

**操作系统课程设计报告**

|  |  |
| --- | --- |
| **题目：** | **GeekOS**操作系统的研究与实现 |
|  |  |

|  |  |  |
| --- | --- | --- |
| **组号** | ： | 26 |
| **组长** | ： | 林浩然 |
| **成员** | ： | 詹闳迪 |
|  |  | 陈梦沙 |
|  |  | 邹竹文 |
| **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** | ： | 王慧娇 |
| **职称** | ： | 副教授 |

2022年 5 月 20

操作系统课设设计报告评分标准及评分

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 评分项目 | 分值 | 考核点 | | 实际得分 | |
| 平时表现 | 10 | 学生学习态度积极认真，小组组织科学，分工合理。 | |  | |
| 文献查阅及归纳综合能力 | 20 | 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。 | |  | |
| 设计方案 | 30 | 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。 | |  | |
| 实验及分析 | 20 | 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。 | |  | |
| 报告书写质量 | 20 | 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（%） |
| 1900300619 | 林浩然 | | Project3 | 95 | 25 |
| 1900300635 | 詹闳迪 | | Project4 | 95 | 25 |
| 1900300502 | 陈梦沙 | | Project0、Project1 | 95 | 25 |
| 1900300639 | 邹竹文 | | Project2 | 95 | 25 |

摘 要

教学操作系统有两大类，一类是针对RISC结构MIPS处理器的，另外一类是针对CISC结构的the Intel IA-32 (or x86)通用处理器的。这样分类是因为：处理器是操作系统运行的硬件环境中最重要的部分。

Nachos（Not Another Completely Heuristic Operating System）。是建立在软件模拟的虚拟机之上的教学操作系统，采用MIPS R2/3000的指令集，能模拟主存、中断、网络以及磁盘系统等所必须的硬件系统，美国加州大学伯克利分校多次采用该操作系统作为课程设计平台。OS/161是运行在与操作系统无关的system/161模拟器上的，操作系统代码是MIPS对应的机器代码。无论是Nachos还是OS/161，若学生使用Windows或Linux 开发环境的话，都需要使用交叉编译器才能把代码编译成MIPS相应的机器代码。

GeekOS是一个基于X86架构的PC机上运行的微操作系统内核，由美国马理兰大学的教师开发，是一个用C语言开发的操作系统， GeekOS主要用于操作系统课程设计，目的是使学生能够实际动手参与到一个操作系统的开发工作中。学生可以在Linux或Unix环境下对其进行功能扩充，且其针对进程、文件系统、存储管理等操作系统核心内容分别设计了7个难度逐渐增加的项目供学生选择。

关键词： RISC，Nachos，GeekOS，Linux

目录

[1、绪论 4](#_Toc104890279)

[1.1、课程设计背景 4](#_Toc104890280)

[1.1.1、GeeKOs教学操作系统 4](#_Toc104890281)

[2、项目设计0 6](#_Toc104890282)

[2.1、项目设计要求 6](#_Toc104890283)

[2.2、项目分析 6](#_Toc104890284)

[2.2.1、Start\_Kernel\_Thread函数 6](#_Toc104890285)

[2.2.2、Geekos是如何处理键盘代码 8](#_Toc104890286)

[2.2、项目实现 8](#_Toc104890287)

[3、项目设计1 11](#_Toc104890288)

[3.1、项目设计目的 11](#_Toc104890289)

[3.2、项目设计要求 11](#_Toc104890290)

[3.3.1、ELF文件格式 11](#_Toc104890291)

[3.3.2、内核线程的建立流程 12](#_Toc104890292)

[3.4、项目实现 14](#_Toc104890293)

[4、项目设计2 16](#_Toc104890294)

[4.1、项目设计目的 16](#_Toc104890295)

[4.2、项目设计提示 16](#_Toc104890296)

[4.2.1、 GeekOS进程状态及转换 16](#_Toc104890297)

[4.2.2、GeekOS的用户态进程 17](#_Toc104890298)

[4.2.3、用户态进程创建流程 18](#_Toc104890299)

[4.2.5、用户态进程创建LDT的步骤 20](#_Toc104890300)

[4.3、项目实现 20](#_Toc104890301)

[4.4、结果测试 35](#_Toc104890302)

[5、项目设计3 35](#_Toc104890303)

[5.1、项目设计目的 35](#_Toc104890304)

[5.2、项目设计要求 36](#_Toc104890305)

[5.3、GeekOS进程管理基础知识 36](#_Toc104890306)

[5.3.1、内核进程控制块 36](#_Toc104890307)

[5.3.2、GeekOS系统中最早的内核进程 37](#_Toc104890308)

[5.3.3、内核进程对象 38](#_Toc104890309)

[5.3.4、进程调度 39](#_Toc104890310)

[5.3.5、GeekOS进程调度策略 43](#_Toc104890311)

[5.3.6、GeekOS进程调度处理过程 43](#_Toc104890312)

[5.4、项目设计原理 45](#_Toc104890313)

[5.5、多级反馈策略测试运行流程 47](#_Toc104890314)

[5.6、信号量和PV操作 51](#_Toc104890315)

[6、项目设计4 54](#_Toc104890316)

[6.1、项目设计目的 54](#_Toc104890317)

[6.2、项目设计要求 54](#_Toc104890318)

[6.2.1、完成函数 54](#_Toc104890319)

[6.2.2、完成函数 54](#_Toc104890320)

[6.3、项目设计原理 55](#_Toc104890321)

[6.3.1、了解请求分页虚拟存储管理的基本原理 55](#_Toc104890322)

[6.3.2、请求分页虚实地址转换过程 55](#_Toc104890323)

[6.4、项目具体实现 58](#_Toc104890324)

[参考文献 61](#_Toc104890325)

# 1、绪论

教学操作系统有两大类，一类是针对RISC结构MIPS处理器的，另外一类是针对CISC结构的the Intel IA-32 (or x86)通用处理器的。这样分类是因为：处理器是操作系统运行的硬件环境中最重要的部分。Nachos（Not Another Completely Heuristic Operating System）。是建立在软件模拟的虚拟机之上的教学操作系统，采用MIPS R2/3000的指令集，能模拟主存、中断、网络以及磁盘系统等所必须的硬件系统，美国加州大学伯克利分校多次采用该操作系统作为课程设计平台。

OS/161是运行在与操作系统无关的system/161模拟器上的，操作系统代码是MIPS对应的机器代码。

无论是Nachos还是OS/161，若学生使用Windows或Linux 开发环境的话，都需要使用交叉编译器才能把代码编译成MIPS相应的机器代码。

1.1、课程设计背景

MINIX：是Andrew S. Tanenbaum（AST）于1987年开发的，目前主要有1.5 版和2.0 版两个版本在使用。MINIX 系统是免费的，可以从许多FTP 上下载，MINIX是一个包括了虚拟内存管理、文件系统、设备驱动程序、网络和用户态程序等的比较完整的操作系统，由两万多行代码组成。

1.1.1、GeeKOs教学操作系统

GeekOS：是一个基于X86架构的PC机上运行的微操作系统内核，由美国马理兰大学的教师开发，是一个用C语言开发的操作系统， GeekOS主要用于操作系统课程设计，目的是使学生能够实际动手参与到一个操作系统的开发工作中。学生可以在Linux或Unix环境下对其进行功能扩充，且其针对进程、文件系统、存储管理等操作系统核心内容分别设计了7个难度逐渐增加的项目供学生选择 。

GeekOS操作系统源文件geekos-0.3.0.zip可以从http://geekos.sourceforge.net下载。

图 1.1 GeekOS系统主目录

在doc目录里的文件hacking.pdf和index.htm是GeekOS系统的参考文档。Scripts目录下有startProject和removeEmptyConflicts两个脚本文件。GeekOS系统的源文件在src目录下，分为7个项目：Project0到Project7。在build文件夹中，

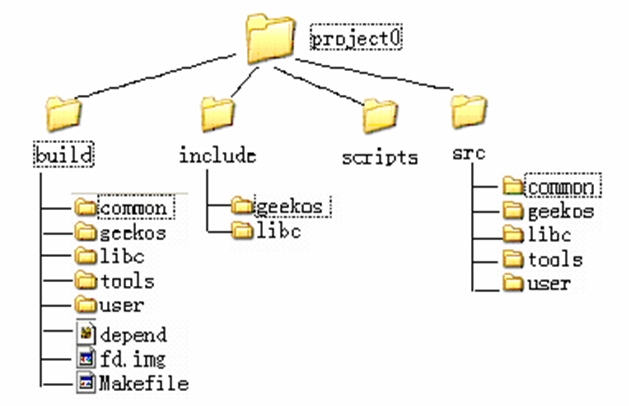
包含系统编译后的可执行文件的文件、软盘镜像或是硬盘镜像、makefile项目管理文件。在include文件夹中有GeekOS和libc两个子目录，在GeekOS子目录中有kthread.h、keyboard.h等文件。

图 1.2 项目文件结构图

# 2、项目设计0

2.1、项目设计要求

项目0要求实现一个内核进程，功能是实现从键盘接收一个按键，并在屏幕上显示。主要是让学生熟悉GeekOS的编译、运行过程，了解计算机系统的启动原理。

2.2、项目分析

在这个项目里面主要用到两部分的内容:内核线程和键盘处理,相应的文件是kthread.c和keyboard.c。GeekOS 在/src/geekos/keyboard.c 和/include/geekos/keyboard.h 分别给出了从键盘获 取字符函数 Read\_Key()的实现以及相应键位所对应的函数返回值，我们在实现 project0() 这个函数的功能时需要调用到 Read\_Key()这个函数，然后根据其返回值分析出所输入的 字符属于什么类型，如果是特殊键就暂时不做处理，等待遇到非特殊键再进行处理。 GeekOS 的启动函数是/src/geekos/main.c 中的 Main()，我们需要在系统初始化完成之后开始执行我们想要执行的代码（即 project0()中的代码），因此需要创建一个内核线 程并将 project0()这个函数作为参数传入，随后系统会自动调度执行。

2.2.1、Start\_Kernel\_Thread函数

①Start\_Kernel\_Thread函数（在kthread.c中）主要功能：建立一个内核线程。

内核线程结构的定义如下：

struct Kernel\_Thread {

unsigned long esp;

volatile unsigned long numTicks;

int priority;

DEFINE\_LINK( Thread\_Queue, Kernel\_Thread );

void\* stackPage;

struct User\_Context\* userContext;

struct Kernel\_Thread\* owner;

int refCount;

Boolean alive;

struct Mutex joinLock;

struct Condition joinCond;

};

esp字段用来存放一个线程挂起的堆栈指针；

stackPage字段指向内核线程的堆栈页面

numTicks和priority分别被调度程序用来实现基于先占权和基于优先权的时间片调度。

DEFINE\_LINK宏定义一个内核线程在线程队列上时的前一个和后一个字段。

userContext字段如果不为空，则指向一个线程用户环境，它是一个允许线程执行用户模式的代码和数据的组合段。

内核线程有两种方式创建。在内核里独立运行的线程可通过Start\_Kernel\_Thread（）函数来创建，该函数通过一个指针指向一个执行线程体的启动函数。线程所执行的用户模式的程序由Start\_User\_Thread（）函数创建，并且用一指针指向一个用户环境和用户环境内存中代码入口点的地址。调用Exit（）函数销毁内核线程。

