
华中科技大学

本科生课程设计报告

课程设计名称: 《操作系统原理》课程设计

题 目: 研究 GEEKOS 小型开源操作系统原理

院 系: 软件学院

专业班级: 软工 1803 班

学 号: U201817044

姓 名: 王粟鹏

2020 年 05 月 30 日

目录

本科生课程设计报告.....	1
一、 任务概述.....	3
1. 设计要求:	3
2. 具体内容:	3
3. 选做内容:	3
二、 功能需求.....	4
1. Project0.....	4
2. Project1.....	4
3. Project2.....	5
三、 设计思路.....	6
1. Project0.....	6
2. Project1.....	7
3. Project2.....	8
四、 开发环境下载、安装和配置.....	13
1. 开发环境配置.....	13
2. 工程文件的目录结构和文件.....	14
五、 程序的难点或核心技术分析.....	15
1. 程序的难点以及解决办法.....	15
2. 核心技术分析.....	17
六、 运行和测试过程.....	17
1. Project0.....	17
2. Project1.....	19
3. Project2.....	20
七、 软件使用说明.....	22
八、 关于音视频录制情况.....	22
九、 编程参考网址.....	22
十、 同组同学列表.....	23

备注：此电子版、源代码以及录制的视频已经打包发给老师邮箱 oscource@163.com

一、任务概述

根据操作系统的相关原理，可以自己编写启动扇区的程序，启动特定的操作系统。用户可以自己开发简化的操作系统支持特定场合中的应用。本设计意图让学生研究一个小型开源的开源操作系统 GeekOS，通过编译，安装该操作系统，了解操作系统的编译过程，裸机上（采用虚拟机）安装和启动操作系统的配置过程，并重点研究 GeekOS 的实现过程。

1. 设计要求：

- （1）通过简化版本的开源操作系统熟悉操作系统的组成结构，尤其是核心结构；
- （2）熟悉一种操作系统的开发和编译环境；
- （3）熟悉操作系统的安装过程/配置过程
- （4）学会阅读英文专业文档，学会从零开始探索新课题的实现过程。

2. 具体内容：

下载 GeekOS 源代码，阅读器文档，编译该操作系统，并虚拟机环境下安装好 GeekOS 操作系统，并编写 3-5 个应用程序测试它提供的系统调用是否能够正常调用。

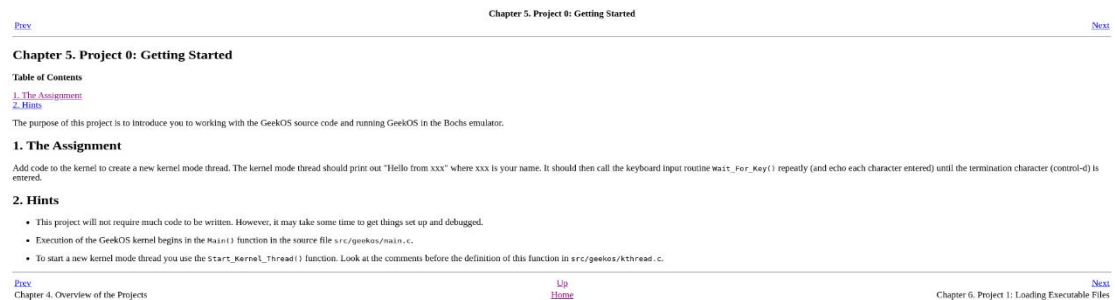
3. 选做内容：

将 PFAT 文件系统替换成 FAT16 或 FAT12 文件系统，完成 FAT16 或 FAT12 文件系统集成在 GeekOS 中，并重新编译和安装新的 GeekOS（虚拟的软盘或硬盘都背格式化为 FAT16 或 FAT12）。

二、功能需求

1. Project0

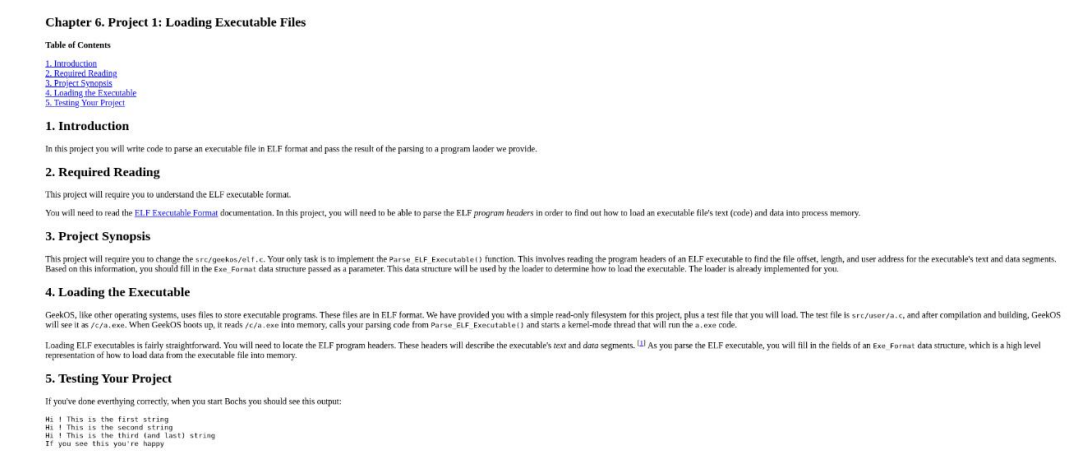
该部分程序为了测试 project0 当中的系统功能而设计。查阅 geekos 中的 readme 文档，可以得到关于 project0 部分的英文原版文档如下。



