

**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** | ： |  |
| **组长** | ： | 刘艺鹏 |
| **成员** | ： | 1900300922刘艺鹏 |
|  |  | 1900300933吴伟铭 |
| **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 |
| **职称** | ： |  |

2022年 5 月 20 日

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（%） |
|  |  | |  |  |  |
|  |  | |  |  |  |
|  |  | |  |  |  |
|  |  | |  |  |  |

摘 要

GeekOS是一个基于X86架构的PC上运行的微操作系统内核，由美国马理兰大学

的教师开发，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。出于教学目的，这个系统内核设计简单，却又兼备实用性，它可以

运行在真正的X86 PC硬件平台。作为一个课程设计平台，GeekOS由一个基本的操作系统内核作为基础，提供了操作系统与硬件之间的所有必备接口，实现了系统引导、实模式到保护模式的转换、中断调用及异常处理、基于段式的内存管理、FIFO进程调度算法以及内核进程、基本的输入输出（键盘作为输入设备，显示器作为输出设备），以及一个用于存放用户程序的只读文件系统PFAT。此课设完成了GeekOS系统环境调试及编译、内核级线程设计及实现、用户级进程的动态创建与执行、进程调度算法与信号量功能、分页存储管理机制和GOSFS文件系统，实现了操作系统的基本操作。

关键词：操作系统、进程创建、进程调度、分页存储

目 录

**[1 绪论 3](#_Toc2968)**

**[2 Project 0：GeekOS系统环境调试及编译 4](#_Toc28514)**

[2.1 Project 0设计目的 4](#_Toc30946)

[2.2 Project 0原理分析 4](#_Toc11768)

[2.3 Project 0运行结果分析 5](#_Toc2407)

**[3 Project 1：内核级线程设计及实现 6](#_Toc30903)**

[3.1 Project 1设计目的 6](#_Toc4318)

[3.2 Project 1原理分析 6](#_Toc5462)

[3.3 Project 1运行结果分析 8](#_Toc25492)

**[4 Project 2：用户级线程的动态创建与执行 9](#_Toc24214)**

[4.1 Project 2设计目的 9](#_Toc4683)

[4.2 Project 2原理分析 9](#_Toc18221)

[4.3 Project 2运行结果分析 11](#_Toc15856)

**[5 Project 3：进程调度算法与信号量功能 12](#_Toc18721)**

[5.1 Project 3设计目的 12](#_Toc2583)

[5.2 Project 3原理分析 12](#_Toc19656)

[5.3 Project 3运行结果分析 14](#_Toc11371)

**[6 Project 4：分页存储管理机制 16](#_Toc11534)**

[6.1 Project 4设计目的 16](#_Toc18538)

[6.2 Project 4原理分析 16](#_Toc27319)

[6.3 Project 4运行结果分析 18](#_Toc546)

**[参考文献 19](#_Toc28321)**

# 

# 1 绪论

操作系统是管理[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA" \t "_blank)[硬件](https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6" \t "_blank)与[软件](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6" \t "_blank)资源的[计算机程序](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A8%8B%E5%BA%8F" \t "_blank)。操作系统需要处理如管理与配置[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98" \t "_blank)、决定[系统资源](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E8%B5%84%E6%BA%90/974435" \t "_blank)供需的优先次序、控制[输入设备](https://baike.baidu.com/item/%E8%BE%93%E5%85%A5%E8%AE%BE%E5%A4%87/10823368" \t "_blank)与[输出设备](https://baike.baidu.com/item/%E8%BE%93%E5%87%BA%E8%AE%BE%E5%A4%87/10823333" \t "_blank)、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 课程设计背景

1.1.1 GeeKOS教学操作系统

GeekOS是一个基于X86架构的PC上运行的微操作系统内核，由美国马理兰大学

的教师开发，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。出于教学目的，这个系统内核设计简单，却又兼备实用性，它可以

运行在真正的X86 PC硬件平台。

1.1.2 GeeKOS设计任务概述

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-project4项目，运行Bochs模拟器，查验项目结果。

1.1.3 GeeKOS实验环境

在此实验中，要使用Bochs来模拟一台计算机的硬件，Bochs是一个x86硬件平台的开源模拟器，可以用来模拟各种硬件的配置。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，通过改变配置，可以指定使用的CPU以及内存大小等。