首先在主程序里调用Start\_Kernel\_Thread 开始创建一条内核线程:Mythread=Start\_Kernel\_Thread (&MyFunction,0,PRIORITY\_NORMAL,false);

入口参数分别为:函数地址，函数参数(无参数就写0),优先级设定，线程属性(false为内核线程，true为用户线程),返回值Mythread 的数据类型是static struct Kernel\_Thread\* thread

②Start\_Kernel\_Thread 完成的工作：

Create\_Thread(priority, detached) //根据优先级创建一条线程

kthread = Alloc\_Page() //为线程分配内存空间

stackPage = Alloc\_Page()

Init\_Thread(kthread, stackPage, priority, detached)

Add\_To\_Back\_Of\_All\_Thread\_List(&s\_allThreadList, kthread)

Setup\_Kernel\_Thread(kthread, startFunc, arg)//配置内核线程的初始化

Make\_Runnable\_Atomic(kthread); //设置线程运行的原子性操作

Disable\_Interrupts(); //禁止中断

Make\_Runnable(kthread); //线程运Enable\_Interrupts(); //使能中断

2.2.2、Geekos是如何处理键盘代码

在keyboard.c里面提供了一个功用函数Keycode Wait\_For\_Key(void),循环等待一个键盘事件,然后返回一个16位的数据 Keycode型的, 在keyboard.h里定义了所有的键盘代码。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。关于Keycode的定义是:

低8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,高六位分别是:

KEY\_SPECIAL\_FLAG (特殊键,比如F1,F2) 用返回的值key&0x0100 就可以判断是否按下特殊健,1为有效,说明是特殊健,0则不是,以下的几种情况类似

KEY\_KEYPAD\_FLAG (小键盘键) 0x0200

KEY\_SHIFT\_FLAG (左,右SHIFT) 0x1000

KEY\_ALT\_FLAG (左,右ALT) 0x2000

KEY\_CTRL\_FLAG (左,右CTRL) 0x4000

KEY\_RELEASE\_FLAG (键弹起来标志位) 0x8000

2.2、项目实现

1) 编写一个C语言函数，函数功能是：接收键盘输入的按键，并将键值在显示器显示出来，当输入ctrl+d就退出；

/project0/src/geekos/main.c：

void project0(){

Print("To Exit hit Ctrl + d.\n");

Keycode keycode;

while(1)

{

if(Read\_Key(&keycode))

{

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

{

Print("\n---------BYE!---------\n");

Exit(1);

}else

{

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

}

}

}

}

}

2) 在Main函数体内调用Start\_Kernel\_Thread函数，将步骤1编写的函数地址传递给参数startFunc，利用Setup\_Kernel\_Thread函数建立一个待运行的线程。

struct Kernel\_Thread \*thread;

thread = Start\_Kernel\_Thread(&project0,0,PRIORITY\_NORMAL,false);

3) 在Linux环境下编译系统得到GeekOS镜像文件。

进入Build目录下:

$ make depend

$ make

4) 编写一个相应的bochs配置文件。

cd ~/geekos-0.3.0/src/project0/build/

gedit .bochsrc

megs: 8

boot: a

floppya: 1\_44=fd.img, status=inserted

Log: ./bochs.out

5) 在bochs中运行GeekOS系统显示结果。

运行Bochs模拟器，执行GeekOS内核。

$ cd ……/build

$ bochs

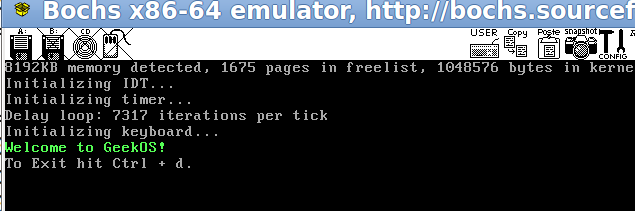
运行后启动界面:

图 2.1 运行后启动界面

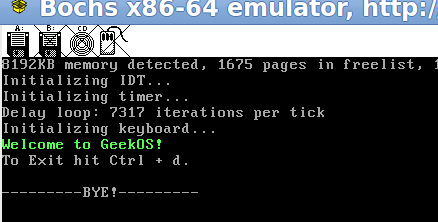
此时可以从键盘向虚拟机输入一下文字，输入完毕后按“Ctrl+d”组合键可退出程序，测试结果如下：

图 2.2 运行后启动界面

# 3、项目设计1

3.1、项目设计目的

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。

3.2、项目设计要求

1) 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。

2) 在Linux环境下编译系统得到GeekOS镜像文件。

3) 编写一个相应的bochs配置文件。

4) 在bochs中运行GeekOS系统显示结果。

3.3、项目分析

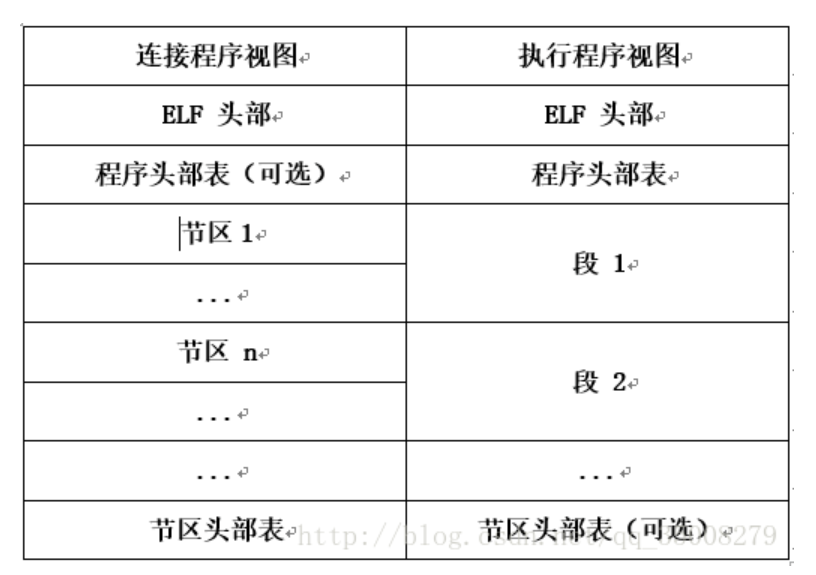
3.3.1、ELF文件格式

表 3.1 ELF文件格式

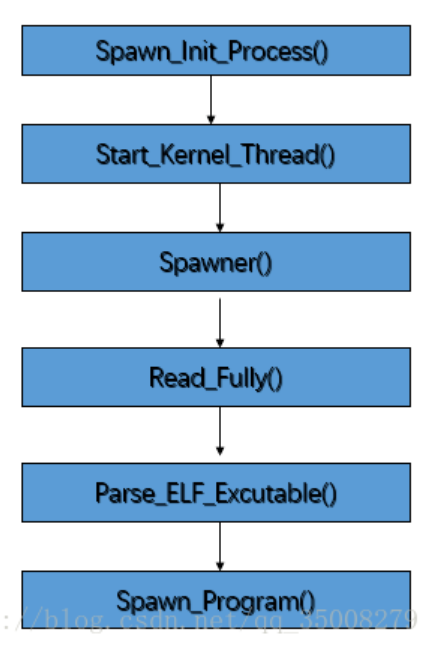
3.3.2、内核线程的建立流程

图 3.1 ELF文件格式

1) Spawn\_Init\_Process函数的功能是调用Start\_Kernel\_Thread()，它以Spawner函数为进程体建立一个核心级进程，并使之准备就绪。

2) Start\_Kernel\_Thread函数通过调用Create\_Thread函数、Setup\_Kernel\_Thread、Make\_Runnable\_Atomic来创建内核进程。

\*Create\_Thread函数主要功能为通过Alloc\_Page函数为Kernel\_Thread结构体和内核进程栈区分配内存空间，并通过Init\_Thread函数初始化Kernel\_Thread结构体。最后将该进程结构体通过Add\_To\_Back\_Of\_All\_Thread\_List函数加入进程队列末尾。

\*Setup\_Kernel\_Thread函数主要是初始化进程栈区，包括参数地址、返回地址、入口地址、寄存器值等。

\*Make\_Runnable\_Atomic函数将该进程设置为就绪态，加入等待执行队列。

3) Spawner函数运行过程为：通过Read\_Fully函数将ELF可执行文件读入内存缓冲区；通过Parse\_ELF\_Executable函数解析ELF文件， 并通过Spawn\_Program函数执行ELF文件。最后通过Free函数释放内存缓冲区。

4) Spawn\_Program函数根据Exe\_Format中的Exe\_Segment结构提供的用户程序段信息及用户进程栈大小计算用户进程所需的内存大小，分配对应的内存空间，并全部初始化为零；根据Exe\_Segment中的segmentList数组，将数据段和代码段从内存缓冲区复制到用户内存空间；生成数据段和代码段的段描述符、段选择子；通过Trampoline函数执行用户进程。

函数Spawn\_Program在/src/geekos/lprog.c文件中已经实现。参数的数据结构Exe\_Format和Exe\_Segment的定义在elf.h中。如下：

struct　Exe\_Format{

struct Exe\_Segment segmentList[EXE\_MAX\_SEGMENTS];

int numSegments; //定义了ELF文件中段的个数

ulong\_t entryAddr; //代码入口地址

};

struct Exe\_Segment{

ulong\_t offsetInFile; //段在可执行文件中的偏移值

ulong\_t lengthInFile; //段在可执行文件中的长度

ulong\_t startAddress; //段在内存中的起始地址

ulong\_t sizeInMemory; //段在内存中的大小

int protFlags; //保护标志

};

3.3.3、Parse\_ELF\_Excutable函数

由于系统已经实现了内核的所有函数，本项目设计只需完成Parse\_ELF\_Executable即可。Parse\_ELF\_Excutable函数声明为：

int Parse\_ELF\_Executable(char \*exeFileData, ulong\_t exeFileLength,struct Exe\_Format \*exeFormat)

参数：exeFileData——已装入内存的可执行文件所占用空间的起始地址

exeFileLength——可执行文件长度

exeFormat——保存分析得到的elf文件信息的结构体指针

根据ELF文件格式，用户可以从exeFileData指向的内容中得到ELF文件头，继续分析可以得到程序头，程序代码段等信息。

3.3.4、a.c文件

在项目1的./src/user目录下有一个a.c文件，编译GeekOS后（即make后）可以得到可执行程序a.exe，并写进PFAT文件系统，路径为c/a.exe。项目将此作为待装入的可执行文件，启动Bochs运行a.exe。

在a.c文件中，既有全局变量也有局部变量，这里还要注意一下局部变量如何输出。

3.4、项目实现

1）编写一个相应的bochs配置文件。

cd ~/geekos-0.3.0/src/project1/build/

gedit .bochsrc

megs: 8

boot: a

floppya: 1\_44=fd.img, status=inserted

Log: ./bochs.out

ata0-master: type=disk, path=diskc.img, mode=flat, cylinders=40, heads=8, spt=64

2）修改/project1/src/geekos/elf.c 文件中的 Parse\_ELF\_Executable()函数，实现对 ELF 格式文件的分析，并将分析结果保存到 Exe\_Format 结构体中，以便系统使用，实现代码如下：

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

elfHeader \*ehdr = (elfHeader\*)exeFileData;

//段的个数

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;

}

return 0;

