



桂林电子科技大学  
GUILIN UNIVERSITY OF ELECTRONIC TECHNOLOGY

## 操作系统课程设计报告

题	目	: Geek0S 操作系统的研究与实现
课	号	: 2222901、2222906
组	长	: 2000201916 李嘉佳
成	员	: 2000300202 何冬梅
		: 2000300601 陈冰鑫
		: 2000500927 吴河山
学	院	: 计算机与信息安全学院
专	业	: 计算机科学与技术
指	导	教 师 : 管军霖、王慧娇

2023 年 5 月 10 日

## 操作系统课设计报告评分标准及评分

评分项目	分值	考核点	实际得分	
文献查阅及归纳综合能力	25	合理使用各种检索工具，查阅相关文献资料，并进行归纳总结。能为制定方案提供依据，能为实施过程解决问题提供参考。		
团队协作能力	25	学生对课程设计的学习态度是否认真，团队分工是否合理及协作情况		
设计方案	25	利用所学知识、对实际工程问题进行了分析归纳，设计方案，论证充分合理，逻辑清晰，数据结构及算法设计正确。		
实验结果及报告规范性	25	对所实现的系统进行实验验证，产生相应的结果数据，对数据及结果进行充分的分析。条理清楚，文理通顺，格式规范，用语及图表符号符合技术规范。		
总 评				
小组成员及分工				
学号	姓名	完成的工作	小 组 自 评	工作量 (%)
2000201916	李嘉佳	项目 5 的设计与实现	100	25
2000300202	何冬梅	项目 0、1、2 设计与实现	90	25
2000300601	陈冰鑫	项目 3 设计与实现	95	25
2000500927	吴河山	项目 4 设计与实现	90	25

## 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-project5 六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。

本课程设计与操作系统理论基础知识相结合，实现了操作系统基本的 I/O 操作，用户态和内核态进程相关的生命周期，多级反馈队列（MLF）和轮转（RR）调度算法以及信号量的相关操作，分段分页内存管理方式，以及基于虚拟文件系统的 GOSFS 文件系统，其中包括部分常用文件操作命令功能。

本课程设计实现了 project0-project5 的基本需求，project0-project4 实现了包括 I/O 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）。

**关键词：** GeekOS、调度算法、内存管理、进程通信、文件系统 GOSFS

## 目 录

1 引言.....	3
1.1 课程设计开发背景.....	3
1.1.1 GeeKOS 课程设计任务概述.....	3
1.1.2 GeeKOS 实验环境.....	3
1.2 课程设计具体开发步骤.....	4
2 GeekOS 设计项目 0—GeekOS 系统环境调试及编译 .....	6
2.1 Project0 项目原理分析 .....	6
2.1.1 项目 0 设计目的 .....	6
2.2.2 项目 0 设计原理及分析过程 .....	6
2.2 Project0 项目运行分析 .....	8
3 GeekOS 设计项目 1—内核级线程设计及实现.....	10
3.1 Project1 项目原理分析 .....	10
3.1.1 项目 1 设计目的 .....	10
3.1.2 项目 1 设计原理及分析过程 .....	10
3.2 Project1 项目运行分析 .....	12
4 GeekOS 设计项目 2—用户级进程的动态创建与执行.....	15
4.1 Project2 项目原理分析 .....	15
4.1.1 项目 2 设计目的 .....	15
4.1.2 项目 2 设计原理及分析过程 .....	15
4.2 Project2 项目运行分析 .....	18
5 GeekOS 设计项目 3—进程调度算法与信号量功能.....	19
5.1 Project3 项目原理分析 .....	19
5.1.1 项目 3 设计目的 .....	19
5.1.2 项目 3 设计原理及分析过程 .....	20
5.2 Project3 项目运行分析 .....	31
6 GeekOS 设计项目 4—分页存储管理机制.....	35
6.1 Project4 项目原理分析 .....	35
6.1.1 项目 4 设计目的 .....	35
6.1.2 项目 4 设计原理及分析过程 .....	35
6.2 Project4 项目运行分析 .....	41

7 GeekOS 设计项目 5—GOSFS 文件系统 .....	44
7.1 Project5 项目原理分析 .....	44
7.1.1 项目 5 设计目的 .....	44
7.1.2 项目 5 设计原理及分析过程 .....	44
7.2 Project5 项目运行分析 .....	50
8 课程设计心得体会总结.....	54
参考文献.....	55

# 1 引言

操作系统是管理计算机硬件与软件资源的计算机程序。操作系统需要处理如管理与配置内存、决定系统资源供需的优先次序、控制输入设备与输出设备、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。

```
romimage: file=/usr/share/bochs/BIOS-bochs-latest
vgaromimage: file=/usr/share/vgabios/vgabios.bin
megs: 8
boot: a
floppya: 1_44=../fd.img, status=inserted
ata0-master: type=disk, path="diskc.img", mode=flat, cylinders=40, heads=8, spt=64
```

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 VMware 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 GeekOS 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。



## 2 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、numTicks 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。

(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。

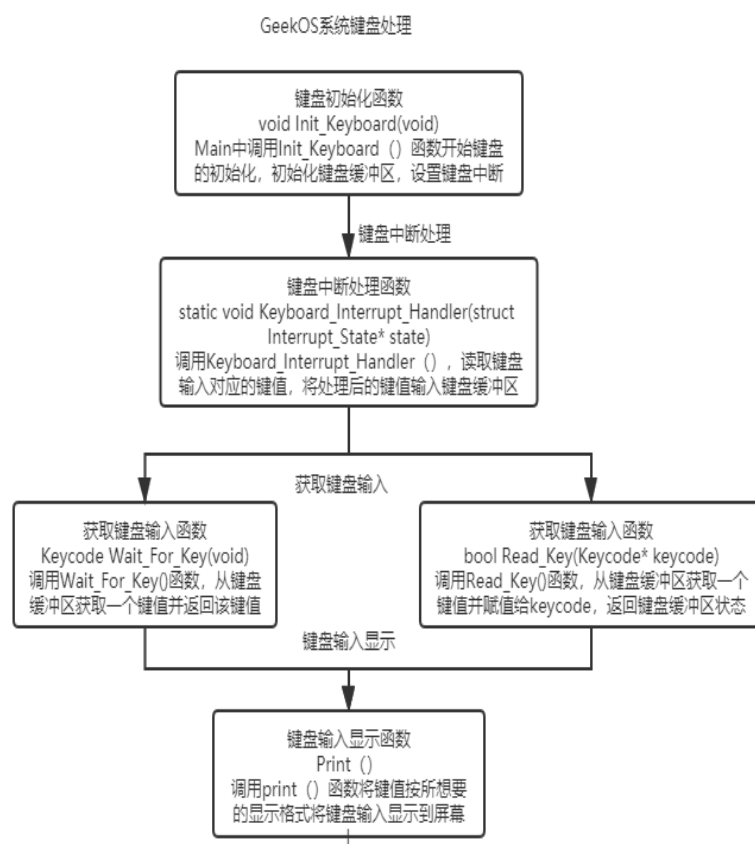


图 2.2 GeekOS 系统键盘处理过程

## 2.2 Project0 项目运行分析

在 main.c 设计 project0()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)`。

project0()函数如下：

```
src/geekos/paging.c | void Init_VM(struct Boot_Info
*bootInfo)

void project0()
{
    Print("To Exit hit Ctrl + d.\n");
    Keycode keycode;
    while(1)
    {
        if( Read_Key(&keycode) ) //读取键盘按键状态
        {
            if(!( (keycode & KEY_SPECIAL_FLAG) || (keycode & KEY_RELEASE_F
LAG))) )
                //只处理非特殊按键的按下事件
            {
                //低 8 位为 Ascii 码,KEY_CTRL_FLAG 0x4000  0x4064 & 0xff = 0x0064
                if( keycode == (KEY_CTRL_FLAG+'d'))
                    //按下 Ctrl 键
                {
                    Print("\n-----BYE!-----\n");
                    Exit(1);
                }
                else
                {
```

```
Print("%c", (keycode == '\r') ? '\n' : (keycode & 0xff));  
  
    }  
  
}  
  
}  
  
}
```

程序运行结果:

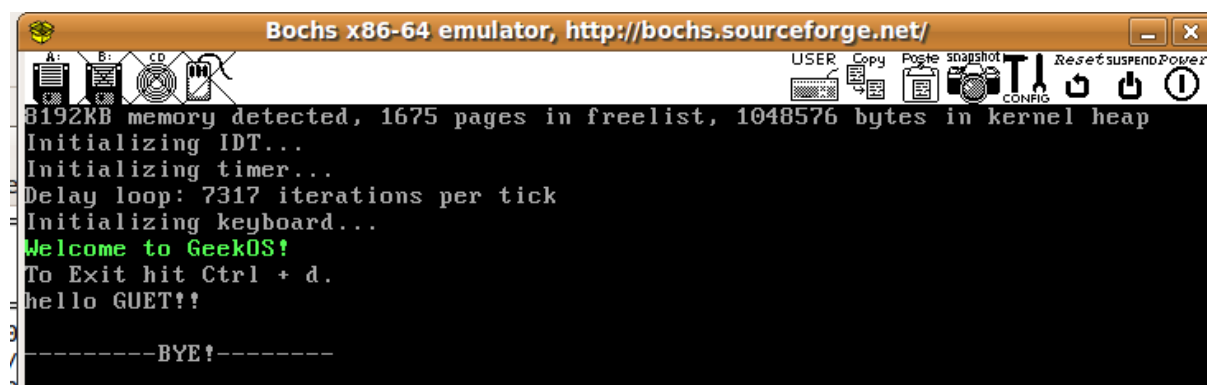


图 2.3 程序运行结果

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 `keycode == (KEY_CTRL_FLAG+'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。

### 3 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,ELF 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。

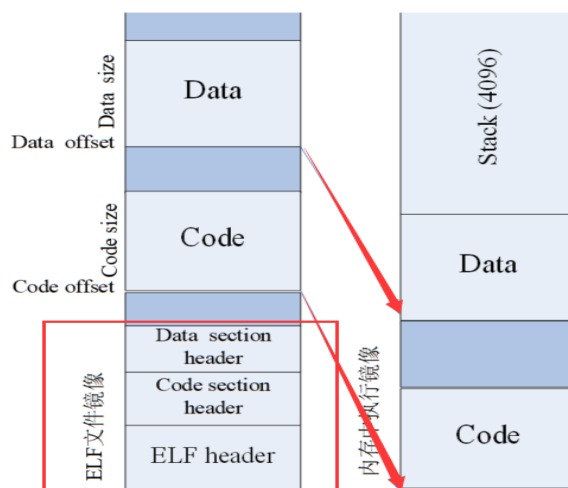


图 3.1 ELF 文件与内存镜像映射

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 `elf.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，执行 bochs 命令，启动运行后得到如图 3.2 所示结果。

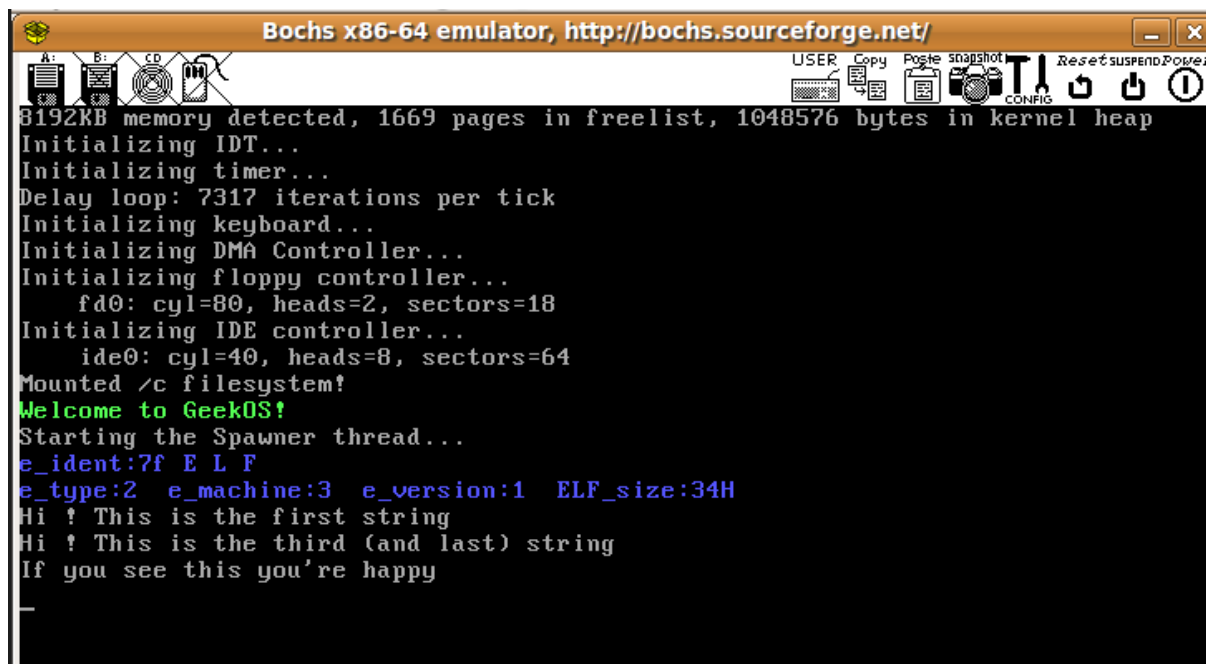


图 3.2 project1 运行结果

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。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。