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。

# 2 Project 0：GeekOS系统环境调试及编译

2.1 Project 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 Project 0原理分析

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键盘IO处理和内核线程的建立两部分内容。

首先要设计一个处理键盘事件的函数。经过阅读 keyboard.c 中的代码之后，发现

其中的Keyboard\_Interrupt\_Handler()函数是处理键盘中断的函数，此函数在main.c文件的main()函数中调用的Init\_Keyboard()函数中被注册，操作系统会在检测到发生键盘中断时调用此函数。在Keyboard\_Interrupt\_Handler()函数中，会根据操作系统传入的参数，将一个描述当前键盘的按键的情况的参数keycode加入到按键缓冲区中，此参数的0到11位描述了按下了哪一个按键，第12到第14位分别描述了是否按下Shift、Alt和Ctrl按键，第15位描述了是按下按键还是送开按键，所以一个keycode参数就可以描述按键的所有状态。调用Wait\_For\_Key() 函数就可以获得由Keyboard\_Interrupt\_Handler()函数放入缓冲区的keycode，若是按键缓冲区为空，则会等待，直到不为空。了解了以上原理后，就要在处理键盘事件的函数中写一个循环，在循环中调用Wait\_For\_Key() 函数，当返回的keycode与0x8000进行与操作后若为0，表明是按下按键，就输出此时的keycode。若keycode与16484相等，就表明同时按下了ctrl和d，就跳出循环，结束进程。

然后就要建立内核进程，要使用的是Start\_Kernel\_Thread()函数，此函数第一个参数就是创建的进程要运行的函数，第二个参数是所传入函数的参数，第三个参数是进程的优先级，最后一个参数表明是否有子进程。所以可以在main.c文件的main()函数中调用Start\_Kernel\_Thread()函数，并将第一步创建的处理键盘事件的函数传入此函数中即可。

2.3 Project 0运行结果分析

当开始运行GeekOS后，按下按键，会打印相应的按键，如下图所示：

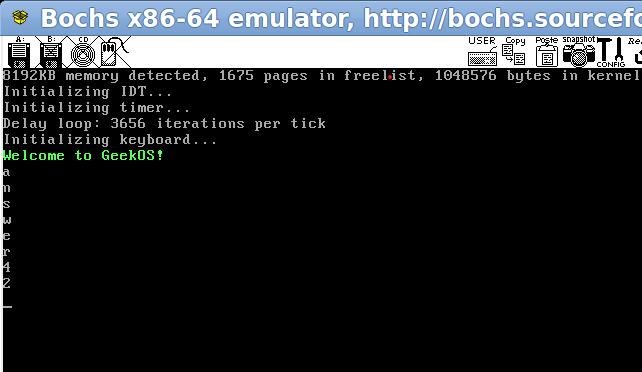


图2.1打印按下的按键

当按下ctrl+d后，会打印process end! 并结束进程，如下图所示：

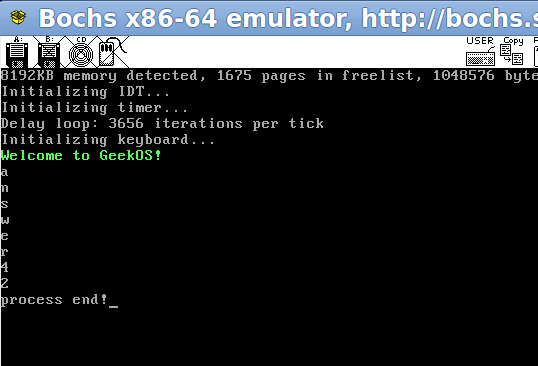


图2.2按下ctrl+d结束进程

# 3 Project 1：内核级线程设计及实现

3.1 Project 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。

3.2 Project 1原理分析

在此项目中，要解析已经读取到内存中的ELF文件，ELF文件的格式如下图：

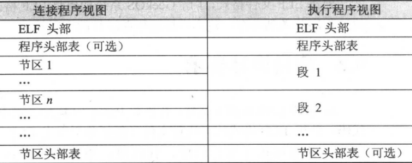


图3.1ELF文件格式

定义在elf.h中的表示ELF头部的结构体如下：

typedef struct {

unsigned char ident[16]; // 目标文件标识

unsigned short type; // 目标文件类型

unsigned short machine; // 机器类型

unsigned int version; // 目标文件版本

unsigned int entry; // 程序入口虚地址

unsigned int phoff; // 程序头部表的地址

unsigned int sphoff; // 节区头部表的地址

unsigned int flags; // 处理器相关的标志信息

unsigned short ehsize; // ELF头部的大小

unsigned short phentsize; // 程序头部表的表项大小

unsigned short phnum; // 程序头部表的表项数量

unsigned short shentsize; // 节区头部表的表项大小

unsigned short shnum; // 节区头部表的表项数量

unsigned short shstrndx;