}

3）修改/project1/src/geekos/lprog.c 文件中的 Spawn\_Program()函数，将其中的 virtSize 局部变量修改为静态全局变量，即在文件头部添加“static unsigned long virtSize;”一行代码。

4）修改 lprog.c 文件中的 Printrap\_Handler()函数，将其改为如下内容：

static void Printrap\_Handler( struct Interrupt\_State\* state )

{

char \*msg;

/\* 此处修改以下内容以正确显示 a.c 中的局部变量 \*/

if (state->eax <= virtSize)

msg = (char \*)virtSpace + state->eax;

else

msg = (char \*)state->eax;

print(msg);

g\_needReschedule = true;

return;

}

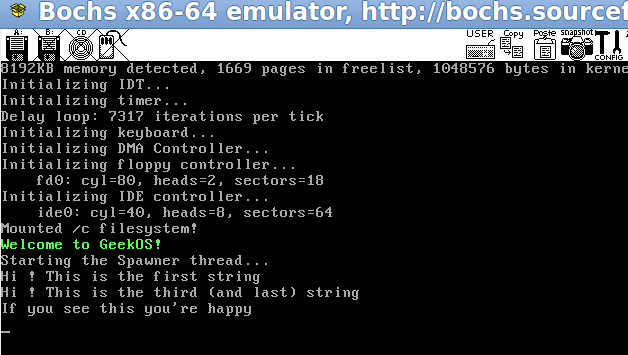
5) 进入“/project1/build/”目录，在终端中执行“bochs”命令，启动运行后得到如下结果：

图 3.2 运行结果

# 4、项目设计2

4.1、项目设计目的

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。

4.2、项目设计提示

4.2.1、 GeekOS进程状态及转换

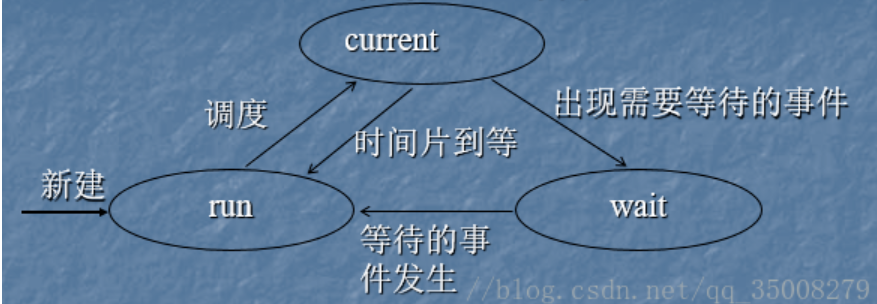
GeekOS系统最早创建的内核进程有Idle、Reaper和Main三个进程，它们由Init\_Scheduler函数创建：最先初始化一个核态进程mainThread，并将该进程作为当前运行进程，函数最后还调用Start\_Kernel\_Thread 函数创建了两个系统进程Idle和Reaper。 所以，Idle、Reaper和Main三个进程是系统中最早存在的进程。

图 4.1 进程状态转换

4.2.2、GeekOS的用户态进程

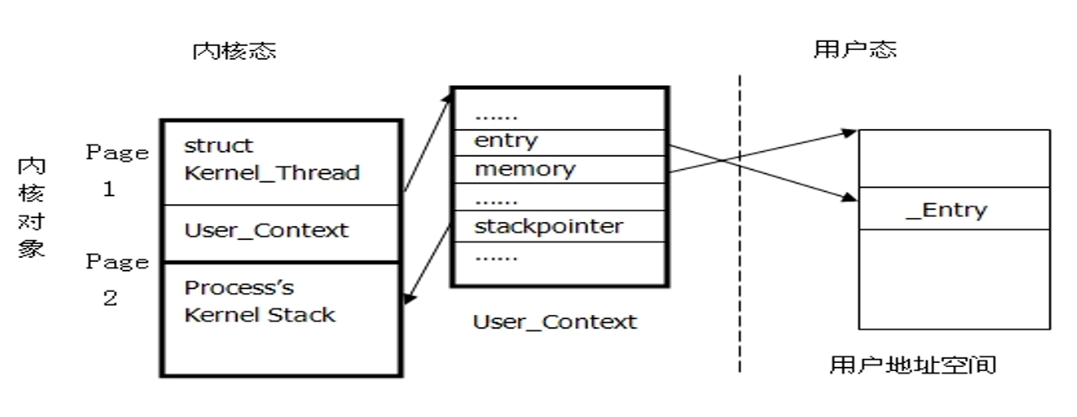
在GeekOS中为了区分用户态进程和内核进程，在Kernel\_Thread结构体中设置了一个字段 userContext，指向用户态进程上下文。对于内核进程来说，这个指针为空，而用户态进程都拥有自己的用户上下文（User\_Context）。因此，在GeekOS中要判断一个进程是内核进程还是用户态进程，只要通过userContext字段是否为空来判断就可以了。

图 4.2 用户进程结构

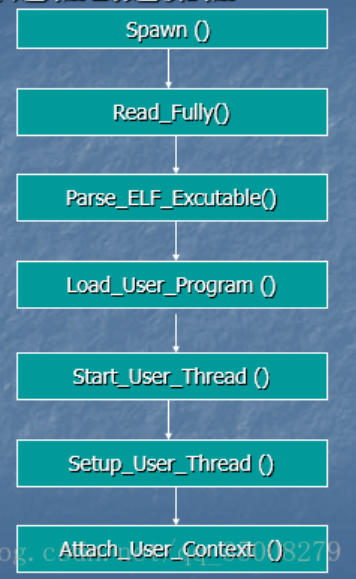
4.2.3、用户态进程创建流程

图 4.3 进程创建流程

Spawn函数的功能

int Spawn(const char \*program, const char \*command, struct Kernel\_Thread \*\*pThread)

参数说明：Program对应的是要读入内存缓冲区的可执行文件，Command是用户执行程序执行时的命令行字符串，pThread是存放指向刚创建进程的指针。

Spawn函数主要完成的主要功能是：

（1）调用Read\_Fully函数将名为program的可执行文件全部读入内存缓冲区。

（2）调用Parse\_ELF\_Executable函数，分析ELF格式文件。Parse\_ELF\_Executable函数功能在项目1中已经实现。

（3）调用Load\_User\_Program将可执行程序的程序段和数据段等装入内存，初始化User\_context数据结构。

（4）调用Start\_User\_Thread函数创建一个进程并使该进程进入准备运行队列。

Load\_User\_Program函数

Load\_User\_Program函数在“/src/geekos/userseg.c”文件中实现，代码也需要开发人员自己完成，函数原型如下：

int Load\_User\_Program(char \*exeFileData, ulong\_t exeFileLength,

struct Exe\_Format \*exeFormat, const char \*command,

struct User\_Context \*\*pUserContext)

/\* 参数说明：

exeFileData——保存在内存缓冲中的用户程序可执行文件；

exeFileLength——可执行文件的长度；

exeFormat——调用Parse\_ELF\_Executable函数得到的可执行文件格式信息；

command——用户输入的命令行，包括可执行文件的名称及其他参数；

pUserContext——指向User\_Conetxt的指针，是本函数完成用户上下文初始化的对象

\*/

Load\_User\_Program主要实现功能如下：

（1）根据Parse\_ELF\_Executable函数的执行结果Exe\_Format中的Exe\_Segment结构提供的用户程序段信息，用户命令参数及用户态进程栈大小计算用户态进程所需的最大内存空间，即要分配给用户态进程的内存空间。

（2）为用户程序分配内存空间，并初始化。

（3）根据Exe\_Segment提供的用户段信息初始化代码段、数据段以及栈段的段描述符和段选择子。

（4）根据段信息将用户程序中的各段内容复制到分配的用户内存空间。

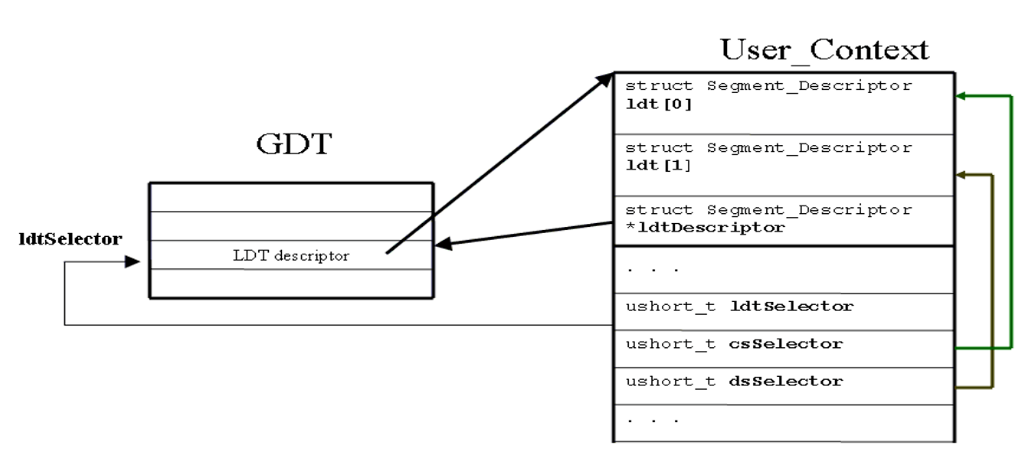
（5）根据Exe\_Format结构初始化User\_Context结构中的用户态进程代码段入口entry字段，并根据command参数初始化用户内存空间中的参数块。

（6）初始化User\_Context结构的用户打开文件列表，并添加标准输入输出文件。

（7）将初始化完毕的User\_Context指针赋予\*pUserContext，返回0表示成功。

4.2.4、用户态进程空间

每个用户态进程都拥有属于自己的内存段空间，如：代码段、数据段、栈段等，每个段有一个段描述符（segment descriptor），并且每个进程有一个段描述符表（Local Descriptor Table），用于保存该进程的所有段描述符。操作系统中还设置一个全局描述符表（GDT，Global Descriptor Table），用于记录了系统中所有进程的ldt描述符。

图 4.4 GDT、LDT、User\_Context

4.2.5、用户态进程创建LDT的步骤

(1)调用函数Allocate\_Segment\_Descriptor()新建一个LDT描述符；

(2)调用函数Selector()新建一个LDT选择子；

(3)调用函数Init\_Code\_Segment\_Descriptor()初始化一个文本段描述符；

(4)调用函数Init\_Data\_Segment\_Descriptor()初始化一个数据段描述符；

(5)调用函数Selector（）新建一个数据段选择子；

(6)调用函数Selector（）新建一个文本（可执行代码）段选择子。

4.3、项目实现

1）“src/GeekOS/user.c”文件中的函数Spawn（），其功能是生成一个新的用户级进程；

int Spawn(const char \*program, const char \*command, struct Kernel\_Thread \*\*pThread) {

int res;

/\* 读取 ELF 文件 \*/

char \*exeFileData = NULL;

ulong\_t exeFileLength = 0;

res = Read\_Fully(program, (void\*\*)&exeFileData, &exeFileLength);

if (res != 0)

{

if (exeFileData != NULL) Free(exeFileData);

return ENOTFOUND;

}

/\* 分析 ELF 文件 \*/

struct Exe\_Format exeFormat;

res = Parse\_ELF\_Executable(exeFileData, exeFileLength, &exeFormat);

if (res != 0)

{

if (exeFileData != NULL) Free(exeFileData);

return res;

}

/\* 加载用户程序 \*/

struct User\_Context \*userContext = NULL;

res = Load\_User\_Program(exeFileData, exeFileLength, &exeFormat, command, &userContext);

if (res != 0)

{

if (exeFileData != NULL) Free(exeFileData);

if (userContext != NULL) Destroy\_User\_Context(userContext);

return res;

}

if (exeFileData != NULL) Free(exeFileData);

exeFileData = NULL;

/\* 开始用户进程 \*/

struct Kernel\_Thread \*thread = NULL;

thread = Start\_User\_Thread(userContext, false);

/\* 超出内存 创建新进程失败 \*/

if (thread == NULL)

{

if (userContext != NULL) Destroy\_User\_Context(userContext);

return ENOMEM;

}

KASSERT(thread->refCount == 2);

/\* 返回核心进程的指针 \*/

\*pThread = thread;

return 0;

}

2）“src/GeekOS/user.c”文件中的函数Switch\_To\_User\_Context（），调度程序在执行一个新的进程前调用该函数以切换用户地址空间；

void Switch\_To\_User\_Context(struct Kernel\_Thread\* kthread, struct Interrupt\_State\* state)

{

/\*

\* Hint: Before executing in user mode, you will need to call

\* the Set\_Kernel\_Stack\_Pointer() and Switch\_To\_Address\_Space()

\* functions.