## 4 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。

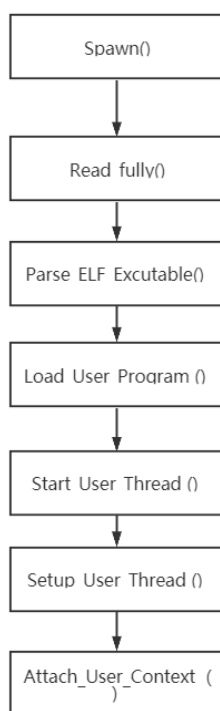


图 4.1 用户进程创建流程

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。

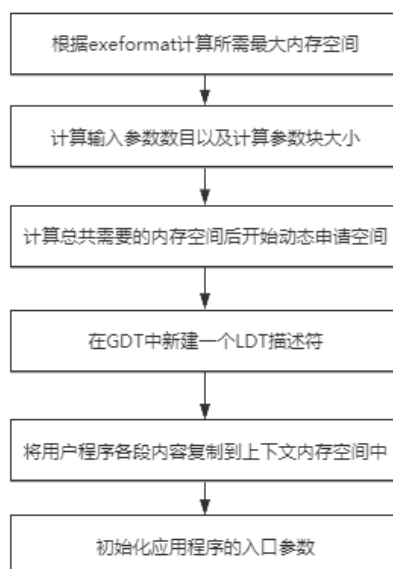


图 4.2 Load\_User\_Program()实现流程

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。

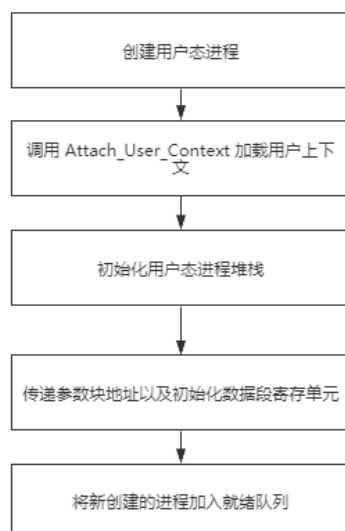


图 4.3 得到 user\_context 后创建用户态进程

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。

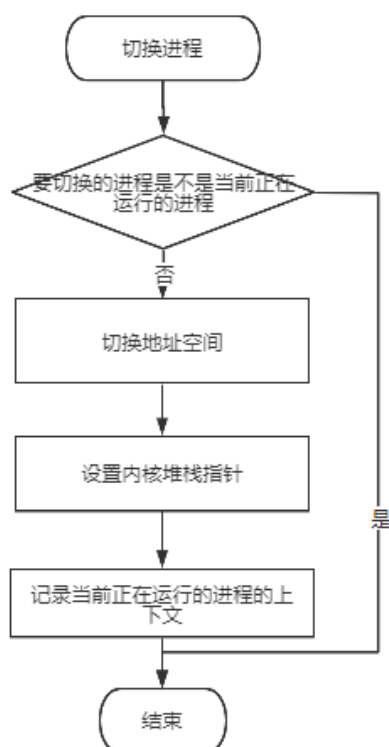


图 4.4 切换当前运行的进程

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。

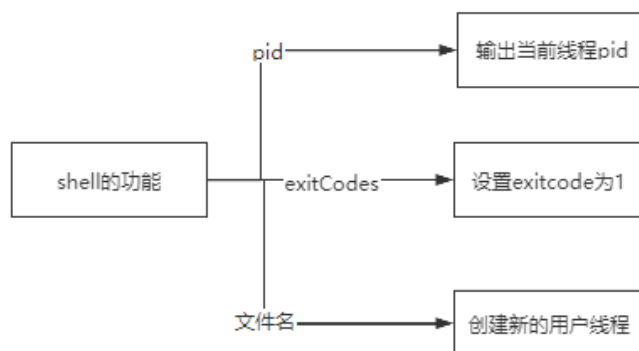


图 4.5 shell 文件执行的功能

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

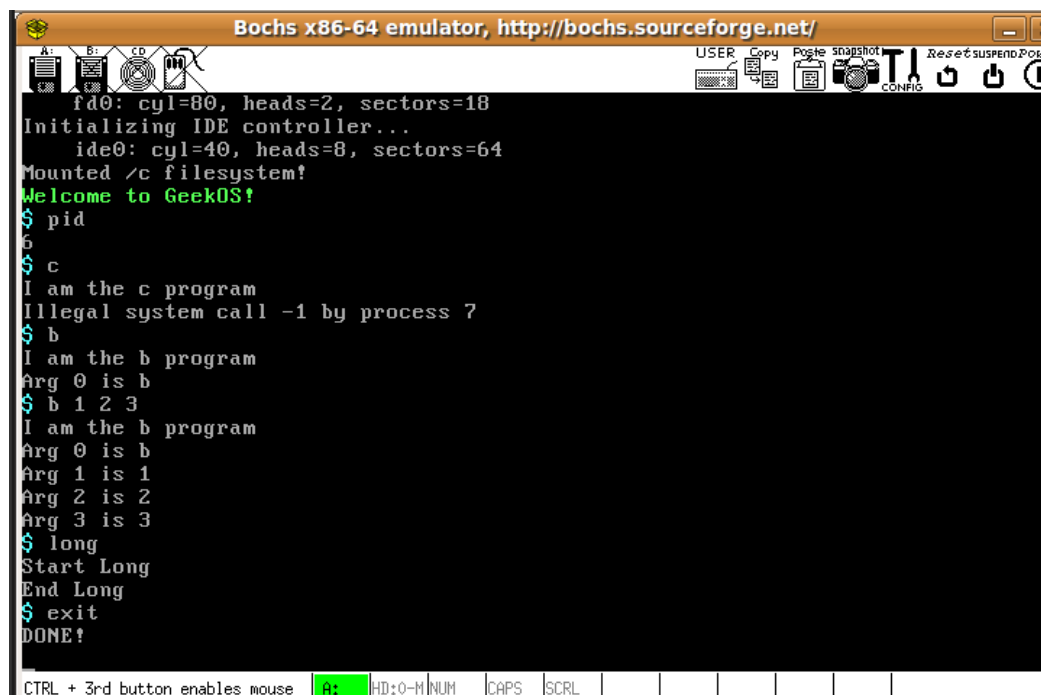
项目 2 中“`user.c`”文件中的函数 `Spawn("/c/shell.exe", "/c/shell.exe", &pThread)` 用于生成一个用户及进程，需要传入可执行文件地址以及指定的进程指针。

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。



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
fd0: cyl=80, heads=2, sectors=18
Initializing IDE controller...
ide0: cyl=40, heads=8, sectors=64
Mounted /c filesystem!
Welcome to GeekOS!
$ pid
6
$ c
I am the c program
Illegal system call -1 by process 7
$ b
I am the b program
Arg 0 is b
$ b 1 2 3
I am the b program
Arg 0 is b
Arg 1 is 1
Arg 2 is 2
Arg 3 is 3
$ long
Start Long
End Long
$ exit
DONE!
CTRL + 3rd button enables mouse  A: HD:0-H NUM CAPS SCRL
```

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。

## 5 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。

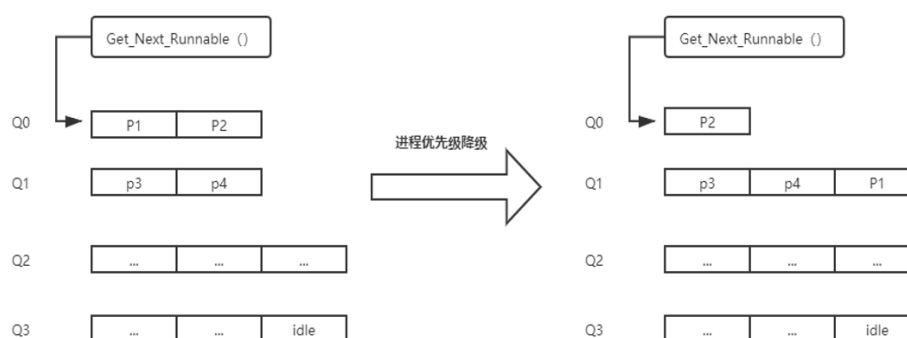


图 5.1 四级反馈队列调度优先级降级

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。

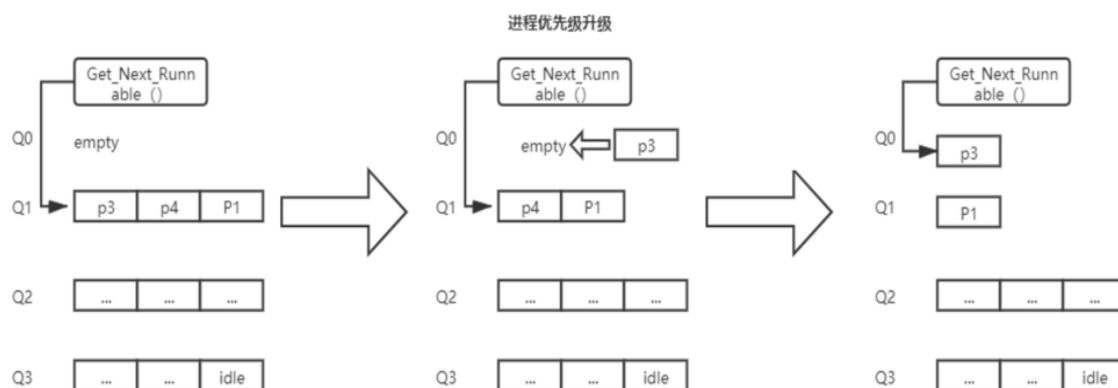


图 5.2 四级反馈队列调度优先级升级

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。

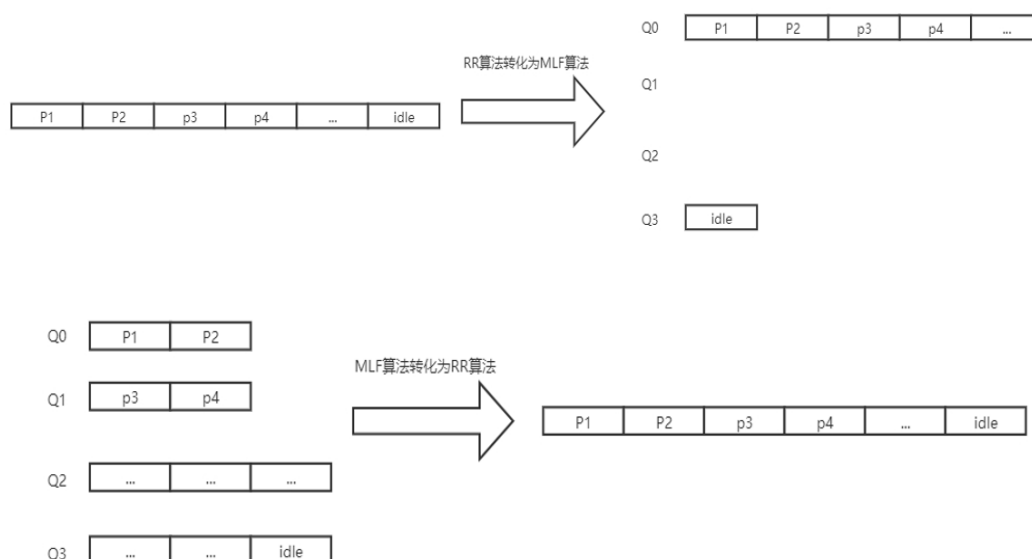


图 5.3 多级反馈队列与分时调度进程队列的转换



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。

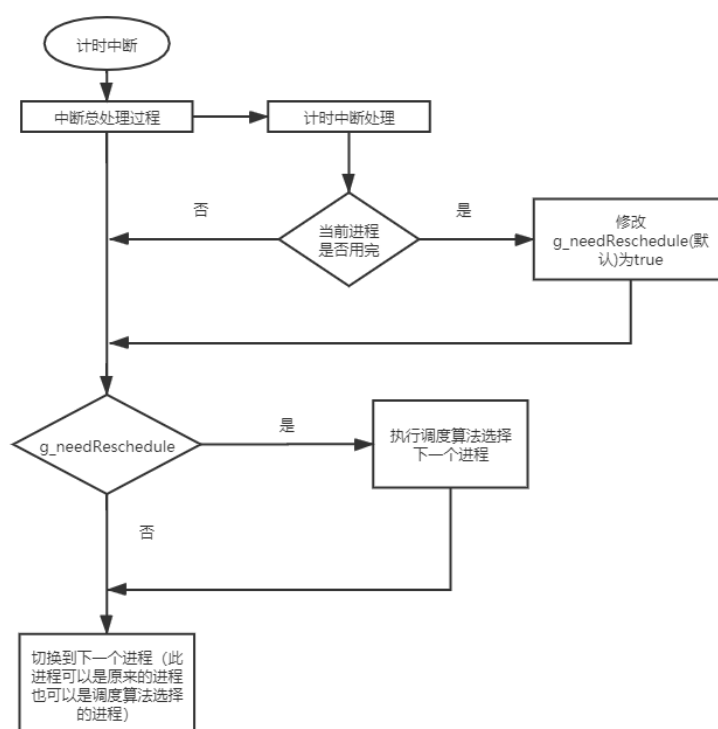


图 5.4 进程调度处理过程

具体的代码如下：

