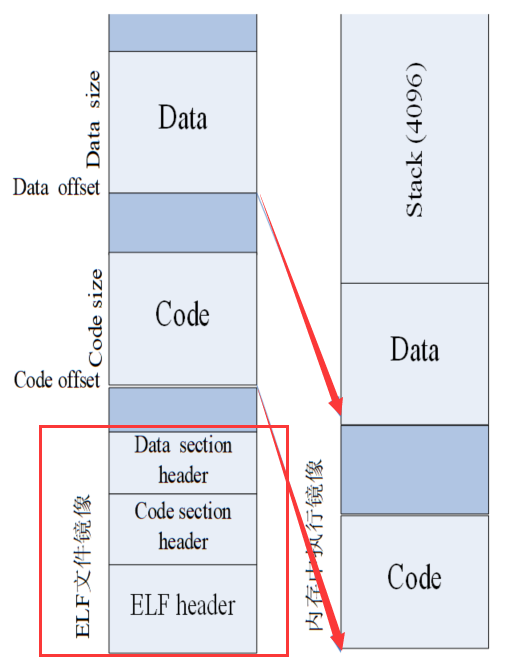


图3.1 ELF文件与内存镜像映射

上述Spawner函数作为start\_Kernel\_Thread()的第一个参数进行加载，在此函数内部按照create\_Thread() 、Setup\_KernelThread()、MakeRUnnable\_atomic()三个函数顺序执行。在create\_Thread()函数内部，使用kernel\_thread 结构体的指针，来为进程上下分配页，对象和线程的堆栈，即创建PCB。之后通过Init\_Thread函数初始化Kernel\_Thread结构体。然后将这个PCB添加到进程队列中。PCB创建完成后，需要为进程创建上下文使其可进行调度，通过Setup\_Kernel\_Thread函数来实现该功能。这个函数初始化进程栈区，如参数地址，返回地址，入口地址等内容，来实现目标功能。完成上述功能后，顺序执行到Make\_Runnable\_Atomic函数，该函数是使进程能够以原子方式运行，即运行时不会被中断。以上为project1的整个分析执行流程。

# 3.2 Project1项目运行分析

在main.c设计Parse\_ELF\_Executable ()函数，进行ELF文件格式解析，函数如下所示。

int Parse\_ELF\_Executable(char \*exeFileData, ulong\_t exeFileLength,

    struct Exe\_Format \*exeFormat)

{

    //利用ELF头部结构体指向可执行文件头部，便于获取相关信息\*/

    //获取ELF文件头,程序头,可执行文件长度,代码段,数据段等信息,并填充Exe\_Format数据结构的各值

    elfHeader \*ehdr = (elfHeader\*)exeFileData; //ELF文件头结构体

    Set\_Current\_Attr(ATTRIB(BLACK, BLUE|BRIGHT));

    Print("e\_ident:%x %c %c %c\n",ehdr->ident[0],ehdr->ident[1],ehdr->ident[2],ehdr->ident[3]);

    //type为2表示可执行文件

    Print("e\_type:%x  e\_machine:%x  e\_version:%x  ELF\_size:%xH\n",ehdr->type,ehdr->machine,ehdr->version,ehdr->ehsize);

    Set\_Current\_Attr(ATTRIB(BLACK, GRAY));

    //program header table表项的个数

    exeFormat->numSegments = ehdr->phnum;

    //代码入口地址

    exeFormat->entryAddr = ehdr->entry;

    //获取头部表在文件中的位置，便于读取信息

    programHeader \*phdr = (programHeader\*)(exeFileData + ehdr->phoff);

    //填充Exe\_Segment

    unsigned int i;

    for(i = 0; i < exeFormat->numSegments; i++, phdr++)

    {

        struct Exe\_Segment \*segment = &exeFormat->segmentList[i];

        //获取该段在文件中的偏移量\*

        segment->offsetInFile = phdr->offset;

        //获取该段的数据在文件中的长度

        segment->lengthInFile = phdr->fileSize;

        //获取该段在用户内存中的起始地址

        segment->startAddress = phdr->vaddr;

        //获取该段在内存中的大小

        segment->sizeInMemory = phdr->memSize;

        //获取该段的保护标志位

        segment->protFlags = phdr->flags;

    }

编写完函数后，进入geekos-0.3.0/src/project1/build 目录中，执行bochs命令，启动运行后得到如图3.2所示结果。

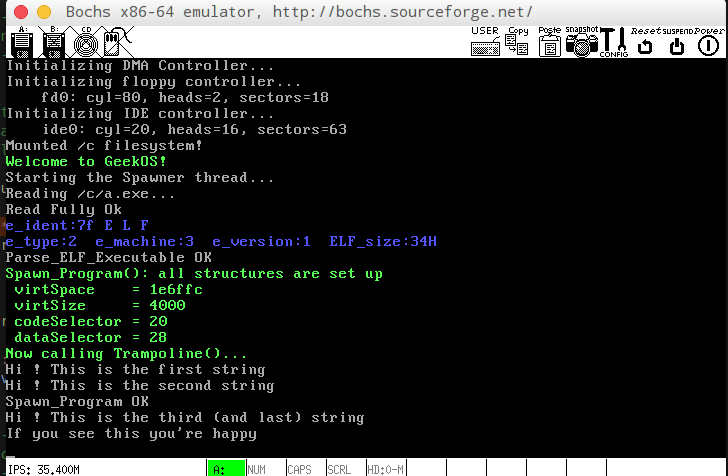


图3.2 project1运行结果

由结果可以看出，程序获取ELF文件后，按照创建内核流程，系统对关键流程以及ELF核心解析内容进行打印，其中蓝色字体部分高亮显示了ELF32\_Ehdr结构中的e\_ident字段，输出7f E L F表明这是ELF文件格式，以及输出了e\_type、e\_machine、e\_version、ELF\_size等部分关键信息。接着在解析完ELF文件过后，输出exeFormat数据结构中的virtSpace、virSize等关键信息。最后五行显示的是执行/a.c用户进程输出的内容，以上便是项目1的全部内容。

# 第四章 GeekOS设计项目2—用户级进程的动态创建与执行

# 4.1 Project2项目原理分析

# 4.1.1 项目2设计目的

扩充GeekOS操作系统内核，使得系统能够支持用户级进程的动态创建和执行。在project2项目中，需要在project1的基础上实现更多的函数，完善内核功能，实现真正的用户态进程创建。主要实现的函数有Spawn()，Switch\_To\_User\_Context()，Load\_User\_Program()等。

# 4.1.2 项目2设计原理及分析过程

GeekOS运行main.c文件中的Main()函数开始，在初始化系统数据后，使用Spawn\_Init\_Process为用户程序指定进程所需的进程。接着使用Spawn()函数寻找可执行文件并装载应用程序进内存空间，解析其可执行文件的信息，判断其是否为用户进程，如果是，为该可执行文件创建并完善一个User\_context结构体信息，函数最后返回改用户进程。

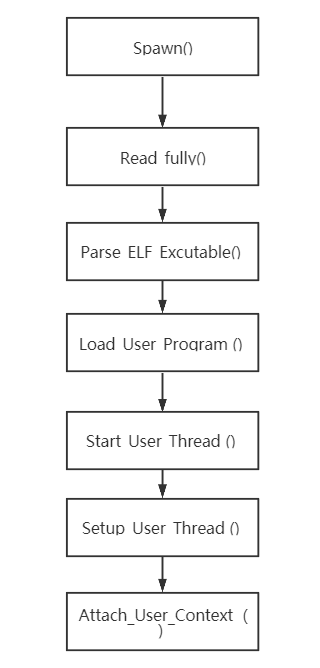


图4.1 用户进程创建流程

如图4.1所示，Spawn()函数在读取完可执行文件的文件结构之后，将该可执行文件的信息以及新开辟的User\_Context传入Load\_User\_Program()中，该函数将解析出该可执行文件的用户地址空间，用于完善传入的User\_context，如图4.2所示。

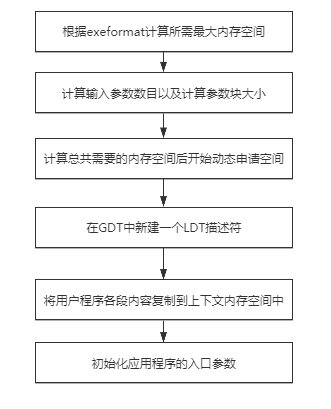


图4.2 Load\_User\_Program()实现流程

完善User\_Context需要用到一张LDT表，对于每一个进程，都可以使用一张LDT表来存储其文件的入口地址。LDT表的一个索引将会被存储在GDT表中，由于本项目的GDT长度固定，需要注意的是最多只能申请16个进程

得到完整的User\_Context后，使用Start\_User\_Thread()函数，传入User\_Context，该函数将根据传入的信息创建进程，分配其优先级，然后使用Setup\_User\_Thread()函数根据User\_Context内容来初始化用户态进程堆栈，使之看上去像刚被中断运行一样，分别调用 Push 函数来向堆栈压入DS选择子，堆栈指针，Eflags,CS选择子，程序计数器，错误代码，中断号，初始化通用寄存器以及数据段单元，如图4.3所示。