\*/

//之前最近使用过的 userContxt

static struct User\_Context\* s\_currentUserContext;

//指向User\_Conetxt的指针，并初始化为准备切换的进程

struct User\_Context\* userContext = kthread->userContext;

KASSERT(!Interrupts\_Enabled());

//userContext为0表示此进程为核心态进程就不用切换地址空间

if (userContext == 0) return;

if (userContext != s\_currentUserContext)

{

//为用户态进程时则切换地址空间

Switch\_To\_Address\_Space(userContext);

//新进程的核心栈指针

ulong\_t esp0 = ((ulong\_t)kthread->stackPage) + PAGE\_SIZE;

//设置内核堆栈指针

Set\_Kernel\_Stack\_Pointer(esp0);

//保存新的 userContxt

s\_currentUserContext = userContext;

}

}

3）“src/GeekOS/elf.c”文件中的函数Parse\_ELF\_Executable（）。该函数的实现要求和项目1相同。

4）“src/GeekOS/userseg.c”文件中主要是实现一些为实现对“src/GeekOS/user.c”中高层操作支持的函数。

Create\_User\_Context（）函数，用于创 建并初始化一个用户上下文结构。

/\*\*需在此文件各函数前增加一个函数，此函数的功能是按给定的大小创建一个用户级进程上下文，具体实现如下：

函数功能:按给定的大小创建一个用户级进程上下文\*\*/

static struct User\_Context\* Create\_User\_Context(ulong\_t size)

{

struct User\_Context \*userContext;

size = Round\_Up\_To\_Page(size);

userContext = (struct User\_Context \*)Malloc(sizeof(struct User\_Context));

/\* 内存分配成功则继续为 userContext 下的 memory 分配内存空间 \*/

if (userContext == NULL)

{

return NULL;

}

userContext->memory = (char \*)Malloc(size);

if (userContext->memory == NULL)

{

Free(userContext);

return NULL;

}

memset(userContext->memory, '\0', size);

userContext->size = size;

/\* 新建一个 LDT 描述符 \*/

userContext->ldtDescriptor = Allocate\_Segment\_Descriptor();

if (userContext->ldtDescriptor == NULL)

{

Free(userContext->memory);

return NULL;

}

/\* 初始化段描述符 \*/

Init\_LDT\_Descriptor(userContext->ldtDescriptor, userContext->ldt, NUM\_USER\_LDT\_ENTRIES);

/\* 新建一个 LDT 选择子 \*/

userContext->ldtSelector = Selector(KERNEL\_PRIVILEGE, true, Get\_Descriptor\_Index(userContext->ldtDescriptor));

/\* 新建一个代码段描述符 \*/

Init\_Code\_Segment\_Descriptor(&userContext->ldt[0], (ulong\_t)userContext->memory, size / PAGE\_SIZE, USER\_PRIVILEGE);

/\* 新建一个数据段描述符 \*/

Init\_Data\_Segment\_Descriptor(&userContext->ldt[1], (ulong\_t)userContext->memory, size / PAGE\_SIZE, USER\_PRIVILEGE);

/\* 新建数据段和代码段选择子 \*/

userContext->csSelector = Selector(USER\_PRIVILEGE, false, 0);

userContext->dsSelector = Selector(USER\_PRIVILEGE, false, 1);

/\* 将引用数清零 \*/

userContext->refCount = 0;

return userContext;

}

Destroy\_User\_Context（）函数的功能是释放用户态进程占用的内存资源。

void Destroy\_User\_Context(struct User\_Context\* userContext)

{

KASSERT(userContext->refCount == 0);

/\* 释放 LDT descriptor \*/

Free\_Segment\_Descriptor(userContext->ldtDescriptor);

/\* 释放内存空间 \*/

Disable\_Interrupts();

Free(userContext->memory);

Free(userContext);

Enable\_Interrupts();

}

Load\_User\_Program（）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。

int Load\_User\_Program(char \*exeFileData, ulong\_t exeFileLength,

struct Exe\_Format \*exeFormat, const char \*command,

struct User\_Context \*\*pUserContext)

{

unsigned int i;

struct User\_Context \*userContext = NULL;

/\* 要分配的最大内存空间 \*/

ulong\_t maxva = 0;

/\* 计算用户态进程所需的最大内存空间 \*/

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

{

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

ulong\_t topva = segment->startAddress + segment->sizeInMemory;

if (topva > maxva) maxva = topva;

}

/\* 程序参数数目 \*/

unsigned int numArgs;

/\* 获取参数块的大小 \*/

ulong\_t argBlockSize;

Get\_Argument\_Block\_Size(command, &numArgs, &argBlockSize);

/\* 用户进程大小 = 参数块总大小 + 进程堆栈大小(8192) \*/

ulong\_t size = Round\_Up\_To\_Page(maxva) + DEFAULT\_USER\_STACK\_SIZE;

/\* 参数块地址 \*/

ulong\_t argBlockAddr = size;

size += argBlockSize;

/\* 按相应大小创建一个进程 \*/

userContext = Create\_User\_Context(size);

/\* 如果进程创建失败则返回错误信息 \*/

if (userContext == NULL)

{

return -1;

}

/\* 将用户程序中的各段内容复制到分配的用户内存空间 \*/

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

{

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

memcpy(userContext->memory + segment->startAddress,

exeFileData + segment->offsetInFile,

segment->lengthInFile);

}

/\* 格式化参数块 \*/

Format\_Argument\_Block(userContext->memory + argBlockAddr, numArgs, argBlockAddr, command);

/\* 初始化数据段、堆栈段及代码段信息 \*/

userContext->entryAddr = exeFormat->entryAddr;

userContext->argBlockAddr = argBlockAddr;

userContext->stackPointerAddr = argBlockAddr;

/\* 将初始化完毕的 User\_Context 赋给\*pUserContext \*/

\*pUserContext = userContext;

return 0;

}

Copy\_From\_User（）和Copy\_To\_User（）函数的功能是在用户地址空间和内核地址空间之间复制数据，在分段存储器管理模式下，只要段有效，调用memcpy函数就可以实现这两个函数的功能。

bool Copy\_From\_User(void\* destInKernel, ulong\_t srcInUser, ulong\_t bufSize)

{

struct User\_Context\* userContext = g\_currentThread->userContext;

/\* 如果访问的用户内存空间非法(越界访问)，则直接返回失败 \*/

if (!Validate\_User\_Memory(userContext, srcInUser, bufSize))

return false;

/\* 拷贝当前用户内存空间数据到系统内核空间 \*/

memcpy(destInKernel, userContext->memory + srcInUser, bufSize);

/\* 拷贝成功则返回 1 \*/

return true;

}

bool Copy\_To\_User(ulong\_t destInUser, void\* srcInKernel, ulong\_t bufSize)

{

struct User\_Context\* userContext = g\_currentThread->userContext;

/\* 如果需要拷贝的内容超出用户内存空间(越界)，则直接返回失败 \*/

if (!Validate\_User\_Memory(userContext, destInUser, bufSize))

return false;

/\* 拷贝当前系统内核空间数据到用户内存空间 \*/

memcpy(userContext->memory + destInUser, srcInKernel, bufSize);

/\* 拷贝成功则返回 1 \*/

return true;

}

Switch\_To\_Address\_Space（）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。

void Switch\_To\_Address\_Space(struct User\_Context \*userContext)

{

/\* 切换到新的局部描述符表(LDT) \*/

ushort\_t ldtSelector = userContext->ldtSelector;

\_\_asm\_\_ \_\_volatile\_\_ (

"lldt %0"

:

: "a" (ldtSelector)

);

}

5)“src/GeekOS/kthread.c”文件中的Start\_User\_Thread函数和Setup\_User\_Thread函数。

修改/project2/src/geekos/kthread.c 文件，添加头文件“#include < geekos/user.h>”。

Setup\_User\_Thread（）函数的功能是为进程初始化内核栈，栈中是为进程首次进入用户态运行时设置处理器状态要使用的数据。

/\*

\* Set up the a user mode thread.

\*/

void Setup\_User\_Thread(

struct Kernel\_Thread\* kthread, struct User\_Context\* userContext)

{

/\*

\* Push the argument to the thread start function, and the

\* return address (the Shutdown\_Thread function, so the thread will

\* go away cleanly when the start function returns).

\*/

ulong\_t eflags = EFLAGS\_IF;

unsigned int csSelector = userContext->csSelector; /\* CS 选择子 \*/

unsigned int dsSelector = userContext->dsSelector; /\* DS 选择子 \*/

/\* 调用 Attach\_User\_Context 加载用户上下文 \*/

Attach\_User\_Context(kthread, userContext);

/\* 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 函数将以下数据压入堆栈 \*/

Push(kthread, dsSelector); /\* DS 选择子 \*/

Push(kthread, userContext->stackPointerAddr); /\* 堆栈指针 \*/

Push(kthread, eflags); /\* Eflags \*/

Push(kthread, csSelector); /\* CS 选择子 \*/

Push(kthread, userContext->entryAddr); /\* 程序计数器 \*/

Push(kthread, 0); /\* 错误代码(0) \*/

Push(kthread, 0); /\* 中断号(0) \*/

// if (uthreadDebug)

// Print("Entry addr=%lx\n", userContext->entryAddr);

/\* 初始化通用寄存单元，向 esi 传递参数块地址 \*/

Push(kthread, 0); /\* eax \*/

Push(kthread, 0); /\* ebx \*/

Push(kthread, 0); /\* ecx \*/

Push(kthread, 0); /\* edx \*/

Push(kthread, userContext->argBlockAddr); /\* esi \*/

Push(kthread, 0); /\* edi \*/

Push(kthread, 0); /\* ebp \*/

/\* 初始化数据段寄存单元 \*/

Push(kthread, dsSelector); /\* ds \*/

Push(kthread, dsSelector); /\* es \*/

Push(kthread, dsSelector); /\* fs \*/

Push(kthread, dsSelector); /\* gs \*/

}

Start\_User\_Thread（）是一个高层操作，该函数使用User\_Context对象开始一个新进程。

/\*

\* Start a user-mode thread (i.e., a process), using given user context.