```

src/geekos/syscall.c|static int
Sys_SetSchedulingPolicy(struct Interrupt_State *state)

/**
 * 设置调度策略
 * Set the scheduling policy.
 * Params:

```

```

*   state->ebx - policy,
*   state->ecx - number of ticks in quantum
* Returns: 0 if successful, -1 otherwise
*/

static int Sys_SetSchedulingPolicy(struct Interrupt_State *state)
{
    // TODO("SetSchedulingPolicy system call");
    /* 如果输入的优先级调度方法参数无效(非 0 或 1)则返回错误 */
    if (state->ebx != ROUND_ROBIN && state->ebx != MULTILEVEL_FEEDBACK)
    {
        Print("Error! Scheduling Policy should be RR or MLF\n");
        return -1;
    }
    /* 如果输入的时间片参数不在[1, 100]之间则返回错误 */
    if (state->ecx < 1 || state->ecx > 100)
    {
        Print("Error! Quantum should be in the range of [1, 100]\n");
        return -1;
    }
    int res = Chang_Scheduling_Policy(state->ebx, state->ecx);
    return res;
}

```

```
src/geekos/kthread.c |
```

```

/**
* 切换调度策略
*/

int Chang_Scheduling_Policy(int policy, int quantum)
{
    /* 如果调度策略不同，则修改线程队列 */

```

```
if (policy != g_schedulingPolicy)
{
/* MLF -> RR */
if (policy == ROUND_ROBIN)
{
/* 从最后一个线程队列(此处为 Q3)开始将其中的所有线程依次移动到前一个队
列,
直到所有线程都移动到 Q0 队列 */
int i;
for (i = MAX_QUEUE_LEVEL - 1; i > 0; i--)
Append_Thread_Queue(&s_runQueue[i - 1], &s_runQueue[i]);
}
/* RR -> MLF */
else
{
/* 判断 Idle(空闲)线程是否在 Q0 队列 */
if (Is_Member_Of_Thread_Queue(&s_runQueue[0], IdleThread))
{
/* 将 Idle 线程从 Q0 队列移出 */
Remove_Thread(&s_runQueue[0], IdleThread);
/* 将 Idle 线程加入到最后一个队列(此处为 Q3) */
Enqueue_Thread(&s_runQueue[MAX_QUEUE_LEVEL - 1], IdleThread);
}
}
/* 将全局变量设置为对应的输入值 */
g_schedulingPolicy = policy;
Print("g_schedulingPolicy = %d\n", g_schedulingPolicy);
}
g_Quantum = quantum;
```

```
Print("g_Quantum = %d\n", g_Quantum);
```

```
return 0;
```

```
}
```

```
/*
```

\*将给定线程添加到运行队列中，以便可以调度它。 必须在禁用中断的情况下调用！

```
*/
```

```
void Make_Runnable(struct Kernel_Thread *kthread)
```

```
{
```

```
KASSERT(!Interrupts_Enabled());
```

```
{
```

```
int currentQ = kthread->currentReadyQueue;
```

```
/* ----- 根据当前调度策略安排线程应该进入的队列 ----- */
```

```
if (g_schedulingPolicy == ROUND_ROBIN)
```

```
currentQ = 0;
```

```
else if (kthread == IdleThread)
```

```
currentQ = MAX_QUEUE_LEVEL - 1;
```

```
KASSERT(currentQ >= 0 && currentQ < MAX_QUEUE_LEVEL);
```

```
kthread->blocked = false;
```

```
Enqueue_Thread(&s_runQueue[currentQ], kthread);
```

```
}
```

```
}
```

```
/*
```

\* 从运行队列中获取下一个可运行的线程。

\* 这是调度程序。

```
*/
```

```
struct Kernel_Thread *Get_Next_Runnable(void)
```

```
{
```

```
/* Find the best thread from the highest-priority run queue */
```

```
// TODO("Find a runnable thread from run queues");
KASSERT(g_schedulingPolicy == ROUND_ROBIN ||
g_schedulingPolicy == MULTILEVEL_FEEDBACK);
/* 查找下一个被调度的线程 */
struct Kernel_Thread *best = NULL;
if (g_schedulingPolicy == ROUND_ROBIN)
{
/* 轮询调度策略：只需要从 Q0 队列找优先级最高的线程取出 */
best = Find_Best(&s_runQueue[0]);
/* 如果找到了符合条件的线程则将其从队列中移出 */
if (best != NULL)
Remove_Thread(&s_runQueue[0], best);
}
else
{
int i;
for (i = 0; i < MAX_QUEUE_LEVEL; i++)
{
/* 从最高层队列依次向下查找本层队列中最靠近队首的线程，
如果找到则不再向下继续查找 */
best = Get_Front_Of_Thread_Queue(&s_runQueue[i]);
if (best != NULL)
{
Remove_Thread(&s_runQueue[i], best);
break;
}
}
}

/* 如果当前没有可执行进程，则至少应该找到 Idle 线程 */
```

```
KASSERT(best != NULL);

//Print("Scheduling %x\n", best);

return best;

}
```

## 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。

具体的代码如下：

```
src/geekos/syscall.c
```

```
/*
 * Create a semaphore.//创建信号量
 * Params:
 *   state->ebx - user address of name of semaphore
 *   state->ecx - length of semaphore name
 *   state->edx - initial semaphore count
 * Returns: the global semaphore id
```

```
*/

static int Sys_CreateSemaphore(struct Interrupt_State *state)
{
    // TODO("CreateSemaphore system call");

    int res;

    ulong_t userAddr = state->ebx; //信号量名字字符串所在用户空间地址
    ulong_t nameLen = state->ecx; //信号量名长度
    ulong_t initCount = state->edx; //信号量初始值

    /* 如果传入参数不正确则返回错误 */
    if (nameLen <= 0 || initCount < 0 || nameLen > MAX_SEMAPHORE_NAME)
    {
        Print("Error! Semaphore Params incorrect\n");
        return EINVAL;
    }

    char *semName = NULL;

    /* 从用户空间拷贝信号量名字字符串到内核空间 */
    res = Copy_User_String(userAddr, nameLen, MAX_SEMAPHORE_NAME,
&semName);

    if (res != 0)
    {
        Print("Error! Cannot copy string from user space\n");
        return res;
    }

    /* 判断信号量名的合法性(中间是否含有'\0'字符) */
    if (strlen(semName, MAX_SEMAPHORE_NAME) != nameLen)
    {
        Print("Error! Semaphore Name is Invalid\n");
        return EINVAL;
    }
}
```

```
/* 创建一个信号量 */
res = Create_Semaphore(semName, nameLen, initCount);
return res;
}
/*
* Acquire a semaphore.//请求一个信号量
* Assume that the process has permission to access the semaphore,
* the call will block until the semaphore count is >= 0.
* Params:
*   state->ebx - the semaphore id
*
* Returns: 0 if successful, error code (< 0) if unsuccessful
*/
static int Sys_P(struct Interrupt_State *state)
{
// TODO("P (semaphore acquire) system call");
int sid = state->ebx;
if (sid <= 0)
{
Print("Error! Semaphore ID is Invalid\n");
return EINVAL;
}
return P(sid);
}
/*
* Release a semaphore.//释放一个信号量
* Params:
*   state->ebx - the semaphore id
*
```