} elfHeader;

定义在elf.h中的表示ELF文件的程序头部表表项的结构体如下：

typedef struct {

unsigned int type; // 段类型

unsigned int offset; // 从文件头到该字段第一个字节的偏移

unsigned int vaddr; // 段被放到用户内存中的起始虚拟地址

unsigned int paddr; // 仅用于与物理内存相关的系统中

unsigned int fileSize; // 段在文件映像中所占的字节数

unsigned int memSize; // 段在内存映像中占用的字节数

unsigned int flags; // 与段相关的标志

unsigned int alignment; // 段在文件中的内存中如何对齐

} programHeader;

定义在elf.h中的保存ELF文件的程序段数据的结构体如下：

struct Exe\_Segment {

ulong\_t offsetInFile; // 在ELF文件中的偏移地址

ulong\_t lengthInFile; // 在ELF文件中的长度

ulong\_t startAddress; // 在用户内存中的地址

ulong\_t sizeInMemory; // 在用户内存长度

int protFlags; // 描述了段的状态

};

定义在elf.h中的用于保存解析到的结构体如下：

struct Exe\_Format {

struct Exe\_Segment segmentList[EXE\_MAX\_SEGMENTS]; // 保存解析到的段

int numSegments; // 保存段的数量

ulong\_t entryAddr; // 保存代码段的入口地址

};

了解了以上信息后，就可以解析ELF文件，首先直接将保存ELF文件数据的exeFileData变量强制转换成elfHeader结构体的指针，这是因为ELF文件头就是存在在ELF文件的最前端，所以可以直接转换。然后从中读取段的个数和代码入口地址，然后获取到程序头部表的入口地址并用一个programHeader类型的指针指向这个地址，利用一个循环，依次读出每个程序头部表表项的数据，保存到segmentList中。

3.3 Project 1运行结果分析

代码正确运行的结果如下：

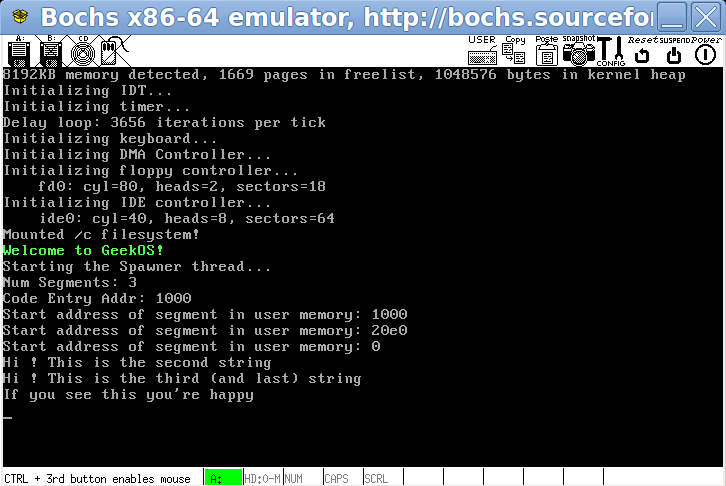


图3.2 ELF文件成功解析

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，同时，输出了部分要保存的数据，比如段的数量、代码段的入口地址和每个段在用户内存中的地址。

# 4 Project 2：用户级线程的动态创建与执行

4.1 Project 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。

4.2 Project 2原理分析

用户通过shell.exe来进行命令的输入，若用户要创建一个进程，则会使用一个名为Sys\_Spawn()的系统调用，在此系统调用中会调用Spawn()函数创建一个用户进程。过程如下：