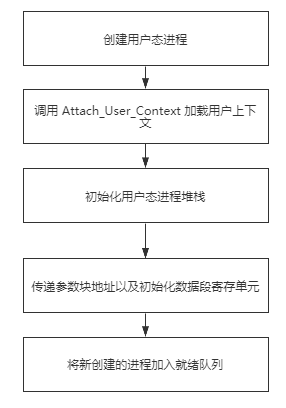


图4.3 得到user\_context后创建用户态进程

当已经创建用户进程后，需要去运行新的用户进程时，需要用Switch\_To\_Address\_Space()对当前运行进程切换，如图4.4所示。

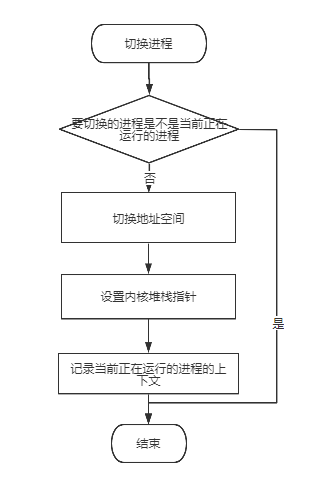


图4.4 切换当前运行的进程

当要切换进程时，需要判断保存在静态变量的当前运行进程的上下文是不是和将要运行的进程的上下文相同。若相同，则不需要进行切换，否则就要把核心栈指针和地址空间切换到将要运行的进程。

系统运行的第一个用户进程是以shell.c为执行文件，创建的user\_context，最后生成第一个用户进程。Shell主要执行功能如图4.5所示。

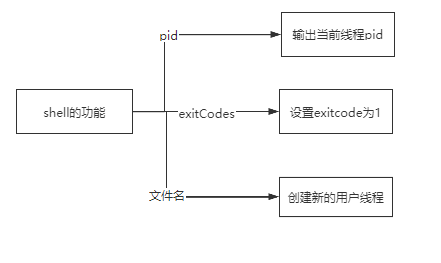


图4.5 shell文件执行的功能

当用户输入pid时，boch界面会输出当前运行的进程的pid。当输入exitcode时，进程会在每次循环后多输出一句Exit code was 1。当输入的是目标文件名称时，就会建立这个目标文件对应的上下文以创建新的用户进程。

# 4.2 Project2项目运行分析

项目2中“user.c”文件中的函数Spawn("/c/shell.exe","/c/shell.exe",&pThread) 用于生成一个用户及进程，需要传入可执行文件地址以及指定的进程指针。

函数Switch\_To\_User\_Context(struct Kernel\_Thread\*, struct Interrupt\_State\*)，切换到指定的用户进程，需要传入一个进程指针。

在“userseg.c”文件中实现Load\_User\_Program(char \*, ulong\_t,struct Exe\_Format \*, const char \*,struct User\_Context \*\*)函数的功能通过加载可执行文件镜像创建新进程的User\_Context结构，通过elf文件格式获取对应的上下文，传入参数为执行文件内容，地址，文件段信息以及用户输入的命令参数。 Switch\_To\_Address\_Space（struct User\_Context \*）函数的功能是通过将进程的LDT装入到LDT寄存器来激活用户的地址空间。

最后，“kthread.c”文件中的Start\_User\_Thread(struct User\_Context\*, bool)函数，输入参数为用户上下文和Setup\_User\_Thread(struct Kernel\_Thread\*, struct User\_Context\*)函数，参数为用户进程和对应的上下文。

Project2用户级线程设计运行如图4.6所示。

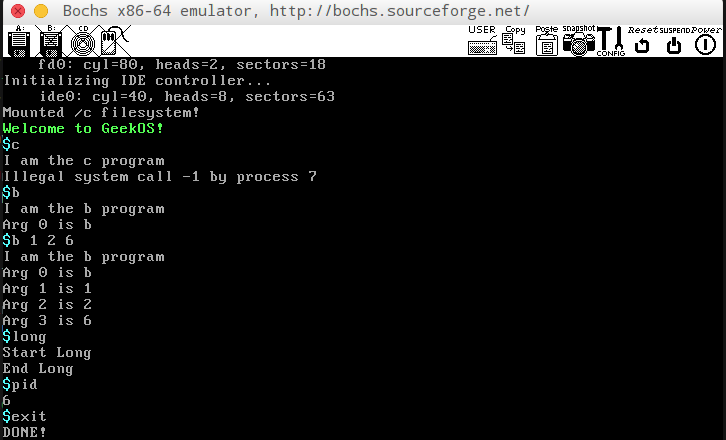


图4.6 project2运行结果

在对应的build目录下输入bochs后进入bochs界面。输入c可以生成一个c为执行文件的进程，并输出当前创建的进程号。输入b以及b 1 2 3可得到对应的输出结果。可以看到b.exe可以处理输入的参数，并且可以按照参数个数进行输出。

输入pid后得到当前shell的进程号为6，也是第一个用户级进程，因为在系统初始化后，除了一开始有通过Init\_Scheduler 函数创建Main、Idle和Reaper三个进程外，还创建了两个进程Init\_Floppy()初始化软盘和Init\_IDE()初始化硬盘。因此该shell进程号才是6，并且只有在另外开启一个shell进程才会显示当前最新pid的数值。

此外，系统还可以运行内置的long、null、exit等子用户进程，如图4.6所示。

# 第五章 GeekOS设计项目3—进程调度算法与信号量功能

# 5.1 Project3项目原理分析

# 5.1.1 项目3设计目的

研究进程调度算法，掌握用信号量实现进程间同步的方法。为GeekOS扩充进程调度算法—基于时间片轮转的进程多级反馈调度算法，并能用信号量实现进程协作。

(1)实现 src/geekos/syscall.c 文件中的 Sys\_SetSchedulingPolicy 系统调用，它的功能是设置系统采用的何种进程调度策略；

(2)实现 src/geekos/syscall.c 文件中的 Sys\_GetTimeOfDay 系统调用，它的功能是获取全局变量g\_numTicks的值；

(3)实现函数 Change\_Scheduling\_Policy()，具体实现不同调度算法的转换。

(4)实现 syscall.c 中信号量有关的四个系统调用：sys\_createsemaphore( )、sys\_P( )、sys\_V( )和 sys\_destroysemaphore( )。

# 5.1.2 项目3设计原理及分析过程

1. 进程调度算法具体分析

GeekOK系统提供的进程调度是时间片轮转调度（RR）。而在project3中，需要实现引入四级反馈队列调度算法（MLF），用四个准备运行队列替代初始时间片轮转调度中的一个队列，改进调度策略，需要补充的调度函数位于syscall.c和kthread.c中。

时间片轮转调度算法流程：起初所有准备运行进程都放在一个FIFO队列里面，新建进程放在该队列尾部。发生进程调度时，系统在准备运行队列中查找优先级最高的进程投入运行。

四级反馈队列调度算法流程：进程就绪队列分为4级，按照优先级从高到低排列分别为Q0、Q1、Q2和Q3队列；新创建的进程会被置入最高优先级的就绪队列Q0；每当一个进程运行完一个时间片长度之后，它就会被置入比之前低一级的就绪队列，直到到达优先级最低的队列Q3，该流程如图5.1所示。

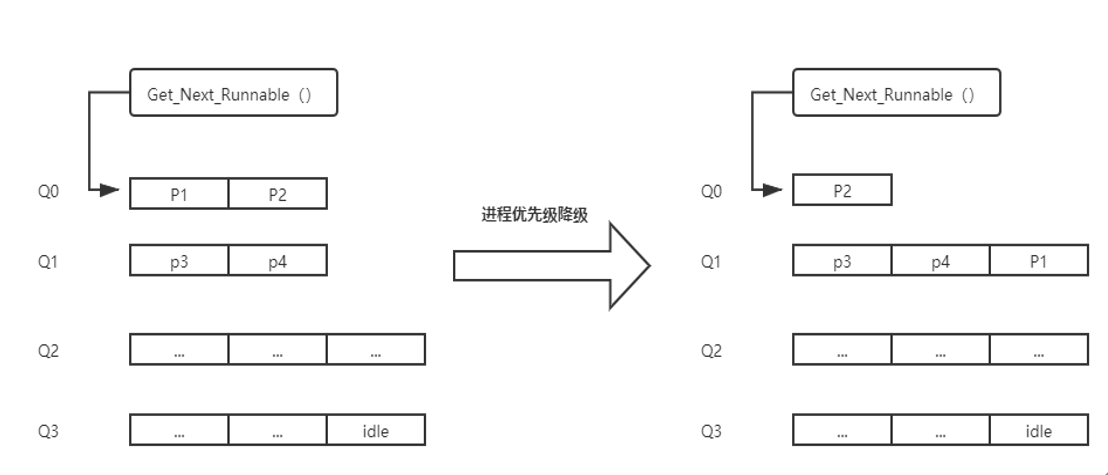


图5.1 四级反馈队列调度优先级降级

此外，如果进程被阻塞，则队列优先级就会提升等级，直到被阻塞三次后达到最高优先级队列Q0，系统中的空闲进程Idle会始终放在优先级最低的队列Q3的尾部，以便系统中没有其他可调度进程时就运行它，具体调度流程如图5.2所示。