```
* Returns: 0 if successful, error code (< 0) if unsuccessful
*/

static int Sys_V(struct Interrupt_State *state)
{
    // TODO("V (semaphore release) system call");
    int sid = state->ebx;
    if (sid <= 0)
    {
        Print("Error! Semaphore ID is Invalid\n");
        return EINVAL;
    }
    return V(sid);
}

/*
* Destroy a semaphore.//删除信号量
* Params:
*     state->ebx - the semaphore id
*
* Returns: 0 if successful, error code (< 0) if unsuccessful
*/

static int Sys_DestroySemaphore(struct Interrupt_State *state)
{
    // TODO("DestroySemaphore system call");
    int sid = state->ebx;
    if (sid <= 0)
    {
        Print("Error! Semaphore ID is Invalid\n");
        return EINVAL;
    }
}
```

```
return Destroy_Semaphore(state->ebx);
}
/*
 * Global table of system call handler functions.//系统调用处理程序函数的全局表
 */
const Syscall g_syscallTable[] = {
    Sys_Null,
    Sys_Exit,
    Sys_PrintString,
    Sys_GetKey,
    Sys_SetAttr,
    Sys_GetCursor,
    Sys_PutCursor,
    Sys_Spawn,
    Sys_Wait,
    Sys_GetPID,
    /* Scheduling and semaphore system calls. */
    Sys_SetSchedulingPolicy,
    Sys_GetTimeOfDay,
    Sys_CreateSemaphore,
    Sys_P,
    Sys_V,
    Sys_DestroySemaphore,
};
```

## 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

[illegible]

图 5.5 schedtest 测试用例使用两个调度算法结果

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

[illegible]

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。

```
Welcome to GeekOS!
$ workload rr 1
g_Quantum = 1
***** Start Workload Generator *****
Process Long has been created with ID = 8
Process Ping has been created with ID = 9
Process Long is done at time: 30
Process Pong has been created with ID = 10
Process Pong is done at time: 0
Process Ping is done at time: 5

Tests Completed at 34
$ workload mlf 1
g_schedulingPolicy = 1
g_Quantum = 1
***** Start Workload Generator *****
Process Long has been created with ID = 12
Process Ping has been created with ID = 13
Process Pong has been created with ID = 14
Process Pong is done at time: 0
Process Ping is done at time: 3
Process Long is done at time: 30

Tests Completed at 32
```

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。

```
$ semtest
Create_Semaphore()...
Create_Semaphore() returned 1
P()...
P() returned 0
P()...
P() returned 0
V()...
V() returned 0
Destroy_Semaphore()...
Destroy_Semaphore() returned 0
```

图 5.8 semtest 测试用例运行结果

```
$ semtest1
Semtest1 begins
p3 created
Produced 0
Consumed 0
Produced 1
Consumed 1
Produced 2
Consumed 2
Produced 3
Consumed 3
Produced 4
Consumed 4
p3 executed
```

图 5.9 semtest1 测试用例运行结果

```
$ semtest2
Error! Semaphore ID is Invalid
+ Identified unauthorized call
Error! Semaphore ID is Invalid
+ Identified invalid SID
Create_Semaphore() called
Create_Semaphore() returned 1
P() called
P() returned 0
V() called
V() returned 0
Destroy_Semaphore() called
Destroy_Semaphore() returned 0
Error! Cannot Find Semaphore with SID=1
+ Removed authority after finish
```

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。

## 6 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 之前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。

<pre>typedef struct {     uint_t present:1;     uint_t flags:4;     uint_t accessed:1;     uint_t reserved:1;     uint_t largePages:1;     uint_t globalPage:1;     uint_t kernelInfo:3;     uint_t pageTableBaseAddr:20; } pde_t;</pre>	<pre>typedef struct {     uint_t present:1;     uint_t flags:4;     uint_t accessed:1;     uint_t dirty:1;     uint_t pteAttribute:1;     uint_t globalPage:1;     uint_t kernelInfo:3;     uint_t pageBaseAddr:20; } pte_t;</pre>
--	--

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。以下为 Init\_VM() 函数的定义：

```
src/geekos/paging.c | void Init_VM(struct Boot_Info
*bootInfo)

//通过为内核和物理内存构建页表来初始化虚拟内存。

//其中页目录项和页表项也可以合在一起看作是二级页表

void Init_VM(struct Boot_Info *bootInfo)
{
    int kernel_pde_entries;
    int whole_pages;
    int i, j;
    uint_t mem_addr;
    pte_t *cur_pte;

    // 计算物理内存的页数

    whole_pages = bootInfo->memSizeKB / 4; //2048 块

    // 计算内核页目录中要多少个目录项，才能完全映射所有的物理内存页。

    kernel_pde_entries = whole_pages / NUM_PAGE_DIR_ENTRIES +
(whole_pages % NUM_PAGE_DIR_ENTRIES == 0 ? 0 : 1); //看是否有余
数 结果为两个目录页

    g_kernel_pde = (pde_t *)Alloc_Page();

    KASSERT(g_kernel_pde != NULL); //断言宏，这个函数用来检查
g_kernel_pde 是否为空指针，如果是，说明内核页目录没有正确分配，程序无法
继续运行
```

```

// 将页中所有位清 0

memset(g_kernel_pde, 0, PAGE_SIZE);

pde_t *cur_pde_entry;

cur_pde_entry = g_kernel_pde;

mem_addr = 0;

for (i = 0; i < kernel_pde_entries - 1; i++)
{
    //这一循环的目的是创建页目录项与页表项，把页目录项指向页表项

    cur_pde_entry->present = 1;

    cur_pde_entry->flags = VM_WRITE;

    cur_pde_entry->globalPage = 1;

    cur_pte = (pte_t *)Alloc_Page();

    KASSERT(cur_pte != NULL);

    // 初始化每一个页目录表项和对应的页表。注意，页表中的页表项不
    一定足够 1024 个

    cur_pde_entry->present = 1;

    cur_pde_entry->flags = VM_WRITE;

    cur_pde_entry->globalPage = 1;

    cur_pte = (pte_t *)Alloc_Page();

    KASSERT(cur_pte != NULL);

    memset(cur_pte, 0, PAGE_SIZE);

    cur_pde_entry->pageTableBaseAddr = (uint_t)cur_pte >>
12;

    int last_pagetable_num;

    last_pagetable_num = whole_pages %
NUM_PAGE_TABLE_ENTRIES;

```



// 注意当 last\_pagetable\_num=0 时，意味着最后一个页目录项对应的页表是满的，就是说页表中 1024 个页表项都指向一个有效的页。

```
    if (last_pagetable_num == 0)
    {
        last_pagetable_num = NUM_PAGE_TABLE_ENTRIES;
    }
    for (j = 0; j < last_pagetable_num; j++)
    {
        cur_pte->present = 1;
        cur_pte->flags = VM_WRITE;
        cur_pte->globalPage = 1;
        cur_pte->pageBaseAddr = mem_addr >> 12;
        cur_pte++;
        mem_addr += PAGE_SIZE;
        // mydebug
        // Print("the mem_addr is %x/n",mem_addr);
    }
```

// 从现在开始，系统的寻址必须经过分页机制转换，以前仅仅经过分段机制转换