\* Returns pointer to the new thread if successful, null otherwise.

\*/

struct Kernel\_Thread\*

Start\_User\_Thread(struct User\_Context\* userContext, bool detached)

{

/\* 如果传入的用户上下文字段为空(非用户态进程)则返回错误 \*/

if (userContext == NULL)

{

return NULL;

}

/\* 建立用户态进程 \*/

struct Kernel\_Thread \*kthread = Create\_Thread(PRIORITY\_USER, detached);

if (kthread == NULL)

{

return NULL;

}

Setup\_User\_Thread(kthread, userContext);

/\* 将新创建的进程加入就绪进程队列 \*/

Make\_Runnable\_Atomic(kthread);

/\* 新用户态进程创建成功，返回指向该进程的指针 \*/

return kthread;

}

6）“src/GeekOS//syscall.c”文件中主要是实现用户程序要求内核进行服务的一些系统调用函数定义。要求用户实现的有Sys\_Exit（）函数、Sys\_PrintString（）函数、Sys\_GetKey（）、Sys\_SetAttr（）、Sys\_GetCursor（）、Sys\_PutCursor（）、Sys\_Spawn（）函数、Sys\_Wait（）函数和Sys\_GetPID( )函数。

在/project2/src/geekos/syscall.c 文件中添加函数 Copy\_User\_String()，用于在 Sys\_PrintString 中使用

/\* 添加函数 Copy\_User\_String 以便在函数 Sys\_PrintString 中使用 \*/

static int Copy\_User\_String(ulong\_t uaddr, ulong\_t len,

ulong\_t maxLen, char \*\*pStr)

{

int result = 0;

char \*str;

/\* 字符串超过最大长度 \*/

if (len > maxLen) return EINVALID;

/\* 为字符串分配内存空间 \*/

str = (char\*)Malloc(len + 1);

if (str == 0){

return ENOMEM;

goto fail;

}

/\* 从用户内存空间中复制字符串到系统内核空间 \*/

if (!Copy\_From\_User(str, uaddr, len))

{

result = EINVALID;

Free(str);

goto fail;

}

str[len] = '\0';

/\* 拷贝成功 \*/

\*pStr = str;

fail:

return result;

}

Sys\_Exit（）函数

static int Sys\_Exit(struct Interrupt\_State\* state)

{

Exit(state->ebx);

}

Sys\_PrintString（）函数

static int Sys\_PrintString(struct Interrupt\_State\* state)

{

int result = 0;//返回值

uint\_t length = state->ecx;//字符串长度

uchar\_t\* buf = 0;

if (length > 0){

/\* 将字符串复制到系统内核空间 \*/

if (Copy\_User\_String(state->ebx, length, 1023, (char\*\*)&buf) != 0)

goto done;

/\* 输出字符串到控制台 \*/

Put\_Buf(buf, length);

}

done:

if (buf != NULL) Free(buf);

return result;

}

Sys\_GetKey（）函数

static int Sys\_GetKey(struct Interrupt\_State\* state)

{

/\* 返回按键码 \*/

/\* /geekos/keyboard.c

Keycode Wait\_For\_Key(void) \*/

return Wait\_For\_Key();

}

Sys\_SetAttr（）函数

static int Sys\_SetAttr(struct Interrupt\_State\* state)

{

/\* 设置当前文本显示格式 \*/

/\* /geekos/screen.c

void Set\_Current\_Attr(uchar\_t attrib) \*/

Set\_Current\_Attr((uchar\_t)state->ebx);

return 0;

}

Sys\_GetCursor（）函数

static int Sys\_GetCursor(struct Interrupt\_State\* state)

{

/\* 获取当前光标所在屏幕位置(行和列) \*/

int row, col;

Get\_Cursor(&row, &col);

if (!Copy\_To\_User(state->ebx, &row, sizeof(int)) ||

!Copy\_To\_User(state->ecx, &col, sizeof(int)))

return -1;

return 0;

}

Sys\_PutCursor（）函数

static int Sys\_PutCursor(struct Interrupt\_State\* state)

{

/\* 设置光标的位置(行和列) \*/

return Put\_Cursor(state->ebx, state->ecx) ? 0 : -1;

}

Sys\_Spawn（）函数函数

static int Sys\_Spawn(struct Interrupt\_State\* state)

{

int res;//程序返回值

char \*program = 0;//进程名称

char \*command = 0;//用户命令

struct Kernel\_Thread \*process;

/\* 复制程序名和命令字符串到用户内存空间 \*/

res = Copy\_User\_String(state->ebx, state->ecx, VFS\_MAX\_PATH\_LEN, &program);

if (res != 0)

{//从用户空间复制进程名称

goto fail;

}

res = Copy\_User\_String(state->edx, state->esi, 1023, &command);

if (res != 0)

{//从用户空间复制用户命令

goto fail;

}

/\* 生成用户进程 \*/

Enable\_Interrupts();//开中断

res = Spawn(program, command, &process);//得到进程名称和用户命令后便可生成一个新进程

if (res == 0) {//若成功则返回新进程ID号

KASSERT(process != 0);

res = process->pid;

}

Disable\_Interrupts();//关中断

fail:

if (program != 0)

Free(program);

if (command != 0)

Free(command);

return res;

}

Sys\_Wait（）函数

static int Sys\_Wait(struct Interrupt\_State\* state)

{

int exitCode;

/\* 查找需要等待的进程 \*/

struct Kernel\_Thread \*kthread = Lookup\_Thread(state->ebx);

/\* 如果没有找到需要等待的进程，则返回错误代码 \*/

if (kthread == 0) return -1;

/\* 等待指定进程结束 \*/

Enable\_Interrupts();

exitCode = Join(kthread);

Disable\_Interrupts();

return exitCode;

}

Sys\_GetPID( )函数。

static int Sys\_GetPID(struct Interrupt\_State\* state)

{

/\* 返回当前进程的 ID(PID) \*/

return g\_currentThread->pid;

}

7）在main.c文件中改写生成第一个用户态进程的函数调用：Spawn\_Init\_Process(void) 。

static void Spawn\_Init\_Process(void)

{

struct Kernel\_Thread \*pThread;

Spawn("/c/shell.exe","/c/shell.exe",&pThread);

}

8）更改bochsrc配置文件，同project1。

4.4、结果测试

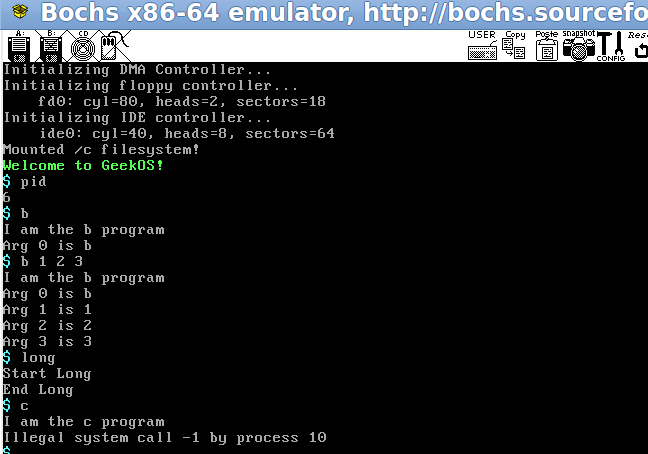
bochs运行后，输入相关命令，即可执行project2/src/user/下的各个可执行文件。由于Spawn\_Init\_Process(void)里面填写的是shell程序，所以从shell开始执行，输入shell程序中的相关命令，即可得到不同结果。

图 4.5 运行结果

# 5、项目设计3

5.1、项目设计目的

熟悉GeekOS的项目编译、调试和运行环境，掌握GeekOS运行工作过程。

5.2、项目设计要求

(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geekos/syscall.c文件中的Sys\_GetTimeOfDaysrc/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( )。

PS：前三个要求都是关于多级反馈调度策略的，第一个是设置，第三个是转换，第二个用来评估算法；第四个要求是有关信号量操作。

5.3、GeekOS进程管理基础知识

5.3.1、内核进程控制块

系统中每个内核进程有且只有一个进程控制块，进程控制块是用于记录进程状态及有关信息的数据结构。GeekOS操作系统中用数据结构Kernel\_Thread作为内核进程控制块，对系统中的进程信息、执行情况、控制信息等加以维护，这个结构在include/kthread.h中定义。

struct Kernel\_Thread {

ulong\_t esp; /\* offset 0 \*/

volatile ulong\_t numTicks; /\* offset 4 \*/

int priority;

DEFINE\_LINK(Thread\_Queue, Kernel\_Thread);

void\* stackPage;

struct User\_Context\* userContext;

struct Kernel\_Thread\* owner;

int refCount;

/\* These fields are used to implement the Join() function \*/

bool alive;

struct Thread\_Queue joinQueue;

int exitCode;

/\* The kernel thread id; also used as process id \*/

int pid;

/\* Link fields for list of all threads in the system. \*/

DEFINE\_LINK(All\_Thread\_List, Kernel\_Thread);

/\* Array of MAX\_TLOCAL\_KEYS pointers to thread-local data. \*/

#define MAX\_TLOCAL\_KEYS 128

const void\* tlocalData[MAX\_TLOCAL\_KEYS];

/\*

\* The run queue level that the thread should be put on

\* when it is restarted.

\*/

int currentReadyQueue;

bool blocked;

};

5.3.2、GeekOS系统中最早的内核进程

GeekOS系统最早创建的内核进程有Idle、Peaper和Main3个进程。在系统初始化时，Main函数调用了一系列初始化函数，其中Init\_Scheduler函数（\src\geekos\kthread.c）的作用：

初始化一个内核进程mainThread，并将该进程作为当前运行进程；

创建两个系统进程Idle和Reaper；

Idle进程类似于Windows中的系统闲置进程，什么也不做，创建后就一直存在于系统中，它存在的唯一目的是保证准备运行队列中有可调度的进程。当系统没有可运行的进程时，CPU就运行Idle，一旦有其他准备运行的进程进入，Idle就会立即放弃CPU。

Reaper负责消亡进程的善后工作，如释放消亡进程占用的资源，内存、堆栈等。

void Init\_Scheduler(void)

{

g\_preSchedulingPolicy = ROUND\_ROBIN;

g\_curSchedulingPolicy = MULTILEVEL\_FEEDBACK;

struct Kernel\_Thread\* mainThread = (struct Kernel\_Thread \*) KERN\_THREAD\_OBJ;

/\*

\* Create initial kernel thread context object and stack,

\* and make them current.

\*/

Init\_Thread(mainThread, (void \*) KERN\_STACK, PRIORITY\_NORMAL, true);

g\_currentThread = mainThread;

Add\_To\_Back\_Of\_All\_Thread\_List(&s\_allThreadList, mainThread);

/\*

\* Create the idle thread.

\*/

/\*Print("starting idle thread\n");\*/

Start\_Kernel\_Thread(Idle, 0, PRIORITY\_IDLE, true);

/\*

\* Create the reaper thread.