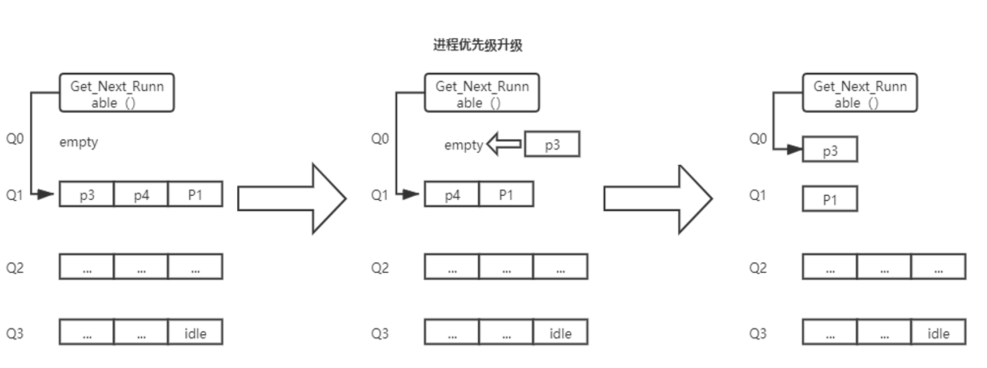


图5.2 四级反馈队列调度优先级升级

接着，系统需要调用Sys\_SetSchedulingPolicy()函数设置待采用的调度算法。其中，函数参数state-> ebx用于顶层调度策略，0表示采用时间片轮循调度策略，1表示系统使用4级反馈队列调度策略。

当系统切换流程调度策略时，需要调用Chang\_Scheduling\_Policy()来修改线程队列和与系统相关的变量，其参数g\_curSchedulingPolicy表示当前的调度策略，参数g\_Quantum表示时间片长度。具体转换流程如图5.3所示，当RR算法转换为MLF算法时，通过将最初位于Q0队列中的Idle空闲进程移至Q3队列的末尾，并且不需要修改其他进程。当MLF算法转换成RR算法时，则Q1-Q3队列中的所有进程都转移到Q0队列，然后按优先级顺序重新排序从高到低。

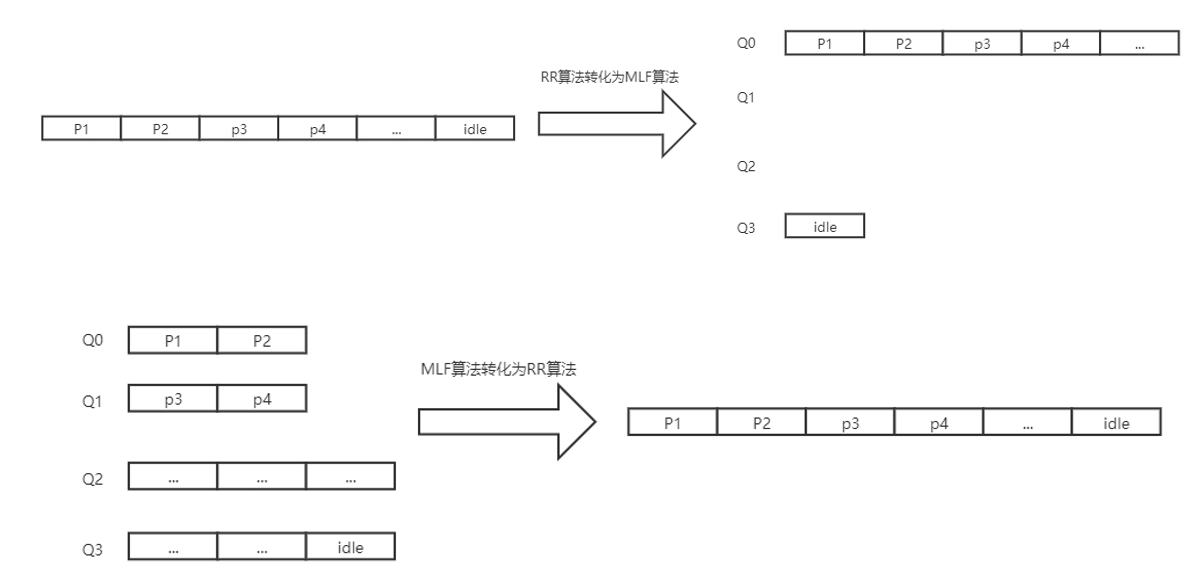


图5.3 多级反馈队列与分时调度进程队列的转换

接着，进程调度是在时钟中断处理函数Time\_Interrupt\_Handle()中实现的。具体涉及Kernel\_

Thread结构体的numTicks变量，该变量初始化为零，当每次触发时间中断，该变量增加1，并检查进程的执行时间是否超过系统指定的时间片g\_Quantum。如果超过该时间，则当前流程时间片已用完，系统从调用Make\_Runnable()函数将当前正在运行的进程放入准备运行的进程队列。

此外，在时钟中断处理函数Time\_Handle\_Interrupt()，检查g\_needReschedule变量。如果为true，则调用Get\_Next\_Runnable函数查找优先级最高的进程；最后将g\_needReschedule返回为false，并切换到新进程来运行，过程如图5.4所示。

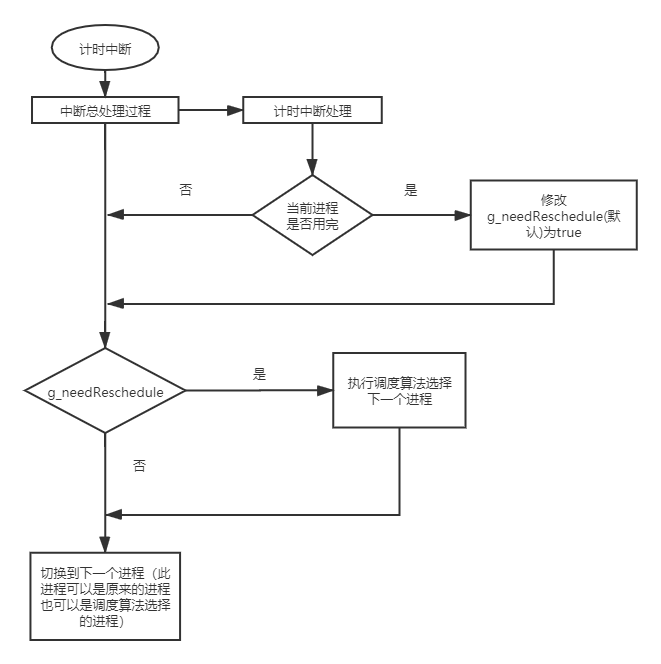


图5.4 进程调度处理过程

2. 信号量PV操作具体分析

信号量结构由信号量ID、名称、已注册的线程数、等待信号的线程队列等构成。信号量的创建和销毁、，P V操作均在syscall.c文件中实现。

在信号量创建Create\_Semaphore()函数中，首先检查请求创建的信号量名称是否存在，如果存在，将该线程添加到该信号量注册的线程链表中，如果不存在，则为新信号量分配内存，清除其线程队列，将当前线程添加到其线程队列，将注册线程数设置为1，返回信号量ID。

同理，在信号量销毁函数Semaphore\_Destroy()中，检查完信号量无误后，则从该信号量的已注册线程数组中删除该线程，并从已注册线程数中减去1。如果该信号量的注册线程为0，则从信号量链接列表中删除该信号量，并释放其内存。

在P操作函数Sys\_P()中，如果信号量ID不存在或者并没有在该线程注册，则返回-1，表示操作失败。如果成功，则从信号量的值中减去1，但是如果减1后信号量小于0，则将当前线程移入该信号量的等待队列中。

同理，在V操作函数Sys\_V()中，如果信号量ID不存在或者并没有在该线程注册，则表示操作失败。成功则将信号量的值加1，判断信号量加1后如果大于或等于1，将等待队列上的线程进行唤醒。

# 5.2 Project3项目运行分析

首先进行线程调度算法测试，调用schedtest和workload 两个用户程序进行测试。Schedtest用户进程需要传入两个参数，rr/mlf和时间片大小。

（1）时间片相同，调度算法不同测试： schedtest rr 10 和 schedtest mlf 10

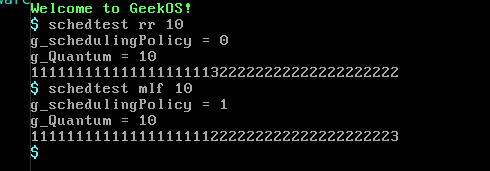


图5.5 schedtest测试用例使用两个调度算法结果

（1）调度算法相同，时间片不同测试： schedtest rr 1 和 schedtest rr 10

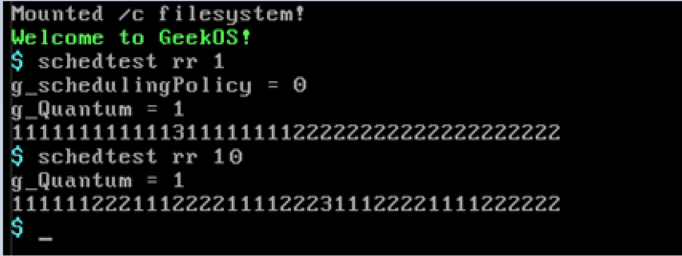


图5.6 schedtest测试用例使用相同调度算法结果

从图5.5和图5.6中可以看到，在schedtest测试用例中，由于RR算法和MLF算法基于时间片旋转，并且时间片的长度相同，因此输出数1、2和3相同，但是在时间片不同的条件下，由于RR算法的过程调度没有优先级调整，而MLF算法进行优先级配置，因此进程切换的时间可能不同，程序执行的周转时间可能不同，所以二者输出结果可能有所不同。