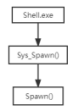


图4.1 通过shell创建用户进程

Spawn()函数的执行过程如下如所示：

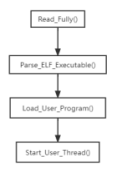


图4.2 Spawn()函数

在此函数中会先调用 Read\_Fully() 函数在文件系统中将要执行的可执行文件读取到内存中，然后交给Project 1中完成的Parse\_ELF\_Executable()函数进行解析，之后将解析好的数据传递给Load\_User\_Program() 函数，此函数会先创建一个User\_Context对象，并把解析到数据复制到此User\_Context对象指向的用户进程内存空间中，最后，将创建好的User\_Context对象传递到Start\_User\_Thread()函数中启动用户进程。此项目的重点就是完成Load\_User\_Program() 函数和Start\_User\_Thread()函数。

在Load\_User\_Program() 函数中，会先根据解析出来的ELF文件的数据，即Exe\_Format对象来计算要分配的内存的大小，用户进程的内存大小分为段、进程堆栈和参数块，与此同时要获取参数块的地址。之后将计算好的内存大小传入Create\_User\_Context()函数中，创建一个User\_Context对象，创建好后，将Exe\_Format对象中的段数据，根据其所储存的每个段的内存和文件中的地址，复制到User\_Context对象的memory字段指向的内存空间。最后利用 Format\_Argument\_Block() 函数格式化参数快，将User\_Context对象的代码段指针、数据块指针和堆栈指针即可。

在Create\_User\_Context()函数中，首先会创建一个User\_Context对象，User\_Context对象有一个memory字段，用于指向用户进程内存的起始地址，Load\_User\_Program() 函数中计算好的内存大小是给此字段指向的地址分配内存空间用的。之后会使用Allocate\_Segment\_Descriptor()函数在GDT中找到一个空余的描述符，紧接着使用Init\_LDT\_Descriptor()函数初始化此描述符，之后就给数据段和代码段创建相应的描述符和选择子，将描述符放到User\_Context对象的 LDT 表中，选择子放到对应的字段中。

对于描述符和选择子的关系，在下图中可以清晰地看出：

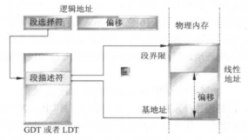


图4.3 描述符与选择子

GDT中的描述符保存的是每一个用户进程的内存的段地址，而每一个用户进程的LDT中的描述符中保存的是数据段和代码段的段地址。

在Load\_User\_Program() 函数中创建好User\_Context对象后，将此对象传递到Start\_User\_Thread()函数函数中，此函数的执行过程如下图：

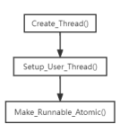


图4.4 Start\_User\_Thread()函数

此函数会首先利用Create\_Thread()函数创建一个Kernel\_Thread对象，然后利用Setup\_User\_Thread()函数将使 Kernel\_Thread 对象的 userContext指针指向刚创建的 User\_Context对象，并和初始化 Kernel\_Thread 对象的堆栈页。最后执行Make\_Runnable\_Atomic()函数将新创建的进程加入就绪进程队列。

4.3 Project 2运行结果分析

Project 2的代码运行结果如下：

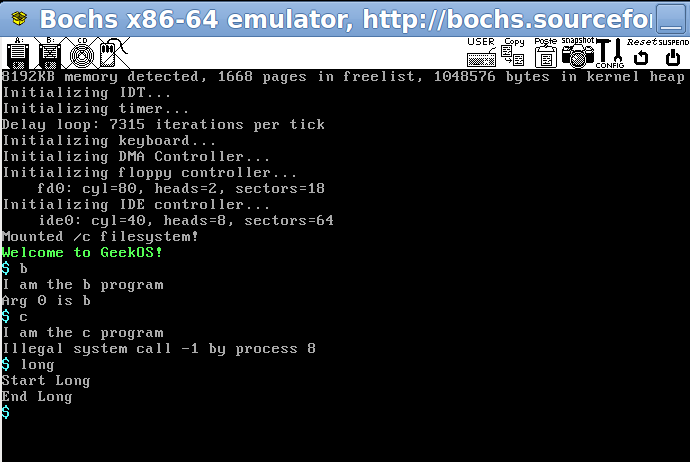


图4.5 成功创建用户进程

在图中可以看出，shell.exe用户进程成功运行，并利用此进程成功创建了三个用户进程。

# 5 Project 3：进程调度算法与信号量功能

5.1 Project 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( )。

5.2 Project 3原理分析

5.2.1 GeeKOS进程调度过程

当一个进程的时间片用完时，GeekOS调度进程的过程如下：

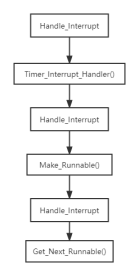


图5.1 GeekOS调度进程

Handle\_Interrupt函数是定义在lowlevel.asm中的中断处理函数，GeekOS会在系统出现中断的时候调用这个函数进行处理，GeekOS每过一段时间就会产生一个时钟中断，