```
    Enable_Paging(g_kernel_pde);
```

// 加入一个缺页中断处理程序，并注册其中断号为 14

```
    Install_Interrupt_Handler(14,
Page_Fault_Handler);
}
}
```

第 40 行代码 `cur_pde_entry->pageTableBaseAddr =`

`(uint_t)cur_pte >> 12;` 的解释:

假设当前要映射的线性地址是 0x80001000，它的最高 10 位是 0x200，中间 10 位是 0x000，最低 12 位是 0x000。那么：

- `cur_pde_entry` 指向页目录表中的第 0x200 项，假设它的地址是 0xe000。
- `cur_pte` 指向页表中的第 0x000 项，假设它的地址是 0xf000。
- `pageTableBaseAddr` 存储了页表所在物理页的基地址，假设它是 0x1000。
- `(uint_t)cur_pte >> 12` 得到了页表所在物理页的页号，它是  $0xf000 / 4096 = 0x3$ 。

这段代码就是将 0x3 存储到 `pageTableBaseAddr` 中，即：

`cur_pde_entry->pageTableBaseAddr = 0x3;` 这样，就建立了线性地址 0x80001000 到物理地址 0x300000 的一级映射关系。

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。

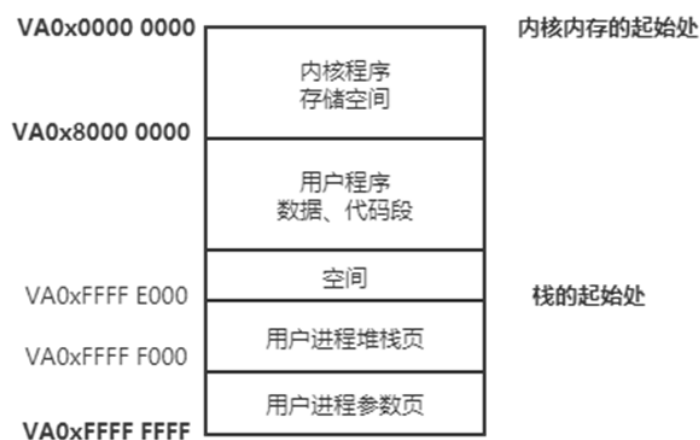


图 6.2 虚拟内存管理分布

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。

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 `0xFFFF 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。

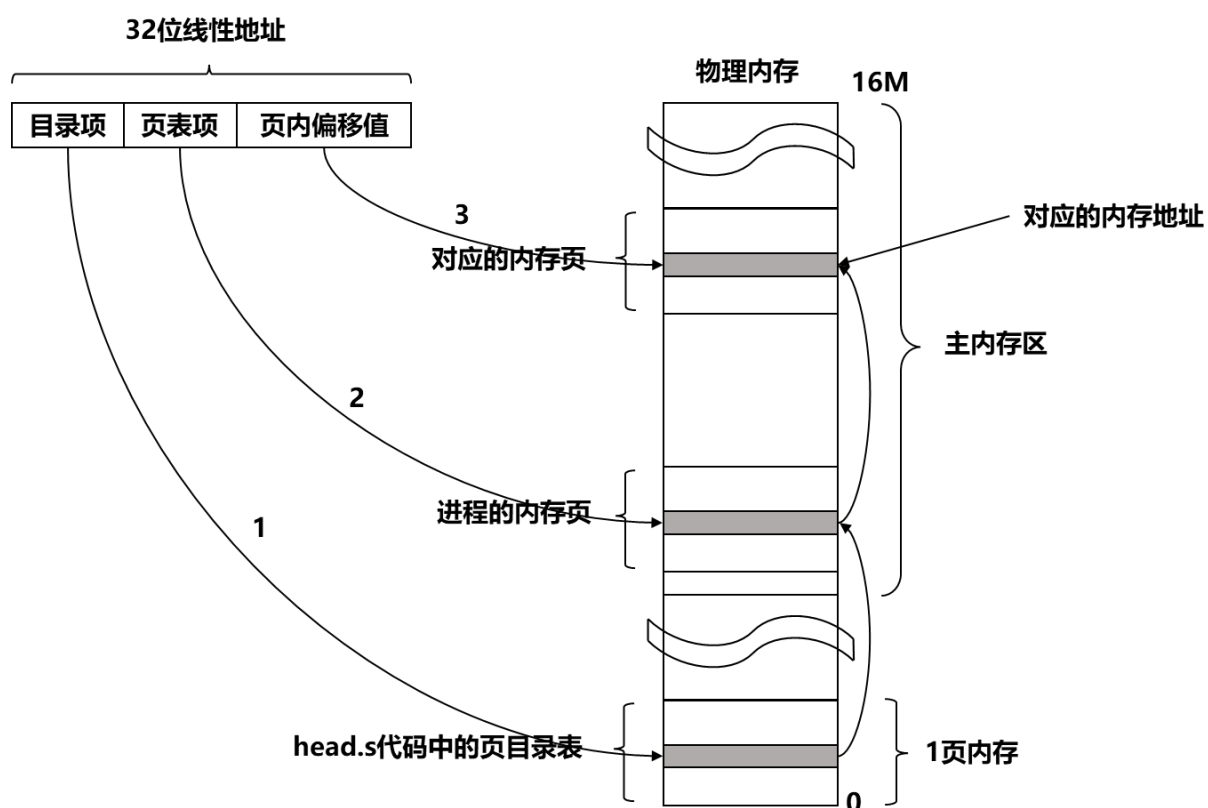


图 6.3 分页存储系统的地址转换机制

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 打印线性地址，标识 5 打印物理地址起始地址，。

```

Initializing IDT...
whole pages are 2048the kernel_pde_entries is 2/nInitializing timer...
Delay loop: 7316 iterations per tick
Initializing keyboard...
Initializing DMA Controller...
Initializing floppy controller...
    fd0: cyl=80, heads=2, sectors=18
Initializing IDE controller...
    ide0: cyl=40, heads=8, sectors=64
Registering paging device: /c/pagefile.bin on ide0
Mounted /c filesystem!
Welcome to GeekOS!
Spawning init process (/c/shell.exe)
Spawn process /c/shell.exe
Load User Program
next will handle page data in Load_User_Program!
1 IN Alloc_User_Page,pagedir_entry=d800
2 IN Alloc_User_Page,new page_entry first=e000
3 IN Alloc_User_Page,page_true=e004
4 IN Alloc_User_Page,liner addr=80001000
5 IN Alloc_User_Page,phical addr=f000
-----
4 IN Alloc_User_Page,liner addr=80002000
5 IN Alloc_User_Page,phical addr=31000
-----
1 IN Alloc_User_Page,pagedir_entry=d800
2 IN Alloc_User_Page,existed page_entry first=e000
3 IN Alloc_User_Page,page_true=e00c
4 IN Alloc_User_Page,liner addr=80003a40
5 IN Alloc_User_Page,phical addr=32000
-----

```

图 6.4 Alloc\_User\_Page()分页管理

第一进程给出的线性地址是 80001000，转换成二进制后是 1000 0000 0000 0000 0001 0000 0000 0000，最高 10 位 1000 0000 00 的十进制是 512，中间 10 位 00 0000 0001 的十进制是 1，最低 12 位 0000 0000 0000 的十进制是 0。

首先，需要从当前页目录表的入口起始地址 d800 开始，找到第 512 项，即  $d800 + 512 * 4 = e000$ 。当前页表当前起始地址 e000 就是第 512 项的内容，表示页表的物理地址。

然后，需要从当前页表当前起始地址 e000 开始，找到第 1 项，即  $e000 + 1 * 4 = e004$ 。当前所在页表位置 e004 就是第 1 项的内容，表示物理页的物理起始地址。

最后，需要从物理页的物理起始地址 f000 开始，加上字节偏移量 0，即  $f000 + 0 = f000$ 。所以当前线性地址对应的物理地址为 f000。

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。

```

-----
$ rec 6
Spawn process rec
Spawn process rec
process -279449600 moving to ready queue -11276288
Queue #0->2
Queue #1->
Queue #2->
Queue #3->
All Thread List:[1][2][3][4][5][6]
Spawn process /c/rec.exe
Spawn process /c/rec.exe
Load_User_Program
next will handle page data in Load_User_Program!
1 IN Alloc_User_Page,pagedir_entry=39800
2 IN Alloc_User_Page,new page_entry first=3a000
3 IN Alloc_User_Page,page_true=3a004
4 IN Alloc_User_Page,liner addr=80001000
5 IN Alloc_User_Page,phical addr=3b000