接着进行workload负载测试，测试结果如图5.7所示。

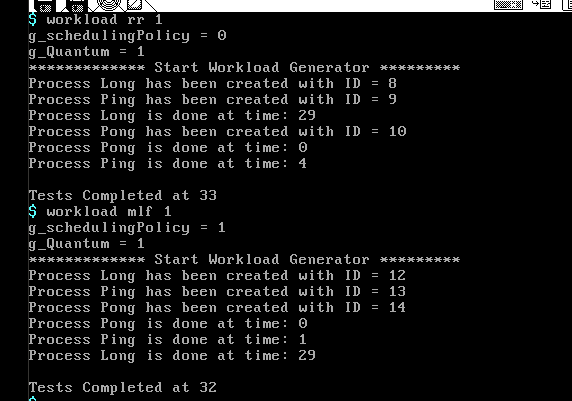


图5.7 workload测试用例使用两个调度算法结果

在工作负载测试案例中，采用Long、Ping两个用户进程计算调度时间，测试结果时间也不同，这也因为MLF算法中存在优先级调整，并且MLF算法中每个队列的时间片都是相同，因此每个过程运行所需的时间都少于RR算法中的时间，总体上表明MLF调度算法优于RR算法。

最后采用semtest、semtest1、semtest2三个用户进程对信号量进行功能测试，其结果如图5.8、图5.9、图5.10所示。



图5.8 semtest测试用例运行结果

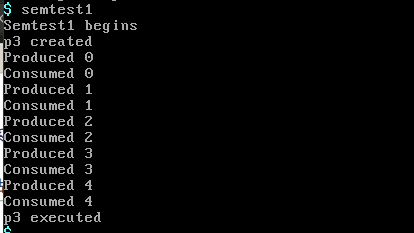


图5.9 semtest1测试用例运行结果

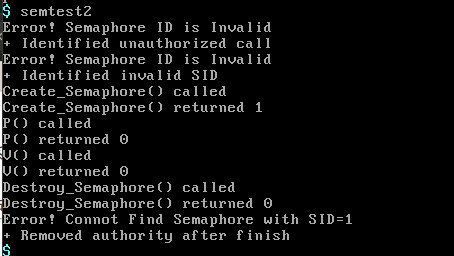


图5.10 semtest2测试用例运行结果

从运行测试结果来看，在semtest测试用例中，可以正确执行已实现的信号量相关操作，PV操作返回0表明该操作成功，信号创建和销毁成功也返回0。

在semtest1测试用例中，信号量通过P操作和V操作实现过程同步，成功解决生产者和使用者的问题。在semtest2测试用例中，系统可以对不正确的信号量操作执行相应处理，从侧面论证了信号量部分相关代码的正确性。

# 第六章 GeekOS设计项目4—分页存储管理机制

# 6.1 Project4项目原理分析

# 6.1.1 项目4设计目的

了解虚拟存储器管理设计原理，掌握请求分页虚拟存储管理的具体实现技术。实现Init\_VM()、Init\_Paging()、Find\_Space\_On\_Paging\_File()、Free\_Space\_On\_Paging\_File()、Write\_To\_Paging\_File()、Read\_From\_Paging\_File()、Destroy\_User\_Context()等函数。

# 6.1.2 项目4设计原理及分析过程

项目4关键在于将GeekOS原生的分段式存储管理方式变成分页虚拟存储管理方式，通过增加分页管理引入地址映射，分页系统把线性地址映射到物理地址，这也是分页系统需要实现的第一步。因此，第一步建立页表需要通过完善实现Alloc\_Page()函数，为存储区域内容分配页表，并填充页目录表项和对应内容。页目录表项和页目录结构体定义如下图6.1所示。

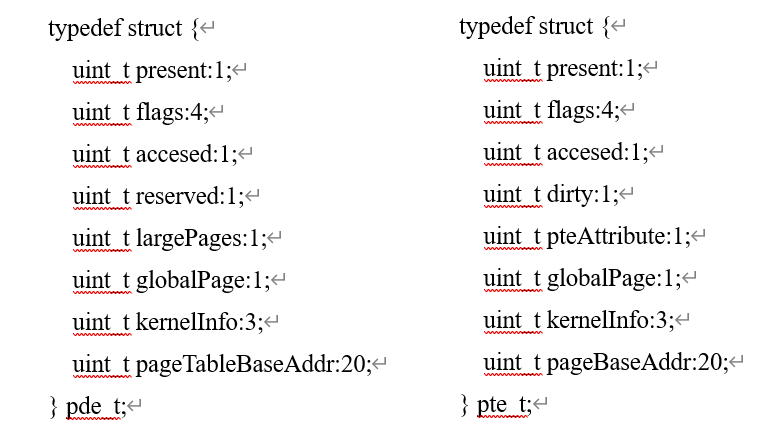


图6.1 页目录表项和页目录结构体

在内核进程中，需要把页表项对应标志位设置为非VM\_USER，保证只允许内核进程进行访问。而Alloc\_Page()函数是在Init\_VM()函数中进行定义的，最后Init\_VM()需要加入缺页中断处理程序Page\_Fault\_Handler，通过Install\_Interrupt\_Handler函数进行安装该中断处理程序，中断号为14。最后在main.c文件调用Init\_VM()初始化虚拟内存管理。

此外，在启动时内核会创建一个页面目录和页面，直接将所有物理内存页面映射到内核空间的虚拟地址的表，如图6.2所示，内核虚拟内存从地址0x00000000开始，大小为4GB。从在用户空间视图中，用户虚拟内存从地址0x00000000开始，大小为2GB。

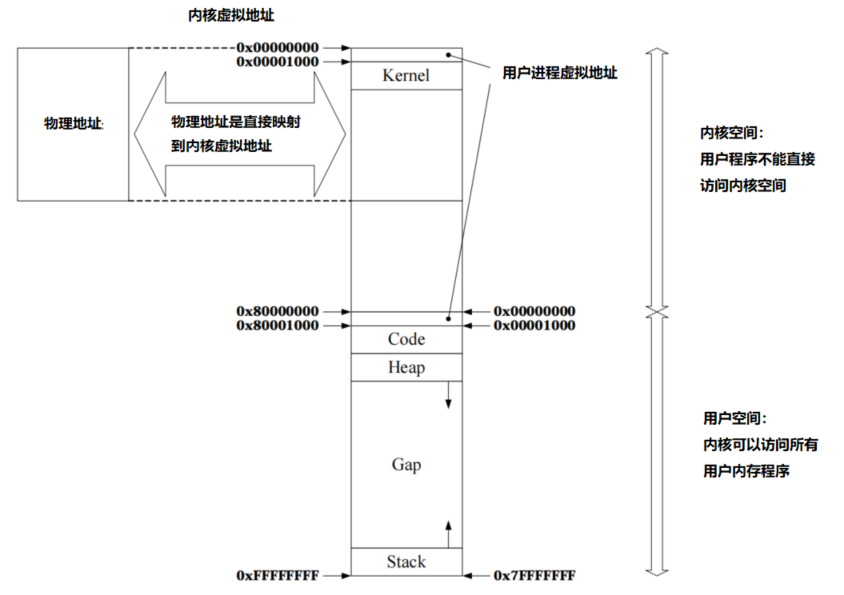


图6.2 虚拟内存管理分布

虽然所有内核线程共享一个页面目录及其页表，但每个用户进程都拥有一个单独的页目录。在创建用户进程时，内核复制内核空间页表引用用户页面目录，并为用户空间内存创建不连接的页表。对于用户进程，内核只分配启动所需的内存。页面错误处理程序在需要时向堆栈或堆添加额外的页面。图6.2还描述了第一页进程和线程第六页内核空间和用户空间的第一页没有映射到物理内存。这个将有助于在内核以及用户程序中捕获空指针引用。

在完成内核进程分页相关配置后，第二步是修改用户进程的存储区域采用分页措施。其中，需要通过一个名为uservm. c用户分页进程替换原文件userseg.c用户分段进程。它们中定义的功能名称是相同的，并且要完成的功能是相同的。区别在于分段系统是在userseg.c(已经在前面项目完成了)中实现的，而分页系统是在uservm.c中实现的。

uservm.c的Load\_User\_Program函数分两步实现，首先为进程分配一个页目录表，复制线性存储空间的低2GB的所有核心页目录表项(在 Init\_VM函数中设置）到用户进程的页目录表，然后为接下来需要为用户进程的代码、数据和堆栈区间分配页表项。三个区间中的每一个将由不同数目的页组成，这些页是由函数Alloc\_Pageable\_Page来分配的。该函数分配的页将返回一个特殊标志PAGE\_PAGEABLE，它在设置在页数据结构表项的标志域。

接着对当前系统使用的数据进行修改，以便系统可以正常使用分页。用户进程线性地址的基地址应为0x8000 0000，而边界地址应被定义为OxFFFF FFFF。用户还需要添加代码来切换PDBR寄存器的内容。加载LDT之后，功能Switch\_To\_Address\_Space添加SET\_PDBR，调用作为上下文切换的一部分。 用户将使用usercontext数据结构的pageDir字段来存储过程页面目录表的地址。到目前为止，通过加载Shell用户程序，测试分页系统基本完成。