此时就会调用Handle\_Interrupt函数。Handle\_Interrupt函数会先在中断向量表中查找对应的中断处理函数，时钟中断的处理函数是Timer\_Interrupt\_Handler()函数。

进程的KemeLThread结构中有一个numTicks变量，numTicks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时，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，如果超过了这个值，就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，用以标志系统需要重新调度新进程运行，在函数的最后，若当前进程的优先级不是最低的3，且不是idle进程，就将进程优先级减少一级。

在时钟中断处理函数返回到Handle lnterrupt后检查g\_needReschedule变量，如果变量值为true，就调用Make\_Runnable函数，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函数找到优先级最高的进程作为当前运行进程，并将其从准备运行进程队列中取出。最后返回Handle lnterrupt将g\_needReschedule改回为false，并切换到新进程运行。

5.2.2 四级反馈队列原理及创建

首先，新创建的进程被放入优先级最高的准备运行队列，即优先级为0的准备运行队列。若优先级为0的进程被调度后，在给定时间片内无法完成，那么进程就被移到下一优先级队列的尾部（即优先级为1的准备运行队列），以此类推，直到进程被放到优先级为3的队列为止（在Timer\_Interrupt\_Handler()函数中实现）。因此，要求运行时间较长的进程最终会被放入到优先级为3的准备运行队列。若进程被阻塞，每经过一个时间片，进程优先级都将增加1，所以当进程阻塞达连续的3个时间片后，又将升到最高优先级（在Timer\_Interrupt\_Handler()函数中实现）。进程调度总是优先调度优先级高的进程运行，当优先级为0的进程队列为空时，进程调度从优先级为1的队列选择进程调度（在Get\_Next\_Runnable()函数中实现）。系统中的空闲进程Idle，在四级反馈队列调度中，该进程应始终放在优先级为3的进程队列尾部，以便系统中没有其他可调度进程时就运行此进程。

要创建四级反馈队列就要先将原来的s\_runQueue变量由原来的指向一个进程队列变为指向一个保存4个进程队列的数组。Chang\_Scheduling\_Policy()函数可以切换系统所用的进程调度算法，当调用此函数切换到mlf调度算法时，就将idle进程从s\_runQueue[0]移动到s\_runQueue[3]，然后设置指定的时间片大小即可。若要切换到切换到rr调度算法时，就将s\_runQueue[1]到s\_runQueue[3]的所有进程按照顺序移动到s\_runQueue[2]即可。

5.2.2 信号量PV操作具体分析

信号量的创建和销毁和P V操作均在syscall.c文件中实现。

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

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

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

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

5.3 Project 3运行结果分析

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用户进程和workload都需要传入两个参数，调度算法和时间片大小。

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

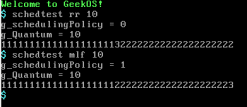


图5.2 调度算法测试

从测试中可以看出，rr调度算法与mlf调度算法的调度过程不同。

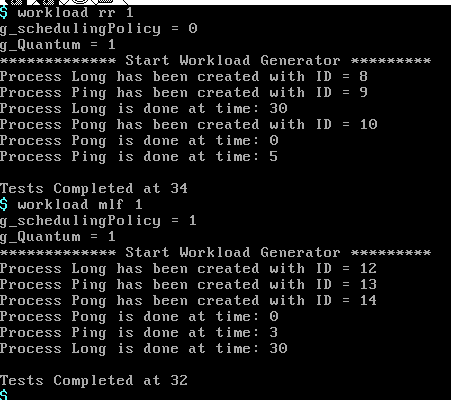


图5.3 调度算法压力测试

在此测试中可以看出使用mlf算法的压力测试的每一个进程的结束时间都小于等于rr算法的压力测试，而且完成测试的总时间也要小，所以mlf算法要强于rr算法。

接着就要进行信号量操作的测试，所需使用的是semtest、semtest1和semtest2用户进程。

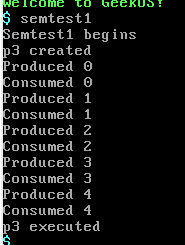
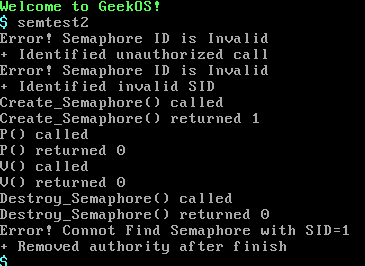
  