$ read fault
address=8000173d
userContext->pageDir=d000
page_dir_entry=d800
page_dir_entry->present=1
page_entry=e004
*page_entry=f027
page_entry->present=1
page_entry->pageBaseAddr=f000
page_entry->kernelInfo=0
Pid 6, Page Fault received, at address 8000173d (1637 pages free)
Non-present page, Read Fault, in User Mode
*** Unexpected interrupt! ***
eax=00000000 ebx=00036000 ecx=00036000 edx=00024120
esi=ffffffff edi=00080200 ebp=00101fcc
eip=000169c3 cs=00000008 eflags=00000202
Interrupt number=39, error code=0
index=0, TI=0, IDT=0, EXT=0
cs: index=1, ti=0, rpl=0
ds: index=2, ti=0, rpl=0
es: index=2, ti=0, rpl=0
fs: index=2, ti=0, rpl=0
gs: index=2, ti=0, rpl=0

```

图 6.5 rec 递归程序测试 1

## 7 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/geekos/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 blockListas 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  $1024 \times 1024$  个数据块。

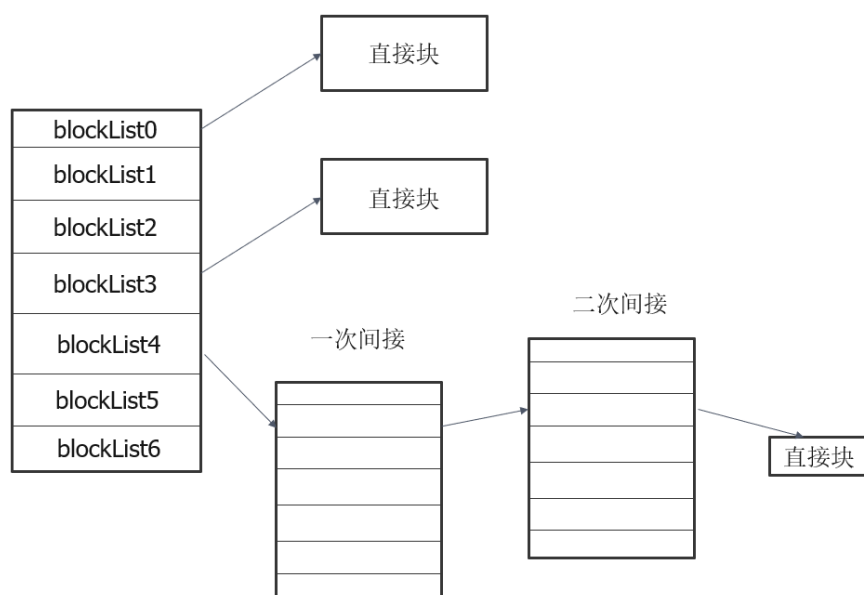


图 7.1 blockList 数组布局

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 系统切换到 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。

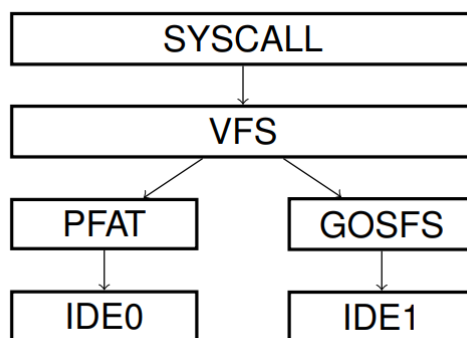


图 7.2 VFS 虚拟文件系统层

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。以下为主函数的代码：

```
src/geekos/main.c |
```

```
/*
 * Define this for a self-contained boot floppy
 * with a PFAT filesystem. (Target "fd_aug.img" in
 * the makefile.)
 */
/*#define FD_BOOT*/
#ifdef FD_BOOT
# define ROOT_DEVICE "fd0"
# define ROOT_PREFIX "a"
#else
# define ROOT_DEVICE "ide0"
# define ROOT_PREFIX "c"
#endif
#define INIT_PROGRAM "/" ROOT_PREFIX "/shell.exe"
static void Mount_Root_FileSystem(void);
static void Spawn_Init_Process(void);
/*
 * Kernel C code entry point.
 * Initializes kernel subsystems, mounts filesystems,
 * and spawns init process.
 */
void Main(struct Boot_Info* bootInfo)
```

```
{
    Init_BSS();
    Init_Screen();
    Init_Mem(bootInfo);
    Init_CRC32();
    Init_TSS();
    Init_Interrupts();
    Init_VM(bootInfo);
    Init_Scheduler();
    Init_Traps();
    Init_Timer();
    Init_Keyboard();
    Init_DMA();
    Init_Floppy();
    Init_IDE();
    Init_PFAT();
    Init_GOSFS();
    Init_MQ();
    Mount_Root_FileSystem();
    Set_Current_Attr(ATTRIB(BLACK, GREEN|BRIGHT));
    Print("Welcome to GeekoS_final!\n");
    Set_Current_Attr(ATTRIB(BLACK, GRAY));
    Spawn_Init_Process();
    /* Now this thread is done. */
    Exit(0);
}

static void Mount_Root_FileSystem(void)
{
    if (Mount(ROOT_DEVICE, ROOT_PREFIX, "pfat") != 0)
```

```
Print("Failed to mount /" ROOT_PREFIX " filesystem\n");
else
Print("Mounted /" ROOT_PREFIX " filesystem!\n");
Init_Paging();
}

static void Spawn_Init_Process(void)
{
// TODO("Spawn the init process");
#ifdef MAIN_DEBUG
Print ("Spawn_Init_Process()\n");
#endif

struct Kernel_Thread *init;
struct File *stdInput = Open_Console_Input();
struct File *stdOutput = Open_Console_Output();
Print("Spawning init process (%s)\n", INIT_PROGRAM);
int rc = Spawn (INIT_PROGRAM, INIT_PROGRAM, stdInput,
stdOutput, &init, 0);

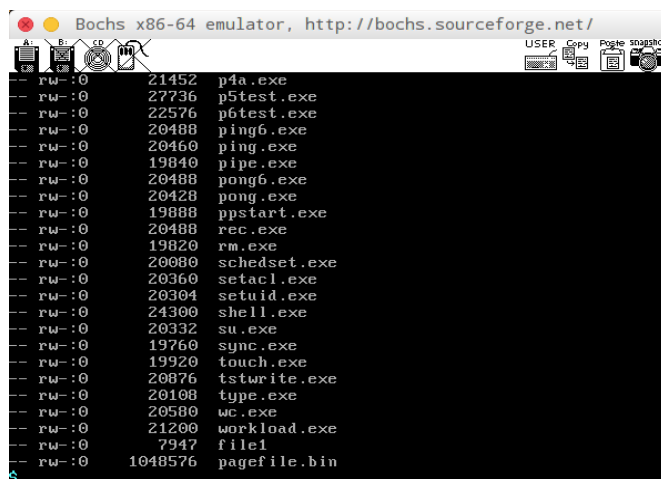
if (rc <= 0) {
Print("Can not Spawn() the INIT process, rc = %d\n",rc);
return;
}

/* wait for termination of the INIT process */
rc = Join (init);
Print("INIT process terminated, rc = %d\n", rc);
}
}
```

## 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

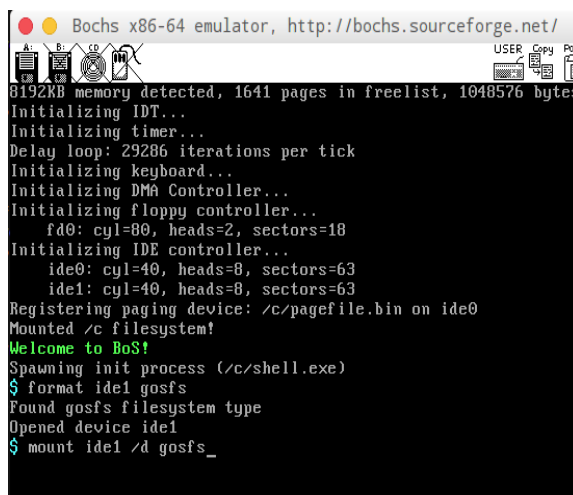
的所有文件，如图 7.3 所示，显示 PFAT 只读文件下挂载的所有文件。



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
-rw-r--r-- 21452 p4a.exe
-rw-r--r-- 27736 p5test.exe
-rw-r--r-- 22576 p6test.exe
-rw-r--r-- 20488 ping6.exe
-rw-r--r-- 20460 ping.exe
-rw-r--r-- 19840 pipe.exe
-rw-r--r-- 20488 pong6.exe
-rw-r--r-- 20428 pong.exe
-rw-r--r-- 19888 ppstart.exe
-rw-r--r-- 20488 rec.exe
-rw-r--r-- 19820 rm.exe
-rw-r--r-- 20080 schedset.exe
-rw-r--r-- 20360 setacl.exe
-rw-r--r-- 20304 setuid.exe
-rw-r--r-- 24300 shell.exe
-rw-r--r-- 20332 su.exe
-rw-r--r-- 19760 sync.exe
-rw-r--r-- 19920 touch.exe
-rw-r--r-- 20876 tstwrite.exe
-rw-r--r-- 20108 type.exe
-rw-r--r-- 20580 wc.exe
-rw-r--r-- 21200 workload.exe
-rw-r--r-- 7947 file1
-rw-r--r-- 1048576 pagefile.bin
```