\*/

/\*Print("starting reaper thread\n");\*/

Start\_Kernel\_Thread(Reaper, 0, PRIORITY\_NORMAL, true);

}

5.3.3、内核进程对象

创建一个GeekOS内核进程需要调用Start\_Kernel\_Thread函数，而Start\_Kernel\_Thread内部调用Creat\_Thread函数 ，该函数主要是创建内核进程对象，并调用Alloc\_Page函数为进程对象、进程内核堆栈各分配一页内存（若失败，返回0，同时释放内核控制块空间）。

struct Kernel\_Thread\* Start\_Kernel\_Thread(

Thread\_Start\_Func startFunc,

ulong\_t arg,

int priority,

bool detached

)

{

struct Kernel\_Thread\* kthread = Create\_Thread(priority, detached);

if (kthread != 0) {

/\*

\* Create the initial context for the thread to make

\* it schedulable.

\*/

Setup\_Kernel\_Thread(kthread, startFunc, arg);

/\* Atomically put the thread on the run queue. \*/

Make\_Runnable\_Atomic(kthread);

}

return kthread;

}

5.3.4、进程调度

GeekOS在一下几种情况会发生进程切换：

1）时间片用完；

2）执行内核进程Idle；

3）进程退出调用Exit函数；

4）进程进入等待调用Wait函数。

第一种情况下，当时间片用完，进程切换在中断处理函数Handle\_Interrupt（在lowlevel.asm中定义）中完成。

align 8

Handle\_Interrupt:

; Save registers (general purpose and segment)

Save\_Registers

; Ensure that we're using the kernel data segment

mov ax, KERNEL\_DS

mov ds, ax

mov es, ax

; Get the address of the C handler function from the

; table of handler functions.

mov eax, g\_interruptTable ; get address of handler table

mov esi, [esp+REG\_SKIP] ; get interrupt number

mov ebx, [eax+esi\*4] ; get address of handler function

; Call the handler.

; The argument passed is a pointer to an Interrupt\_State struct,

; which describes the stack layout for all interrupts.

push esp

call ebx

add esp, 4 ; clear 1 argument

; If preemption is disabled, then the current thread

; keeps running.

cmp [g\_preemptionDisabled], dword 0

jne .restore

; See if we need to choose a new thread to run.

cmp [g\_needReschedule], dword 0

je .restore

; Put current thread back on the run queue

push dword [g\_currentThread]

call Make\_Runnable

add esp, 4 ; clear 1 argument

; Save stack pointer in current thread context, and

; clear numTicks field.

mov eax, [g\_currentThread]

mov [eax+0], esp ; esp field

mov [eax+4], dword 0 ; numTicks field

; Pick a new thread to run, and switch to its stack

call Get\_Next\_Runnable

mov [g\_currentThread], eax

mov esp, [eax+0] ; esp field

; Clear "need reschedule" flag

mov [g\_needReschedule], dword 0

.restore:

; Activate the user context, if necessary.

Activate\_User\_Context

; Restore registers

Restore\_Registers

; Return from the interrupt.

iret

后三种情况，函数内部都有调用Schedule()函数，其中又调用Switch\_To\_Thread函数（在lowlevel.asm中定义）

align 16

Switch\_To\_Thread:

; Modify the stack to allow a later return via an iret instruction.

; We start with a stack that looks like this:

;

; thread\_ptr

; esp --> return addr

;

; We change it to look like this:

;

; thread\_ptr

; eflags

; cs

; esp --> return addr

push eax ; save eax

mov eax, [esp+4] ; get return address

mov [esp-4], eax ; move return addr down 8 bytes from orig loc

add esp, 8 ; move stack ptr up

pushfd ; put eflags where return address was

mov eax, [esp-4] ; restore saved value of eax

push dword KERNEL\_CS ; push cs selector

sub esp, 4 ; point stack ptr at return address

; Push fake error code and interrupt number

push dword 0

push dword 0

; Save general purpose registers.

Save\_Registers

; Save stack pointer in the thread context struct (at offset 0).

mov eax, [g\_currentThread]

mov [eax+0], esp

; Clear numTicks field in thread context, since this

; thread is being suspended.

mov [eax+4], dword 0

; Load the pointer to the new thread context into eax.

; We skip over the Interrupt\_State struct on the stack to

; get the parameter.

mov eax, [esp+INTERRUPT\_STATE\_SIZE]

; Make the new thread current, and switch to its stack.

mov [g\_currentThread], eax

mov esp, [eax+0]

; Activate the user context, if necessary.

Activate\_User\_Context

; Restore general purpose and segment registers, and clear interrupt

; number and error code.

Restore\_Registers

; We'll return to the place where the thread was

; executing last.

iret

; Return current contents of eflags register.

以上两个关键函数均在汇编文件中实现，最终执行一个iret指令中断返回，跳入目标进程。

5.3.5、GeekOS进程调度策略

GeekOS的初始系统提供的进程调度是时间片轮转调度，所有准备运行进程（即Kernel\_Thread）都放在一个FIFO队列里面，进程调度时找优先级最高的进程投入运行。

在GeekOS中，Get\_Next\_Runnable函数就是进程调度算法实现的地方，由\src\geekos\kthread.c文件中的Find\_Best函数在准备运行进程的队列（s\_runQueue指针指向）中查找，找优先级最高的。

5.3.6、GeekOS进程调度处理过程

进程调度是在时钟中断处理函数Time\_Interrupt\_Handle(/src/geekos/timer.c)内实现的。进程的Kernel\_Thread结构中有一个numTicks变量，在进程对象初始化时被初始化为零，之后每次时钟中断处理，进程的该变量都会加1，然后系统会检查进程执行的时间是否超过了系统规定的时间片g\_Quantum，如果超过，说明当前进程时间片已用完，系统应调度新的进程运行，于是将变量g\_needReschedule置为true，用以标志需要重新调度新进程运行。

在时钟中断处理函数返回到Handle\_Interrupt后检查g\_needReschedule变量，如果为true，就调用Make\_Runnable函数（kthread.c），将当前运行进程放入准备运行进程队列s\_runQueue；之后再调用Get\_Next\_Runnable函数（kthread.c）找到优先级最高的进程；最后返回Handle\_Interrupt将g\_needReschedule返回为false，并切换到新进程运行。

static void Timer\_Interrupt\_Handler(struct Interrupt\_State\* state)

{

int i;

struct Kernel\_Thread\* current = g\_currentThread;

Begin\_IRQ(state);

/\* Update global and per-thread number of ticks \*/

++g\_numTicks;

++current->numTicks;

/\* update timer events \*/

for (i=0; i < timeEventCount; i++) {

if (pendingTimerEvents[i].ticks == 0) {

if (timerDebug) Print("timer: event %d expired (%d ticks)\n",

pendingTimerEvents[i].id, pendingTimerEvents[i].origTicks);

(pendingTimerEvents[i].callBack)(pendingTimerEvents[i].id);

} else {

pendingTimerEvents[i].ticks--;

}

}

/\*

\* If thread has been running for an entire quantum,

\* inform the interrupt return code that we want

\* to choose a new thread.

\*/

if (current->numTicks >= g\_Quantum) {

g\_needReschedule = true;

/\*

\* The current process is moved to a lower priority queue,

\* since it consumed a full quantum.

\*/

if (current->currentReadyQueue < (MAX\_QUEUE\_LEVEL - 1)) {

/\*Print("process %d moved to ready queue %d\n", current->pid, current->currentReadyQueue); \*/

current->currentReadyQueue++;

}

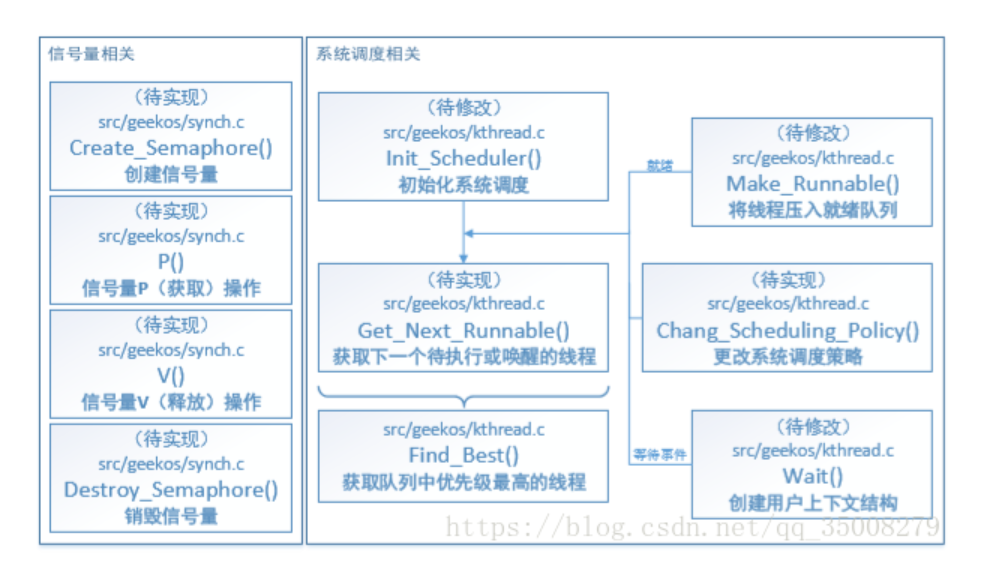
}

End\_IRQ(state);

}

5.4、项目设计原理

设计流程图：

图 5.1 设计流程图

本项目主要实现多级反馈队列(MLF)和轮询(RR)调度算法以及信号量的相关操作， 需要我们填写 syscall.c 和 kthread.c 等不同文件中的多个函数。 在之前的项目中，GeekOS 使用的系统调度算法均为轮询调度算法，因此在此项目 中，我们需要实现 MLF 调度算法的相关操作以及 RR 与 MLF 两种算法之间的队列转换 算法。GeekOS 中，MLF 算法的规则描述为：进程就绪队列共分为 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（此处为 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）， 因此，CPU 密集型的进程最终会 被放到最低优先级的队列中；如果一个进程被阻塞（blocked），它的队列优先级就会提 升一个等级，直到被阻塞三次后达到最高优先级队列（Q0）。

在 MLF 策略与 RR 策略的进程队列转换时，GeekOS 给出的规则如下：MLF->RR 时，将 Q1-Q3 队列中所有进程转移至 Q0 队列，然后按照优先级从高到低的顺序重新排 序；RR->MLF 时，只需将原本位于 Q0 队列中的 Idle（空闲）进程转移至 Q3 队列，其 他进程无需再做修改。在实际编写代码的时候，由于 GeekOS 提前实现了从队列中获取 最高优先级进程的函数 Find\_Best()，因此在 MLF->RR 时只需要将所有队列中进程移至 Q0 队列中即可。

为实现四级队列，设计中首先要修改s\_runQueue（src/geekos/kthreas.c）的定义，从原来一个结构体改为一个4元素的结构体数组，每一个结构体元素用于存放一个优先级队列的队首指针。

GeekOS系统将Idle进程始终放在优先级为3的进程队列末尾，且不允许移动到其它队列，以保证进程调度时一定能找到进程投入运行。

当实现四级反馈队列调度策略后，GeekOS系统就拥有2种进程调度策略，即单队列时间片轮转调度策略和四级反馈队列调度策略。到底采用哪种，是通过系统调用Sys\_SetSchedulingPolicy（src/geekos/syscall.c）实现的，函数原型：

static int Sys\_SetSchedulingPolicy(struct Interrupt\_State\* state);