项目4还需要实现请求分页技术，其具体原理我们在上学期的理论课已经进行学习。具体流程如图6.3所示，为将逻辑地址分解为页码和内部页地址。首先，根据页码查找快表TLB。如果快表命中，请立即发出页面号，并将其与内部页面地址拼接起来以形成物理地址。然后检查访问权限。如果通过，则进程可以访问物理地址。

如果快表未命中，则使用页码查找页表。如果页表命中，则表示访问页已在内存中。可以发送页面号，并与页面中的地址进行拼接以形成物理地址。然后检查访问权限。如果通过进程可以访问物理地址。如果快表未命中，并且页面表显示页面缺失异常，则将处理页面缺失异常。根据页码检查外部页码，找到磁盘的物理地址，并检查内存中是否有可用的页框。如果有，请分配一个。否则

，请根据替换算法选择被淘汰的页面。

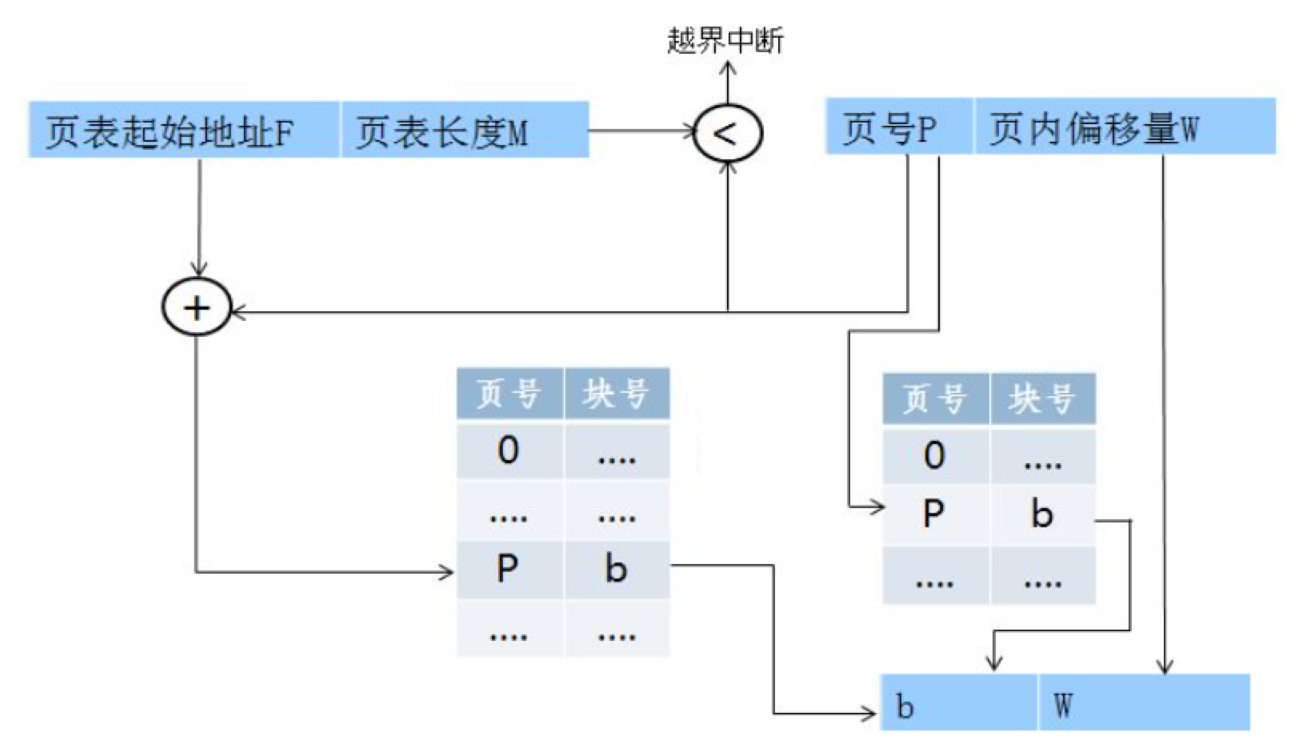


图6.3 分页存储系统的地址转换机制

用户设计的页面错误中断处理程序应该能够识别该页面存在于页面文件中，并将其从磁盘读取到内存中。当用户将页面从磁盘转移到内存中时，该页面所占用的空间需要释放，具体如表6-1所

示。

表6-1 缺页具体情况分析

|  |  |  |
| --- | --- | --- |
| 缺页情况 | 标识 | 相应处理 |
| 堆栈生长到新页 | 超出原来分配一页的限制 | 分配一个新页，进程继续 |
| 此页保存在磁盘上 | 数据标识在papaefile 中存在 | 从papaefile读入需要的页，继续进程 |
| 因为无限地址缺页 | 非法地址访问 | 终止用户进程 |

但是GeekOS原生并不支持相应分配新页处理的具体代码实现，在查阅资料后，通过对Find()\_Page\_To\_Page\_Out()函数进行改良，具体实现见下文运行分析。

# 6.2 Project4项目运行分析

系统在Alloc\_User\_Page()分配用户进程页面函数中对分配过程中的关键信息进行打印，如图6.4所示，标识1打印当前页目录表的入口起始地址，标识2打印当前页表当前起始地址，标识3打印当前所在页表位置，标识4打印物理地址起始地址，标识5打印线性地址。

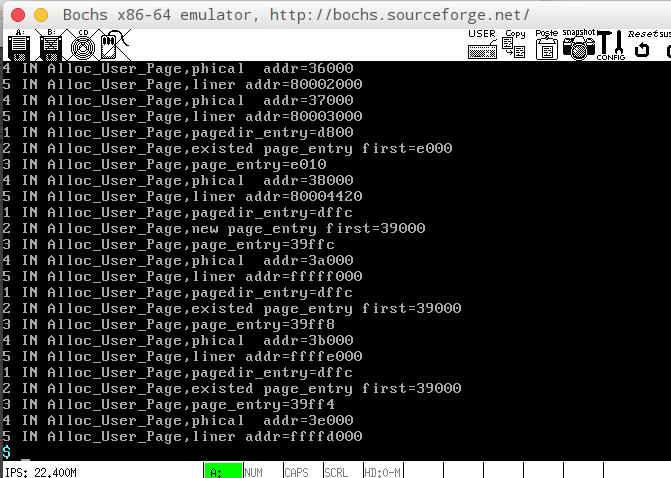


图6.4 Alloc\_User\_Page()分页管理

接着利用rec.c递归用户程序来测试动态堆栈增长的情况，由图6.5所示，当层数depth为6，产生了缺页中断中的堆栈生长到新页情况，如果不设置相应新页处理，程序会在非法地址引发read错误。

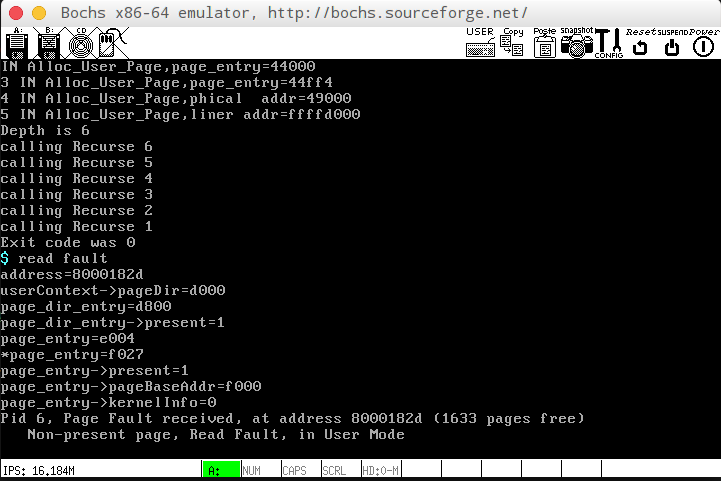


图6.5 rec递归程序测试1

为了处理该缺页中断，系统对Find()\_Page\_To\_Page\_Out()函数进行完善，每次计时器中断到来时，系统将扫描内存并在其页面中记录页面的最后访问时间信息。在每个页面错误处，页面替换算法都从最后一页开始扫描外页，寻找未使用时间超过WORKING\_SET\_TIMEOUT阈值时间的页面，然后替换它。如果该算法找不到这样的旧页面，它将随机选择一个页面进行替换，类似于LRU算法进行页面替换。

完善缺页中断处理后，系统支持有限内存页面分配，如图6.6所示，递归depth=550，每递归1次，系统输出一个“.”，每递归50次，系统输出一行calling Recurse提示，完成在不超过系统内存空间的前提下，完成中规模的动态堆栈增长测试。