图5.4 semtest 图5.5 semtest1 图5.6 semtest2

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 Project 4：分页存储管理机制

6.1 Project 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。

6.2 Project 4原理分析

要使用分页存储管理内存，首先要在 main.c 中的 main() 函数中使用Init\_VM()函数初始化内核使用的内存，在此函数中，会先计算内存中有多少给页，然后给内核创建对应的 pde 和 pte，最后使用Enable\_Paging()函数开启分页存储的功能，并使用Install\_Interrupt\_Handler()函数注册使用的分页故障处理函数Page\_Fault\_Handler()。

要使用请求分页技术，就要在 main.c 中的 main() 函数中使用Init\_Paging()函数初始化磁盘上的虚拟内存，即创建一个bitmap，用于表示磁盘上的页是否被使用，用于管理虚拟内存。此项目中要完成的函数Find\_Space\_On\_Paging\_File()、Free\_Space\_On\_

Paging\_File()和Write\_To\_Paging\_File()都要使用此bitmap。

当用户创建进程时，也要使用内存，所以要对Load\_User\_Program()进行修改，使之能把ELF文件加载到分页存储管理的内存中。在此函数中，首先要给用户进程创建其 pde 和pte，然后将内核的 pde 的表项复制给此进程的 pde。因为在内存中前2G的空间为内核所使用，如下图所示：

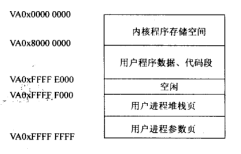


图6.1 GeekOS内存分布

所以内核进程使用的进程只能在2G之后的空间，要将代表这部分的页目录表的表项清零。然后就可以通过计算或者直接读取的方式获得用户进程中要放进去的数据的大小和地址，即ELF中的所有段、参数块和堆栈。根据获得的大小的地址，利用Alloc\_User\_Page()函数为这些数据分配页，之后将再利用Copy\_User\_Page()函数将数据复制到对应位置，最后将userContext剩余字段赋值即可，需要注意的是，堆栈是从高地址向低地址生长的，堆栈的指针要指向分配的页面的末尾。

在使用Alloc\_User\_Page()函数时，由于使用的数据的地址都是内存的真实的物理地址，所以要经过转化后才可以使用。GeekOS是32位的操作系统，所以地址都是32位，在分页存储管理中，线性地址的最高10位为 pde 的索引，中间 10 位为 pte 的索引，最低12位（4KB）为页内地址，所以可以通过线性地址获取要分配的pde表项和pte表项，最后利用Alloc\_Pageable\_Page()函数分配需要的页。

在使用Copy\_User\_Page()函数时，因为用于复制数据的memcpy()函数使用的都是物理地址，所以要把线性地址转换为物理地址，可以通过函数Lin\_To\_Phyaddr()来转换。此函数先利用线性地址获取pde表项和 pte表项，代表这两者的结构体如下：

typedef struct {

uint\_t present:1; //页在内存中是否存在

uint\_t flags:4; //页的访问权限

uint\_t accesed:1; //页是否被访问过

uint\_t reserved:1;

uint\_t largePages:1; //页表大小

uint\_t globalPage:1; //是否为全局页表

uint\_t kernelInfo:3; //内核页表

uint\_t pageTableBaseAddr:20; //页表在内存中的物理地址

} pde\_t;

typedef struct {

uint\_t present:1; //页在内存中是否存在

uint\_t flags:4; //页的访问权限

uint\_t accesed:1; //页是否被访问过

uint\_t dirty:1; //是否被修改过

uint\_t pteAttribute:1; //页大小

uint\_t globalPage:1; //是否位全局页

uint\_t kernelInfo:3; //内核页

uint\_t pageBaseAddr:20; //页在内存中的物理页号

} pte\_t;