参数state是一个结构体，其中，state->ebx成员用于指定调度策略，值为0代表系统采用时间片轮转调度策略，值为1则代表系统采用四级反馈队列调度策略，若取其他值则出错。state->ecx成员用于记录相应调度策略下的时间片长度。

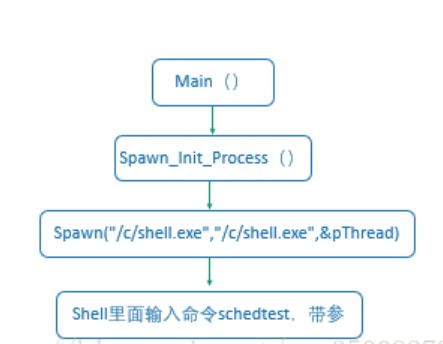
5.5、多级反馈策略测试运行流程

图 4.2 运行流程

schedtest里面参数相关代码：

if (argc == 3) {

if (!strcmp(argv[1], "rr")) {

policy = 0;//选择时间片轮转调度

} else if (!strcmp(argv[1], "mlf")) {

policy = 1;//选择多级反馈调度

} else {

Print("usage: %s [rr|mlf] <quantum>\n", argv[0]);

Exit(1);

}

} else {

Print("usage: %s [rr|mlf] <quantum>\n", argv[0]);

Exit(1);

}

quantum = atoi(argv[2]);//设置时间片长度

Set\_Scheduling\_Policy(policy, quantum);

接着调用Sys\_SetSchedulingPolicy（）（\src\geekos\syscall.c）设置调度策略：

static int Sys\_SetSchedulingPolicy(struct Interrupt\_State\* state)

{

/\* 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;

}

到Chang\_Scheduling\_Policy（）（\src\geekos\kthread.c）修改线程队列及系统相关变量，主要是 g\_curSchedulingPolicy 为当前调度策略， g\_Quantum 为时间片长度：

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\_curSchedulingPolicy)

{

/\* 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);

}

}

/\* 保存原来的调度策略 \*/

g\_preSchedulingPolicy = g\_curSchedulingPolicy;

/\* 将全局变量设置为对应的输入值 \*/

g\_curSchedulingPolicy = policy;

Print("g\_schedulingPolicy = %d\n", g\_curSchedulingPolicy);

}

g\_Quantum = quantum;

Print("g\_Quantum = %d\n", g\_Quantum);

return 0;

}

系统创建进程并运行后，每次时钟中断，进程的numTicks变量加1，这就用到了上面的Time\_Interrupt\_Handle(/src/geekos/timer.c)，通过进程所用时间片与用户输入时间片长度比较，设置g\_needReschedule变量，再调用Handle\_Interrupt进行进程调度。倘若超过时间片，g\_needReschedule为true，Handle\_Interrupt就会调用Get\_Next\_Runnable（\src\geekos\kthread.c），Get\_Next\_Runnable函数就是进程调度算法实现的地方：

struct Kernel\_Thread\* Get\_Next\_Runnable(void)

{

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

KASSERT(g\_curSchedulingPolicy == ROUND\_ROBIN ||

g\_curSchedulingPolicy == MULTILEVEL\_FEEDBACK);

/\* 查找下一个被调度的线程 \*/

struct Kernel\_Thread\* best = NULL;

if (g\_curSchedulingPolicy == 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);

return best;

/\*

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

\*/

}

系统调入新的进程运行后，时间片轮转结束，继续调度新的进程。

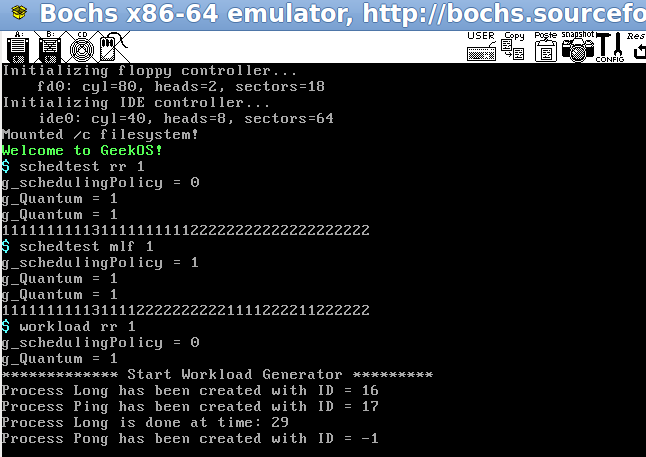
运行结果：

图 5.3 运行流程

从上面的测试结果来看，这两种调度算法都是基于时间片轮转，输出2和输出1的的数量基本一样多（多队列优先级调度算法的时间片都是一样的）。进程的周转时间还包括进程切换的时间。同样的调度算法，同样的时间片，程序执行的周转时间可能不一样（进程切换花费的时间可能不一样）。

5.6、信号量和PV操作

信号量操作：

Semaphore\_Create( )

Semaphore\_Acquire（P操作）

Semaphore\_Release（V操作）

Semaphore\_Destroy( )

Create\_Semaphore（）函数首先检查请求创建的这个信号量的名字是否存在，如果存在，那么就把这个线程加入到这个信号量所注册的线程链表上；如果不存在，则分配内存给新的信号量，清空它的线程队列，把当前的这个线程加入到它的线程队列中，设置注册线程数量为1，初始化信号量的名字，值和信号量的ID，并把这个信号量添加到信号量链表上，最后返回信号量的ID。

P操作Semaphore\_Acquire（）中，首先检查传入的信号量ID是否存在，如果存在，接着检查当前线程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了，就把信号量的值减去1，如果减去1后信号量的值小于0，那么就把当前线程放入这个信号量的等待队列上。

V操作Semaphore\_Release（）中，首先也是检查传入的信号量ID是否存在，如果存在，接着检查当前线程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了，那就把信号量的值加上1，如果加上1后信号量的值小于或等于0，则要把该信号量里等待队列上的一个线程唤醒。

Semaphore\_Destroy（）中，首先也是检查传入的信号量ID是否存在，如果存在，接着检查当前线程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了，就把该线程从这个信号量的注册的线程数组中删除，并把注册的线程数量减去1。如果这个信号量的注册线程为0了，则把这个信号量从信号量链表中删除，并释放它的内存。

运行结果：

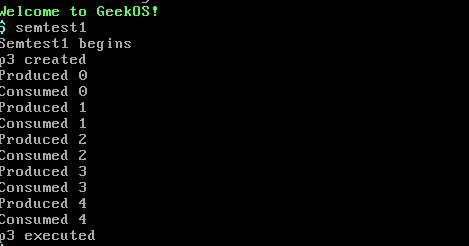
semtest1：

图 5.4 运行流程

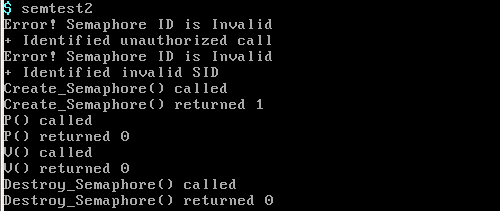
semtest2：

图 5.5 运行流程

从结果中我们可以看出，我们实现的信号量相关操作（创建、获取、释放和销毁） 都能够正确的在系统中执行，而且在 semtest2.exe 的测试中对于不正确的信号量操作（线 程对未授权信号量进行操作、无效信号量 ID 和销毁信号量后再对其进行获取操作）系 统也能够进行相应的处理和反馈，从此证明了信号量部分相关代码的正确性。

# 6、项目设计4

6.1、项目设计目的

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。

6.2、项目设计要求

6.2.1、完成函数

在<~/geekos-0.3.0/src/project4/src/geekos/paging.c>文件中编写代码完成以下函数：

（1） Init\_VM()(defined in )函数将建立一个初始的内存页目录和页表，并且安装一个页面出错处理函数。

（2） Init\_Paging()函数(定义在src/geekos/paging.c)初始化操作页面调度文件所需的所有数据结构。就如前面说到的，Get\_Paging\_Device()函数指定分页调度文件定位在哪一个设备和占用磁盘块的地址范围。

（3） Find\_Space\_On\_Paging\_File()函数应该在分页调度文件里面找到一个空闲的足够大的页空间。它将返回这个大块的索引，或者当没有合适的空间就返回-1。

（4）Free\_Space\_On\_Paging\_File()函数将释放由Find\_Space\_On\_Paging\_File()函数在分页调度文件里所分配的的磁盘块。

（5）Write\_To\_Paging\_File()函数将把存储在内存的一页数据写出到分页调度文件里。

（6）Read\_From\_Paging\_File()函数将读取分页调度文件里的一页数据到内存空间。

6.2.2、完成函数

在<~/geekos-0.3.0/src/project4/src/geekos/uservm.c>文件中编写代码完成以下函数：

（1） Destroy\_User\_Context()释放进程所占用的所有内存和其它资源。

（2） Load\_User\_Program()装载可执行文件到内存里，创建一个就绪的用户地址空间，功能类似于分段系统的实现。

（3） Copy\_From\_User()从一个用户缓冲区复制数据到一个内核缓冲区。

（4） Copy\_To\_User()从一个内核缓冲区复制数据到一个用户缓冲区。

（5）Switch\_To\_Address\_Space()利用它装载相应页目录和LDT来切换到一个用户地址空间。

6.3、项目设计原理

6.3.1、了解请求分页虚拟存储管理的基本原理

请求分页虚拟管理是将进程信息副本存放在外存中，当它被调度投入运行时，程序和数据没有全部装入内存，仅装入当前使用界面，进程执行过程中访问到不在内存的页面时，再由系统自动调入。用得较多的分页虚存管理是请求分页，当需要执行某条指令或使用某个数据而发现它们不在内存时产生缺页异常，系统从磁盘中把此指令或数据所在页面装入，这样做能够保证用不到的页面不会被装入内存。

  需要通过扩充页表项内容，增加驻留标志位等所需的各种控制位来发现并处理页面不在内存中的情况，通常页表项至少包含以下信息。



表 6.1 页表项

驻留标志位用来标志用来指出页面是否已装入内存。当访问一个页面时，如果驻留标志位为1，表示此页已在内存，可被正常访问；如果某页的驻留标志位为0，不能立即访问，产生缺页异常。缺页异常是由于发现当前访问页面不在内存时由硬件所产生的一种特殊中断信号，一条指令可能涉及多个页面，完全有可能在执行一条指令的过程中产生多次缺页异常，但最多产生缺页中断信号6次。

  在请求分页虚存管理中，页表存放的是把逻辑地址转换成物理地址时硬件所需要的信息，其作用有获得页框号以实现虚实地址转换、设置各种访问控制位、对页面信息进行保护等。

6.3.2、请求分页虚实地址转换过程

虚地址–>物理地址过程：将逻辑地址分解为页号+页内地址，先根据页号查快表TLB，若（1）快表命中，则立即送出页框号并与页内地址拼接成物理地址，然后进行访问权限检查，如获通过，进程就可以访问物理地址。若（2）快表不命中，则通过页号查页表，若页表命中则说明访问页面已在内存中，可送出页框号，并与页内地址拼接成物理地址，然后进行访问权限检查，如获通过，进程就可以访问物理地址。若（3）快表不命中，页表显示缺页异常，则进行缺页异常处理。根据页号查外页号，找到磁盘物理地址，查看内存是否有空闲页框，有则分配一个，否则按照替换算法选择淘汰页面。简化过程如下：