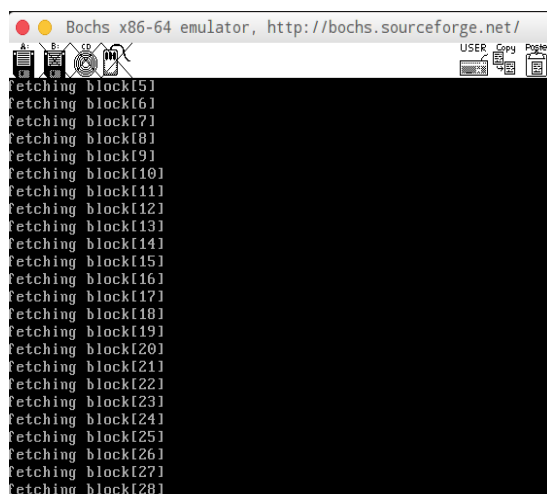
图 7.3 ls 指令测试

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。



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
8192KB memory detected, 1641 pages in freelist, 1048576 bytes
Initializing IDT...
Initializing timer...
Delay loop: 29286 iterations per tick
Initializing keyboard...
Initializing DMA Controller...
Initializing floppy controller...
fd0: cyl=80, heads=2, sectors=18
Initializing IDE controller...
ide0: cyl=40, heads=8, sectors=63
ide1: cyl=40, heads=8, sectors=63
Registering paging device: /c/pagefile.bin on ide0
Mounted /c filesystem!
Welcome to BoS!
Spawning init process (/c/shell.exe)
$ format ide1 gosfs
Found gosfs filesystem type
Opened device ide1
$ mount ide1 /d gosfs_
```

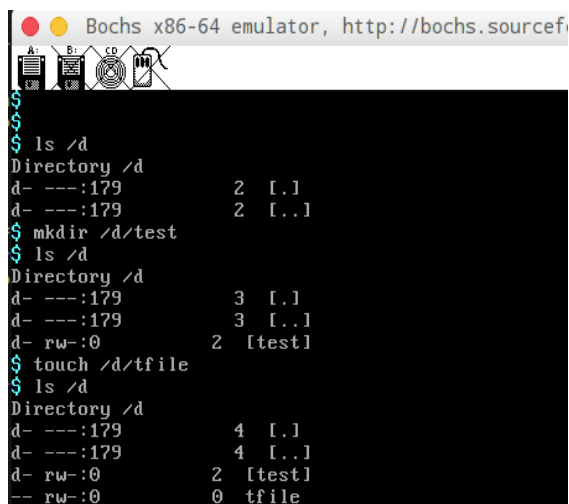
图 7.4 format 指令测试



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
fetching block[5]
fetching block[6]
fetching block[7]
fetching block[8]
fetching block[9]
fetching block[10]
fetching block[11]
fetching block[12]
fetching block[13]
fetching block[14]
fetching block[15]
fetching block[16]
fetching block[17]
fetching block[18]
fetching block[19]
fetching block[20]
fetching block[21]
fetching block[22]
fetching block[23]
fetching block[24]
fetching block[25]
fetching block[26]
fetching block[27]
fetching block[28]
```

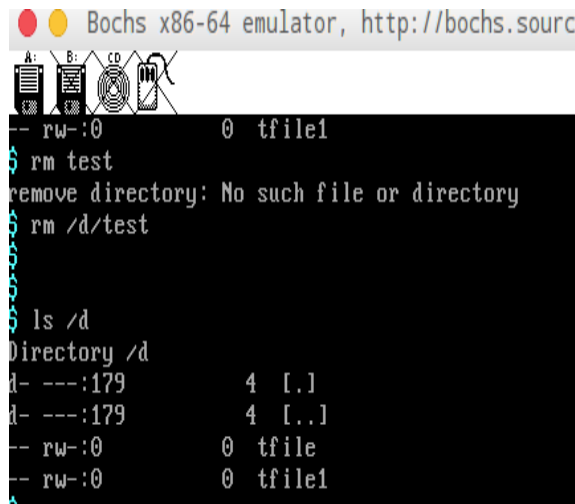
图 7.5 mount 指令测试

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。



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
$
$ ls /d
Directory /d
d- ---:179      2 [.]
d- ---:179      2 [..]
$ mkdir /d/test
$ ls /d
Directory /d
d- ---:179      3 [.]
d- ---:179      3 [..]
d- rw-:0        2 [test]
$ touch /d/tfile
$ ls /d
Directory /d
d- ---:179      4 [.]
d- ---:179      4 [..]
d- rw-:0        2 [test]
-- rw-:0        0 tfile
```

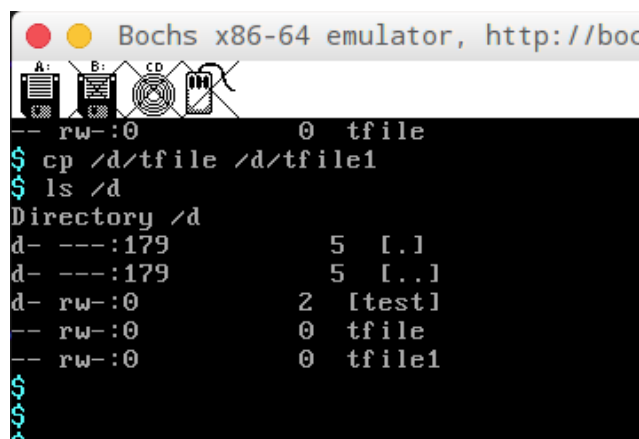
图 7.6 touch 指令测试



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
-- rw-:0        0 tfile1
$ rm test
remove directory: No such file or directory
$ rm /d/test
$ ls /d
Directory /d
d- ---:179      4 [.]
d- ---:179      4 [..]
-- rw-:0        0 tfile
-- rw-:0        0 tfile1
```

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。



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
-- rw-:0        0 tfile
$ cp /d/tfile /d/tfile1
$ ls /d
Directory /d
d- ---:179      5 [.]
d- ---:179      5 [..]
d- rw-:0        2 [test]
-- rw-:0        0 tfile
-- rw-:0        0 tfile1
```

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 25 条测试，失败两条测试，得分为 81。



The screenshot shows the Bochs x86-64 emulator window. The title bar reads 'Bochs x86-64 emulator, http://bochs.sourceforge.net'. The window contains a black terminal-like area with white text showing the output of the p5test program. The tests performed include Basic Seek, Seek w/ Reread, Basic Stat, Stat-File, Stat-Directory, Delete Empty Directory, Delete Non-Empty Directory, 10k Write/Reread, 100k Write/Reread, Read Entry, and 5 MB Write. The results show 25 tests passed and 2 failed, with a total score of 81.

```
Bochs x86-64 emulator, http://bochs.sourceforge.net
Testing: Basic Seek...PASSED (1) crt score: 41
Testing: Seek w/ Reread...PASSED (1) crt score: 46
Testing: Basic Stat...PASSED (1) crt score: 48
Testing: Stat-File...PASSED (1) crt score: 50
Testing: Stat-Directory...ERROR: GOSTFS_FStat_Directory: fsData == NULL!
PASSED (1) crt score: 52
Testing: Delete Empty Directory...PASSED (1) crt score: 55
Testing: Delete Non-Empty Directory...PASSED (1) crt score: 57
Testing: 10k Write/Reread... 0 50 0 50PASSED (1) crt score: 62
Testing: 100k Write/Reread... 0 50 100 150 200 250 300 350 400 450 500 550 600 6
50 700 750 800 850 900 950 0 50 100 150 200 250 300 350 400 450 500 550 600 650
700 750 800 850 900 950PASSED (1) crt score: 69
Testing: Read Entry...PASSED (1) crt score: 73
Testing: 5 MB Write...Writing in first 5MB at random ...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49Reading back from fi
rst 5MB ...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49PASSED (1) crt score
: 81
*****
Tests attempted: 27. Passed: 25. Failed: 2
SCORE: 81
```

图 7.9 p5test 文件批量测试



## 8 课程设计心得体会总结

geekos 操作系统是一个教学用的简单操作系统，它包含了多个设计项目，涉及到操作系统的各个方面，如内核级线程、用户级进程、进程调度、分页存储管理、文件系统等。本次课程实验实现了 project0-project5 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。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)。通过完成这些设计项目，我们小组学习到了操作系统的基本概念、原理和技术，以及如何使用 C 语言和汇编语言编写操作系统代码。操作系统的学习需要结合理论和实践，通过阅读书籍、资料 and 代码，以及动手编写和运行程序，来深入理解操作系统的工作原理和过程，它是一个持续的过程，需要不断地反思和总结，发现自己的优点和不足，提高自己的能力和水平。

## 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>.
- [3] GeekOS web site [EB/OL].<http://geekos.sourceforge.net>. 2005,12.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] R.Gove. CMSC 412 GeekOS Project5 File System[EB/OL].  
[https://www.cs.umd.edu/~hollings/cs412/s10/project5/proj5\\_section.pdf](https://www.cs.umd.edu/~hollings/cs412/s10/project5/proj5_section.pdf).