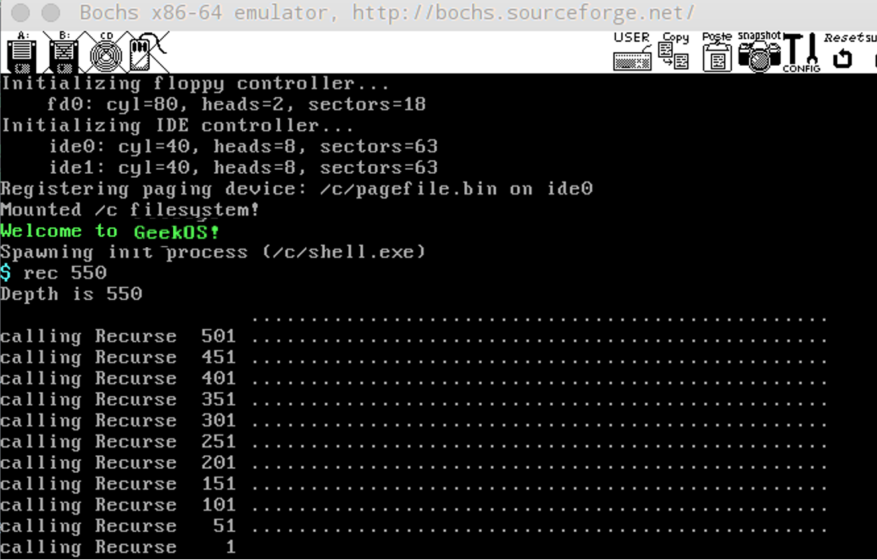


图6.6 rec递归程序测试2

# 第七章 GeekOS设计项目5—GOSFS文件系统

# 7.1 Project5项目原理分析

# 7.1.1 项目5设计目的

了解文件系统的设计原理。掌握操作系统文件系统的具体实现技术。在/src/geeekos/gosfs.c中实现以下函数：

GOSFS\_Fstat()函数:为给定的文件得到元数据。

GOSFS\_Read()函数:从给定文件的当前位置读数据。

GOSFS\_Write()函数:从给定文件的当前位置写数据。

GOSFS\_Seek()函数:在给定文件中定位。

GOSFS\_Close( )函数:关闭给定文件。

GOSFS\_Fstat\_Directory()函数:为一个打开的目录得到元数据。

GOSFS\_ Close\_Directory()函数:关闭给定目录。

GOSFS\_Read\_Entry()函数:从打开的目录表读一个目录项。

GOSFS\_Open()函数:为给定的路径名打开一个文件。

GOSFS\_Create\_Directory()函数:为给定的路径创建一个目录。

GOSFS\_Open\_Directory()函数:为给定的路径打开一个目录。

GOSFS\_Delete()函数:为给定的路径名删除一个文件。

GOSFS\_stat()函数:为给定的路径得到元数据（大小、权限等信息)。

GOSFS\_Sync()函数:对磁盘上的文件系统数据实现同步操作。

GOSFS\_Format()函数:格式化GOSFS 文件系统操作。

GOSFS\_Mount()函数:挂载文件系统操作。

# 7.1.2 项目5设计原理及分析过程

项目5需要实现GOSFS文件系统，由于GeekOS原生支持了PFAT文件系统，本项目在通过磁盘块中的第0块(超级块)的起始标记magic(魔数)内容来验证该系统为GOSFS文件系统，并且PFAT文件系统挂载在存储设备ide0硬盘上，路径通常规定为/c, 而GOSFS文件系统挂载在二级存储设备ide1硬盘上，路径通常规定为/d。

以下为具体实现方式：

GOSFS文件系统通过GOSFS\_Format()函数格式化磁盘。首先调用函数Get\_Num\_Blocks（）以获取磁盘容量，将其转换为磁盘块数，然后计算用于维护可用磁盘块的位图矢量的大小，清空相应位置，然后创建根目录，并使Root Dir指针指向目录，然后将相关数据填充到超级块中。

接着是设计文件系统的磁盘空间分布，其主要涉及GOSFS\_Dir\_Entry结构体，其中保存了目录和文件的相关信息，其中的blockLisas成员数组记录了文件数据块指针，布局如图7.1所示，其采用直接块和间接块（二级间接）的方式进行存储，每级可以存储1024个直接块，因此文件系统最多可以保存1024\*1024个数据块。

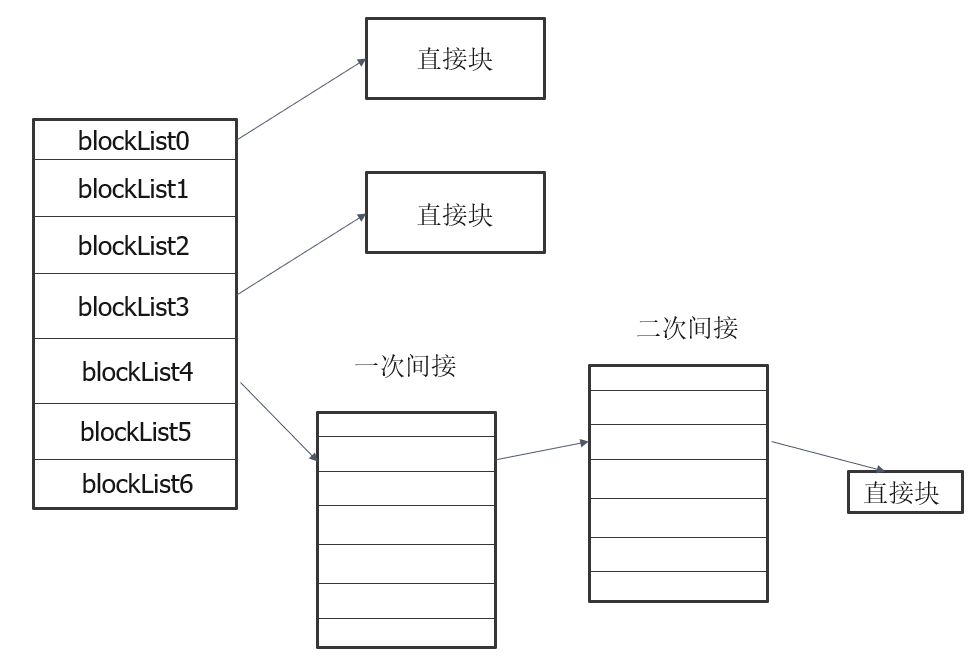


图7.1 blockList数组布局

GosFS文件系统的实现使用类UNIX文件系统的inode形式。其有更大的灵活性和更通用的抽象，以减轻进一步的增强。每个目录分配一个inode，就像文件一样，只是inode中指示的大小表示目录条目的数量，而不是物理大小。目录的内容存储在单独的块中，仅由inode的直接块指针引用。这些块包含一个或多个目录条目以表示目录的内容。通过仅使用直接块指针，目录项的数量限制为240。这对于教学实验系统GeeKOS其实来说已经足够了。

从PFAT系统切换成GOSFS文件系统，其主要操作包含GOSFS\_Format()和GOSFS\_Mount()两个函数。

GOSFS\_Format操作用于文件系统的格式化，进一步加载安装到GeekOS。主要作用是将原始磁盘格式化为GOSFS格式，同时利用魔数标记位检查是否已经格式化GOSGS格式。

GOSFS\_Mount操作负责加载操作，该操作首先初始化内存中的GOSFS\_Superblock，之后初始化Mount\_Point，最后创建一个空目录，通常规定为/d。

测试格式化和挂载GOSFS文件系统的操作为$format ide1 gosfs以及$mount ide1 /d gosfs。

在格式化并挂载完成GOSFS文件系统后，需要调用Mount\_Point()函数，里面需要利用一个Mount\_Point对象（包含一个指向文件系统映像所在块设备的指针）。用户可能需要创建一个辅助的数据结构，存储在挂载点里。这个挂载点是进行以下Open、Delete、Create\_Directory等等文件操作的关键对象。比如调用打开文件函数，我们需要将指向辅助数据结构的指针存入该对象的FSData域中。Mount\_Point对象具体支持文件操作如下：

1. GOSFS\_Open()：在挂载文件系统里打开文件。
2. GOSFS\_Create\_Directory()：在挂载文件系统里面创建目录。
3. GOSFS\_Open\_Directory()：在挂载文件系统里面打开目录。
4. GOSFS\_Stat()：为已经命名的文件恢复文件元数据，比如文件权限。
5. GOSFS\_Sync()：刷新所有存储在内存的，还没有写入磁盘的元数据。
6. GOSFS\_Delete()：在挂载文件系统里删除指定路径的文件或文件夹。

系统创建文件和目录之后，就可以进行文件操作，包括GOSFS\_Read()、GOSFS\_ReadEntry()、GOSFS\_Write()、GOSFS\_Create\_Directory()等，可以通过mkdir、rm、ls、cat等用户子程序在shell进程中进行测试。

gosf.c文件包含了绝大部分与文件系统相关的函数操作，main.c中调用Init\_GOSFS()函数，而其中间接调用Register\_FileSystem(“gosfs”,&s\_gosfsFilesystemOps)，其中gosfsFilesystemOps关联了上文分析的GOSFS\_Format()和GOSFS\_Mount()函数。

GOSFS\_Open()打开文件函数具体过程：

通过调用自定义Find\_InodeByName()子函数检查文件是否存在，如果文件不存在，进行写操作的检查，允许写操作则调用CreateFileINode()函数来创建inode节点，其中进行搜索根节点和间接节点的操作；如果文件存在，则调用vfs.c中内置的Allocate\_File()分配文件对象，其中文件对象参数s\_gosfsFileOps关联了文件相关操作&GOSFS\_Write()，&GOSFS\_Read()，&GOSFS\_Seek()，&GOSFS\_Clone()等，最后返回该文件指针。