可以看出，在pde表项的结构体中保存有页表的物理地址，在pte表项的结构体中保存有页的物理页号，所以就可以根据获取pde表项和 pte表项中的地址，再加上线性地址的低12位页内地址，就可以得到数据所在的物理地址。

实现请求分页技术时，要完善的函数是Page\_Fault\_Handler()用于处理页面错误中断。

系统会出现页面错误中断的情况如下表：

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 |

当遇到堆栈生长到新页时，就利用Alloc\_Pageable\_Page()函数在出现错误的地址上分配新的一页内存。当目标保存在磁盘上时，就先Alloc\_Pageable\_Page()函数分配一页内存，再使用Read\_From\_Paging\_File()函数从磁盘读取数据到刚才分配的页上。当出现非法地址访问时，就直接结束此用户进程。

6.3 Project 4运行结果分析

运行rec程序进行创建用户进程和堆栈生长到新页从而产生页面错误中断的测试，运行结果如下图：

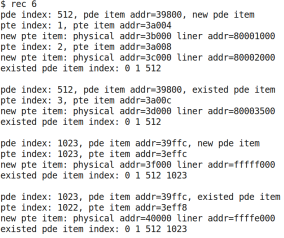
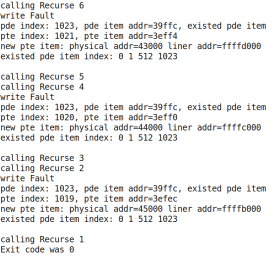
 

图6.2 rec 第一段输出 图6.3 rec 第二段输出

Depth is 6输出之前的都是创建用户进程的过程的情况输出，分为四段，前两段为ELF文件的段数据页面分配的情况，第三段为参数块页面分配的情况，第四段输出为堆栈页面分配的情况。因为整个系统的内存只有2048个页面，所以为内核分配初始化分页存储时（Init\_VM()函数），只是分配了两个pde表项。所以在为用户进程复制pde表项时就只是复制了两个。所以在给第一次分配页面内存时，因为pde表项的索引是512（2G内存之后才是用户进程的内存），就只能创建新的pde表项及其对应的pte表项和页面。第二次分配页面内存时，pde表项的索引还是512，就不需要分配新的pde表项，只需要分配对应的pte表项和页面即可。在第三次和第四次分配页面内存时，因为参数块和堆栈临近，就在一个pde表项（索引为1023）上，所以只需分配参数块的页面内存时需要创建新的pde表项，分配堆栈的页面内存时不需要，只需为各自分配 pte 表项即可。

Depth is 6输出之后都是用户进程运行期间的内存分配情况的输出和进程自身的输出。可以清楚的看到，大致两层递归就可以占满一个堆栈（堆栈用于存放函数的信息，每进入一层函数堆栈就将当前的函数的信息入栈，每退出一层函数，就将当前函数的信息出栈），第一个堆栈页面存放的是main函数和第一层递归函数的信息，之后没两层递归的函数信息占满一个堆栈页面。因为初始只为用户进程分配了一个堆栈页面，所以第二层递归时就出现了写错误，堆栈生长到新页，就要调用Alloc\_Pageable\_Page()函数分配新的堆栈页面，之后每两层递归就出现一次写错误，每次都要分配新的页面。从每分配新的堆栈页面时的pte表的索引直线变小可以看出，堆栈确实是从高地址向低地址生长的。

运行另一个用户进程rec1进行更大数量的递归，运行结果如下：

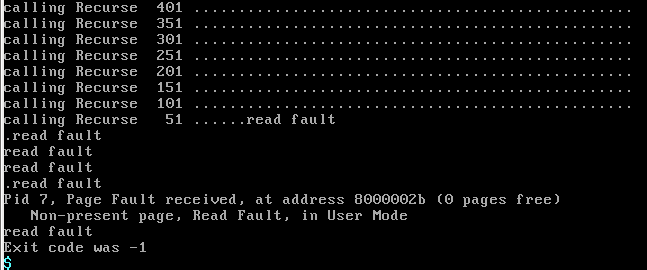


图6.3 读错误

可以看到系统先是被此用户进程占满了内存，出现了多次读错误，多次从磁盘中读取数据放到内存中。最后，磁盘上的虚拟内存也被用完，出现了非法地址访问的错误，进程最后终止。

# 

# 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.