（1）快表命中；

（2）快表不命中，页表命中；

（3）快表不命中，页表显示缺页，搜索外页表，访问磁盘。

求数据的平均访问时间的结果分别如下：

（1）t=内存访问时间+快表的时间开销；

（2）t=内存访问时间+快表时间开销+内存访问时间开销；

（3）t=内存访问时间+快表时间开销+内存访问时间开销+磁盘访问时间开销。

其中第一个内存访问时间是取数据所必须花费的时间。

  为了实现分页存储系统的地址转换机制，系统增如了个新的寄存器CR3作为指向当前页目录的指针。这样,从线性地址到物现地址的映射过程为:

（1）从CR3取得页目录的基地址；

（2）以线性地址中的页目录位段为下标，在目录中取得相应页表的基地址；

（3）以线性地址中的页表位段为下标，在所得到的页表中取得相应的页面描述项；

（4）将页面描述项中给出的页面基地址与线性地址中的页内偏移位段相加得到物理地址。

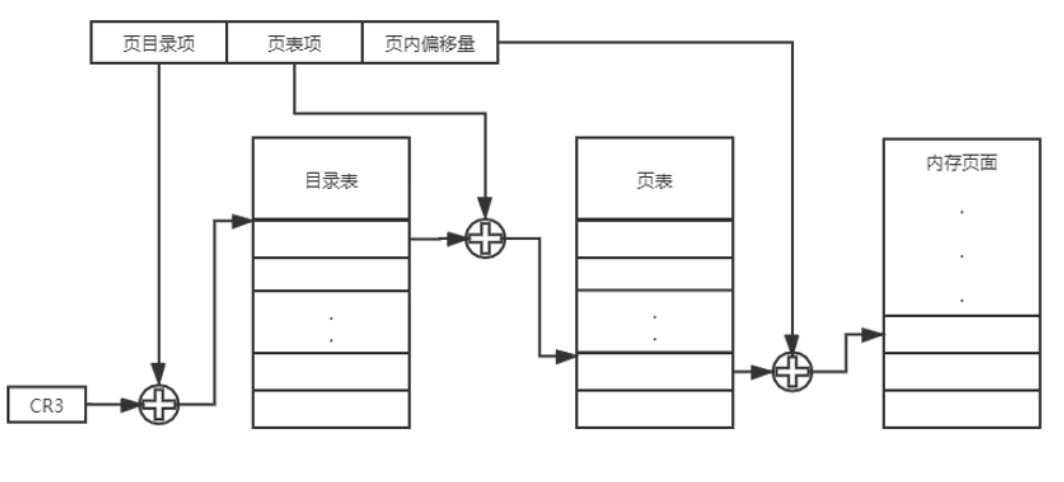
上述映射过程可用图只管地表示，具体如下图所示。

图 6.2 映射过程图

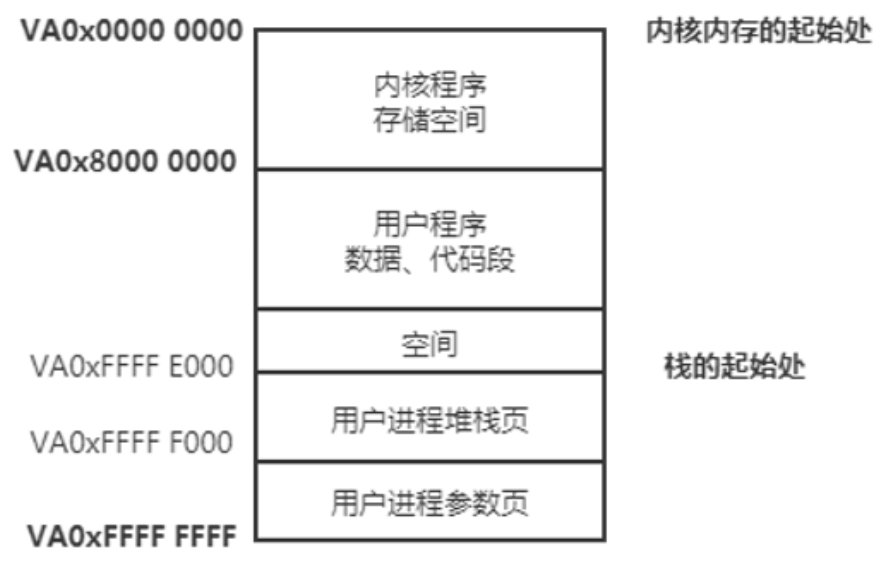
在GeekOS中，所有内核级进程共享一个页表，而每个用户级进程都有各自的页表。当系统需要运行某个进程时，就把该进程对应的页表调入内存，并使之驻留在内存中，这样就可以运行该用户级线程。此外，用户模式进程的页表也包含访问内核模式内存的入口。GeekOS的内存布局如下图所示。

图 6.3 GeekOS的内存布局

通过编写一个初始化页表和允许在处理器中使用分页模式的函数来为内核级进程创建一个页目录和页表入口，这个函数就是<project4\src\geekos\paging.c>中的Init\_ VM函数。在<paging.c>的Init \_VM的Hints (提示)中，用户可以看到此函数的功能主要有以下三个：

  (1)建立内核页目录表和页表；

  (2)调用Enable\_Paging函数使分页机制有效；

  (3)加入一个缺页中断处理程序，并注册其中断号为14。

  用户设计的缺页中断处理程序应该能够认识到此页在page file中是存在的，并将其从磁盘读进内存。而当用户将一页从磁盘调入内存时，需要释放这一页占用的空间。

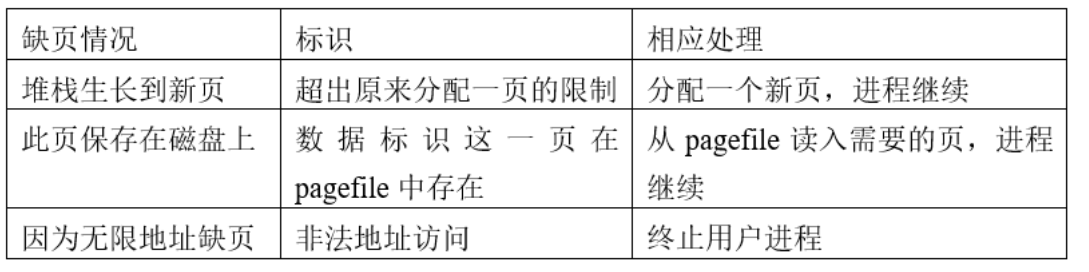
  缺页中断处理程序应做的工作如下表所示：

图 6.4 缺页中断处理程序工作表

6.4、项目具体实现

操作系统将需要在磁盘设备上创建一个page file文件暂时保存从内存中替换出去的页，实现一个类LRU算法在内存中选取一个替换页把它写到磁盘的page file文件中，然后根据表4.1 缺页处理表进行缺页中断处理。

  在“/src/geekos/mem.c”文件中，已经定义了一个函数Alloc\_Pageable\_Page实现交换一页到磁盘的操作，具体执行步骤如下：

  (1)调用mem.c文件中已经实现的Find\_Page\_To\_Page\_Out函数来确定要替换的页（这个函数依赖于页数据结构中的clock域）；

  (2)调用paging.c文件中已经实现的Find\_Space\_On\_Paging\_File函数在page file中找到空闲的存储空间；

  (3)调用paging.c文件中已经实现的Write\_To\_Paging\_File函数把被替换的页写到page file文件中；

  (4)修改页表的相应表项，清除页存在的标志，标识为此页在内存为不存在；

  (5)修改页表项的页基地址为包含这一页的第一个磁盘块号；

  (6)修改页表项的kernelInfo位标识为KINFO\_PAGE\_ON\_DISK状态（标识这一页是在磁盘上存在，而不是没有效）；

  (7)调用lowlevel.asm文件中已经实现的Flush\_TLB来刷新TLB。

  在Ubuntu9.04的终端中编译GeekOS项目0，步骤如下：

  (1)进入geekos-0.3.0/src/project4/build目录；

  (2)执行 make depend；

  (3)执行 make，成功之后在build 目录下生成软盘镜像文件fd.img文件和diskc.img文件；

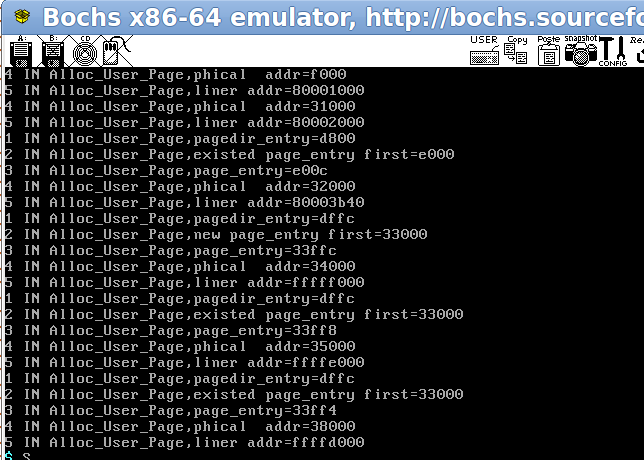
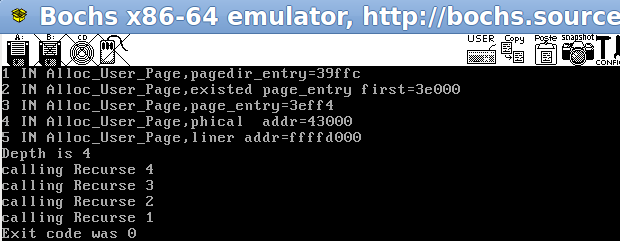
****  (4)在build目录下启动执行bochs，成功后，运行结果如下：

图 6.5 运行结果

由图Project4运行截图所示，当前系统有进程号为1~5的进程正在运行，进程1打印当前页目录表的入口起始地址，进程2打印当前页表当前起始地址，进程3打印当前所在页表位置，进程4打印物理地址起始地址，进程5打印线性地址，再运行一次进程4打印当前线性地址对应物理地址。

图 6.6 指令rec 4运行结果

输入命令rec 4后，可以得到如图迭代递归下的 Project4运行截图所示结果。由结果可以看出，当前进程队列下共有7个进程，进程1显示出当前页目录表入口为39ffc，进程2显示已存在页表的入口地址为3e000，进程3显示该线性地址空间下对应查找到的页表下标地址为3eff4，进程4显示对应物理地址为43000，进程5显示当前线性地址为ffffd000；最后输出此次迭代递归的搜索深度为4，在这个过程中一共递归调用过4次进程，分别是调用递归进程4、进程3、进程2，以及进程1，最后结束递归调用，程序运行结束。

# 参考文献

1. 李志欣, 魏海洋, 黄飞成, 等. 结合视觉特征和场景语义的图像描述生成[J]. 计算机学报, 2020, 43(9): 1624-1640.
2. 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.
3. 周飞燕,金林鹏,董军.卷积神经网络研究综述[J].计算机学报, 2017, 40(06):1229-1251.
4. 黄廷辉, 王宇英, 崔更中. "操作系统"课程实践教学的研究与实践[J]. 计算机教育, 2007, 000(022):18-20.
5. 赵艳红, 沈峰, 段汉根. 基于应用型本科的操作系统实验及实践教学的研究[J]. 洛阳师范学院学报, 2010, 29(5):3.