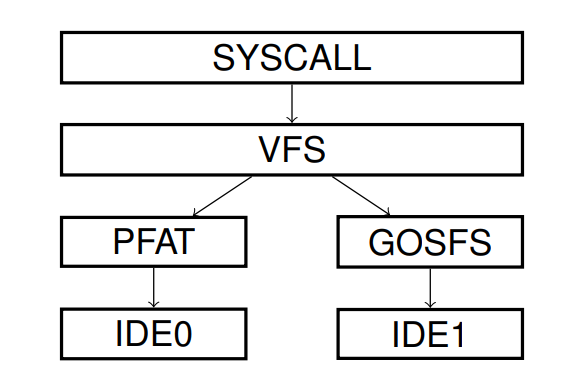
以GOSFS\_Write()写文件操作函数为例分析具体运行过程：

检查写操作是否被运行，接着计算需要写入的数据块startblock以及写入起始地址startblock

Offset，循环计算需要写入的数据块数量，每一次循环调用CreateFileBlock()分配空间，通过GetPhysicalByLogical()函数来计算开始写入的物理地址，最后通过Get\_FS\_Buffer()函数写数据。

GOSFS\_Read()、GOSFS\_Seek()、GOSFS\_Clone()等文件操作同理。

下面对系统如何兼容同时PFAT只读文件系统和GOSFS可读写文件系统进行探讨，其主要是通过VFS虚拟文件系统层进行管理的。虚拟文件系统（VFS）层在较高层次上抽象了文件系统。 诸如PFAT和GosFS之类的具体实现会在内核启动时注册其文件系统驱动程序。 每个用户进程通过C库启动的文件系统操作被捕获到内核中，在该内核中，相应的系统调用将请求转发到VFS。 然后，VFS将请求重定向到使用所谓的虚拟功能表的相应文件系统实现，引用实际实现，具体流程如图7.2所示。

  
图7.2 VFS虚拟文件系统层

VFS为处理文件的进程提供了通用的文件结构。在此文件结构内对于特定的文件系统实现，只有一个指针指向任意数据，其在GOSFS中称为“文件条目”，其中包含文件系统所需的信息使用文件。这种抽象使不同的流程可以在相同的环境下工作，而不会干扰彼此的文件位置，因为每个进程都有自己的文件对象。然后，此文件对象中的任意数据结构将缩小与实际物理文件的距离。

为了进一步加快文件读写速度，GeekOS系统内置原生的Create\_FS\_Buffer\_Cache()可以创建一个新的高速缓冲区，通过包含头部文件<geekos/bufcache.h>来使用。Buffer\_Cache 数据结构表示一个特定文件系统对象的高速缓冲区。通过传递给函数Create\_FS\_Buffer\_Cache()可以创建-一个新的高速缓冲区。但是缓冲区的存取是互斥进行的，为了避免读取“脏”数据，当完成存取数据或者修改缓冲区的数据后，就用Release\_FS\_Buffer函数释放缓冲区。同一时刻只有一个线程能使用缓冲区。如果遇到同时请求使用缓冲区的情况，就第二个请求的线程需要被阻塞，只到缓冲区被释放为止。

# 7.2 Project5项目运行分析

Bochs模拟器启动GeekOS系统后，通过ls指令进行测试，列出/c路径文件系统下的所有文件，如图7.3所示，显示PFAT只读文件下挂载的所有文件。

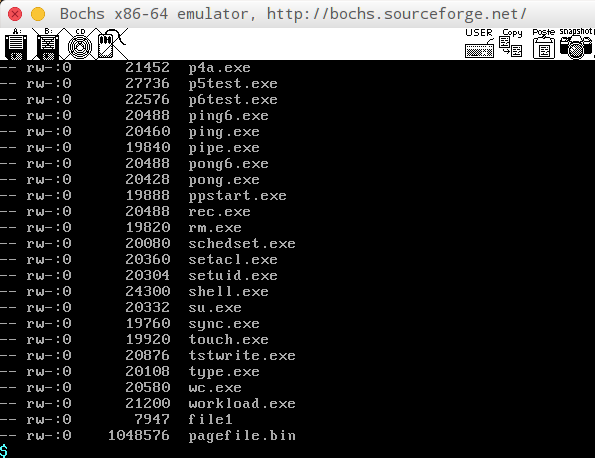


图7.3 ls指令测试

接着，通过format和mount指令格式化和加载GOSFS文件系统，如图7.4和图7.5所示，成功格式化和挂载GOSFS文件系统。

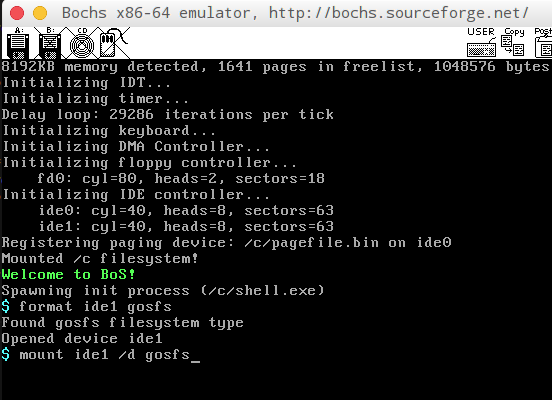
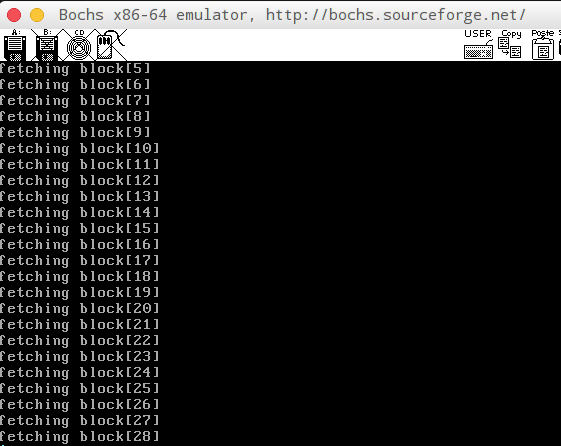
 

图7.4 format指令测试 图7.5 mount指令测试

系统还可以通过mkdir指令创建文件夹，touch指令创建文件，rm指令删除文件夹或文件，如图7.6和图7.7所示。

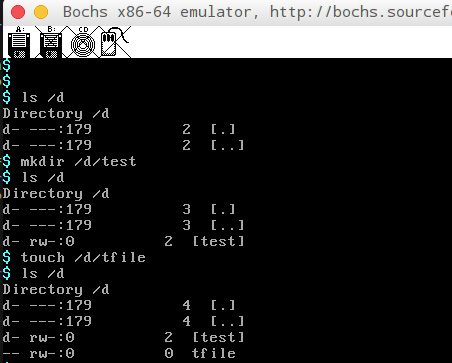
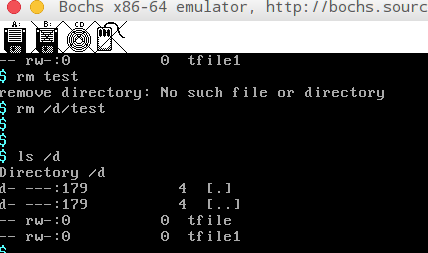
 

图7.6 touch指令测试 图7.7 rm指令测试

系统可以通过cp指令进行文件拷贝操作，如图7.8所示。

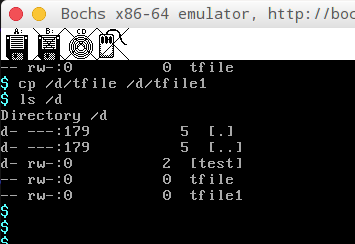


图7.8 cp指令测试

为了测试一个文件系统常见的大部分操作，系统调用p5test子程序来进行批量测试，包括基本文件创建、10k文件读写、删除空/非空文件夹、文件指针定位等，pass表示通过，fail表示失败，测试结果如图7.9所示，通过27条测试，得分为85。

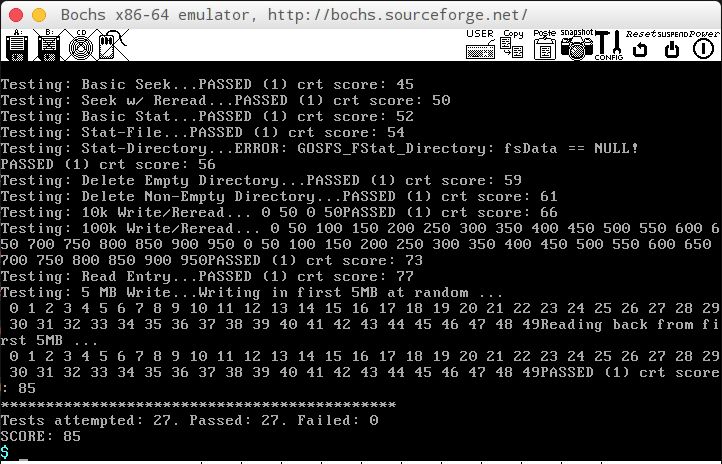


图7.9 p5test 文件批量测试

# 第八章 GeekOS设计项目6—进程通信

# 8.1 Project6项目原理分析

# 8.1.1 项目6设计目的

要求使用匿名半双工管道实现进程间通信，为文件系统增加访问控制列表。

# 8.1.2 项目6设计原理及分析过程

本项目旨在实现“进程通信”，下面针对本项目实现的“消息队列”进程通信以及“管道操作”进行原理分析，二者的实现方式类似。

1. 消息队列Message Queue

系统通过引入多个读取器和写入器的消息队列，在GeekOS中，消息被视为单个实体，它可以作为一个实体发送或接收。发送消息后，将无法对其进行修改。如果一个进程间通信进程为接收到的消息提供了太小的缓冲区，消息被截断为提供的长度，消息的切除部分将丢失。如图8.1所示，系统中的消息队列是可变长度的先进先出（FIFO）队列。创建消息队列时，调用进程可以定义其长度，最大尺寸一条消息的最大长度限制为4KB。

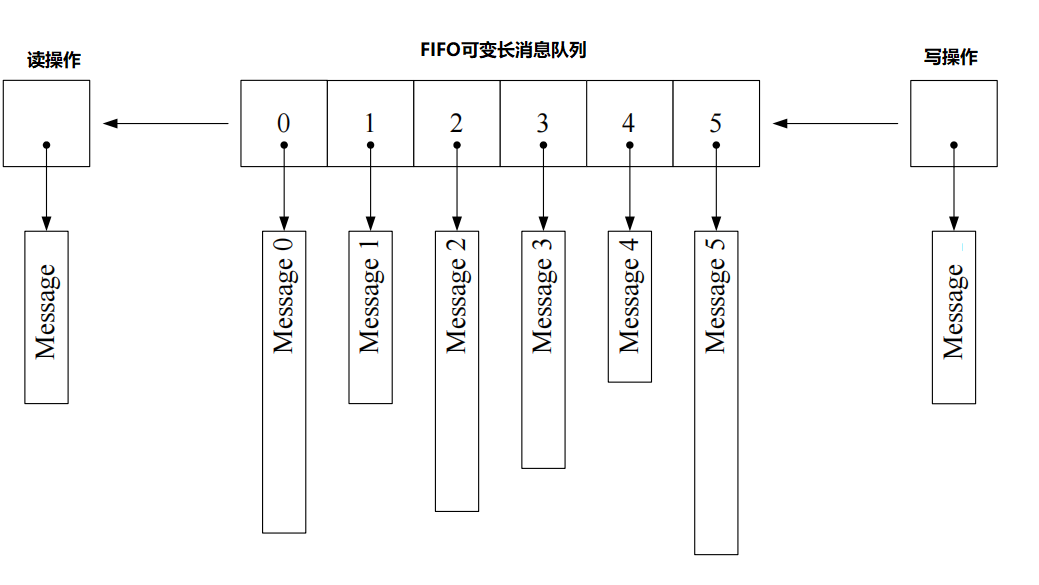


图8.1 可变长消息队列Message工作示意图

成功创建消息队列后，该进程可以将消息发送到该队列，无论是否从队列中读取消息的不是接收进程。队列达到最大值长度时，发送调用将阻塞发送过程，直到队列中获取消息为止。如果接收进程要从空队列中读取消息，则接收呼叫将阻塞直到消息进入队列。将多个进程附加到消息队列后，内核依次将消息分发到接收进程。只要消息队列中有消息，就可以保存工作状态，无需其他进程。

1. 管道操作Pipe Operation

项目系统提供匿名管道，不提供命名管道，进程之一可能只向管道写入，而对等进程可能仅从管道读取。如图8.2所示，管道基本上是缓冲区以及管理数据，每个进程都拥有一个文件结构，该文件结构引用了内核中的通用管道结构。Pipe管道结构保存运行连接的环形缓冲区所需的信息。发送过程可以将数据写入管道，直到缓冲区已满为止，进一步的写调用将阻止发送过程。

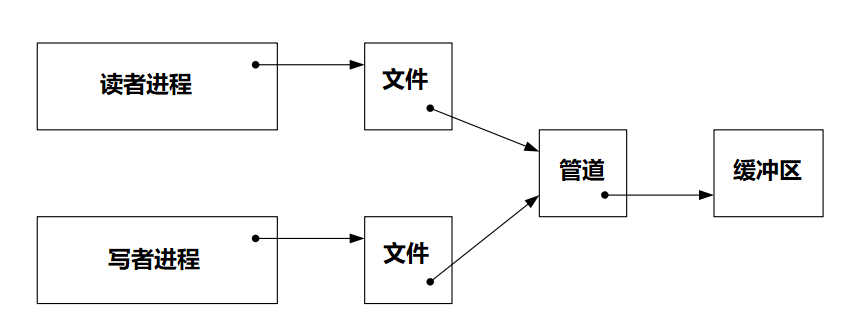


图8.2 系统管道接口

当发送进程关闭管道一侧时，接收进程仍可以读取剩余数据。如果管道为空，则接收进程从系统调用中获取文件错误响应。

在具体实现中，通过Message\_Queue\_Create()函数创建发送者和接受者消息进程，fsend和frecv两个子程序中分别调用Message\_Queue\_Send()和Message\_Queue\_Receive()函数来分别实现从缓冲区中发送和接收信息。同时，定义参数缓冲区的大小，如果缓冲区大小小于实际通信内容，则接收方会出现截断情况。

系统在main.c中调用Init\_MQ()初始化，其调用了clear\_msg\_queue\_list()初始化消息队列。MQ\_Create()创建消息队列函数中，如果MQ存在则返回已经存在的ID，否则创建MQ。MS\_

Destory()销毁消息队列函数中，是通过ref引用计数来控制的，销毁该进程的消息队列则将ref减1，如果ref计数为0，则完全销毁MQ。MQ\_Send()和MQ\_Receive()是从消息队列中发送和读取数据函数，其数据通过一个缓冲区进行维护。

# 8.2 Project6项目运行分析

Bochs模拟器启动GeekOS系统后，通过ls /c | more指令进行管道操作测试，ls /c作为发送者进程，more作为接收者进程，ls /c的内容通过管道发送到more程序中，逐页逐行加载打印出来，如图8.3所示。

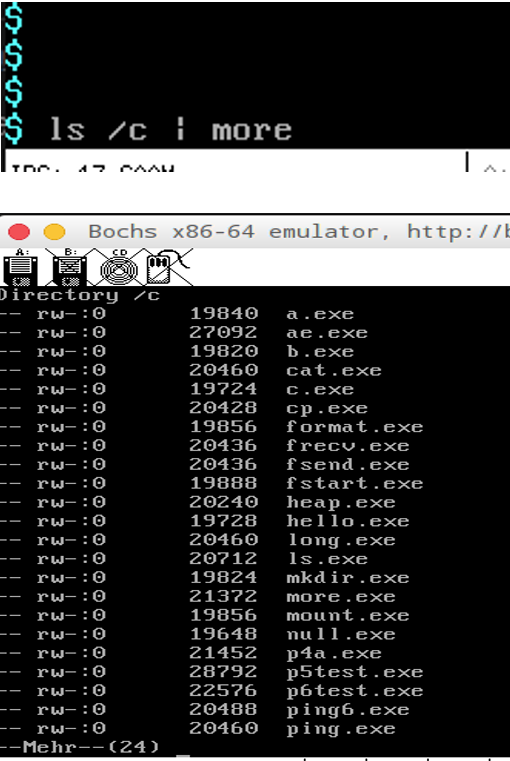


图8.3 ls /c | more 指令测试

由于在shell.c用户程序中实现了“|”管道操作符，将ls /c的内容通过消息队列的方式发送到缓冲区，more子程序从缓冲区读取内容，进行逐行逐页打印，用户可以通过space或者enter键控制内容的逐行显示。

接着，还可以通过fsend和frecv 指令来进行进程通信的测试，fsend作为发送方输入了10条message到缓冲区，frecv从缓冲区进行读取并显示，过程如图8.4和图8.5所示。

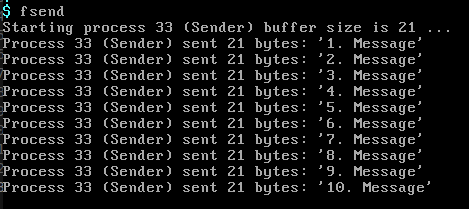


图8.4 fsend 指令测试

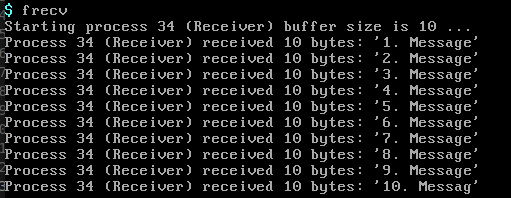


图8.5 frecv 指令测试

# 第九章 课程设计心得体会总结

# 参考文献

1. 黄廷辉,王宇英.计算机操作系统实践教程[M].北京：清华大学出版社，2007.
2. David H, Jeffrey K, Hollingsworth. Running on the bare metal with GeekOS [EB/OL].

<https://doi.org/10.1145/1028174.971411>.

1. GeekOS web site [EB/OL].http://geekos.sourceforge.net. 2005,12.
2. 徐虹.操作系统实验指导—基于Linux内核[M].北京:清华大学出版社,2005.
3. 顾宝根等.操作系统实验教程—核心技术与编程实例[M].北京:科学出版社，2003.
4. B.Atkin and E.G.Sirer. PortOS: An Educational Operating System for the Post-PC Environment.

In Proceedings of the ACM Technical Symposium on Computer Science Education, 2002.

1. R.Gove. CMSC 412 GeekOS Project5 File System[EB/OL]. <https://www.cs.umd.edu/~hollings/cs412>

/s10/project5/proj5\_section.pdf.