该部分要求使用者学会初步使用 Geekos 系统，给出的任务是通过调用 Wait_for_Key() 来实现键盘响应，通过系统调用将系统检测到的用户按键显示在显示屏上，如果用户按下“Ctrl+d”，则退出程序。

2. Project1

该项目是为了测试 project1 文件下的系统调用而设计。查阅 Geekos 中的 readme 文档可以得到如下部分。



这一部分主要是关于 elf 文件的内容。要求理解 elf 文件的结构，并需要修改 src/geekos/elf.c 中的 Parse_ELF_Executable()函数，实现对 elf 文件的分析，然后将分析的结果传递给系统提供的加载程序。

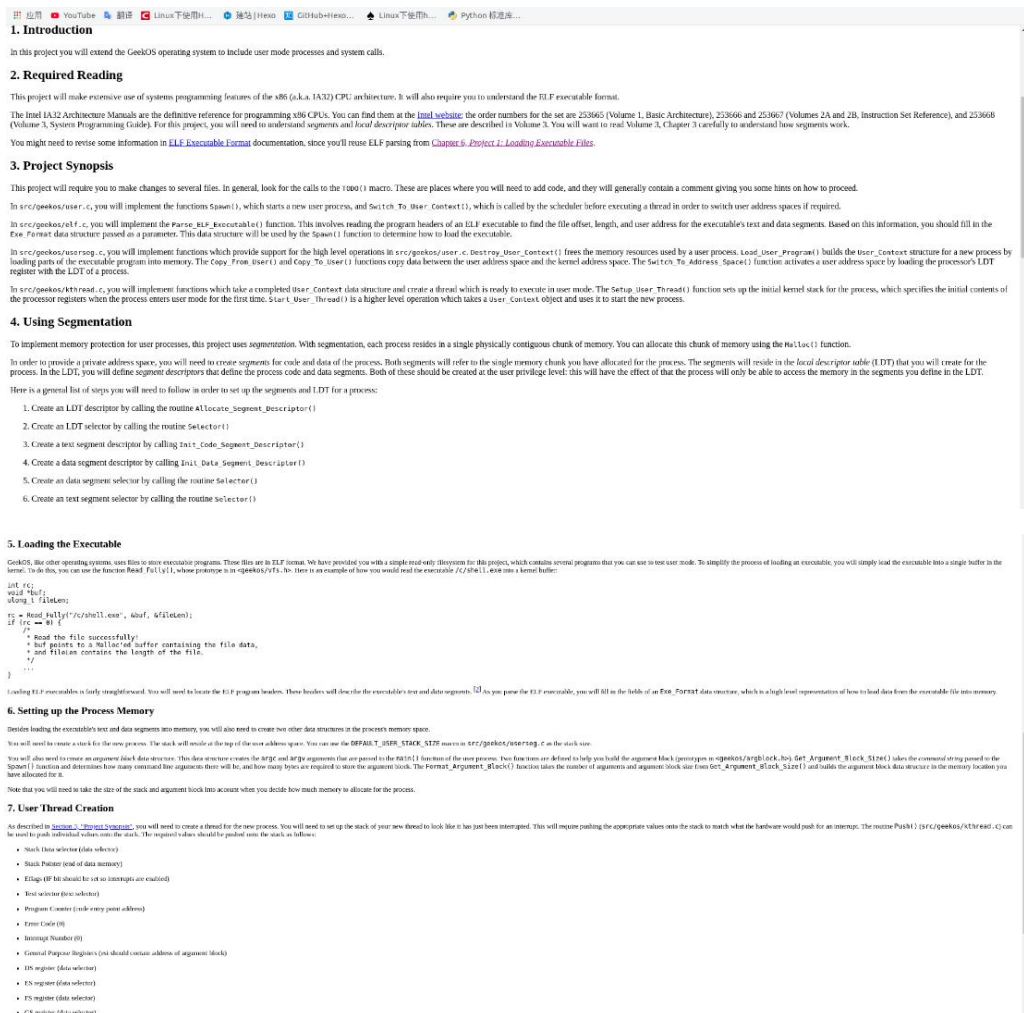
系统在该项目当中已经提供了一个示例文件 a.c，在经过汇编和构建之后，会生成 a.exe 文件，当 Geekos 启动时，会调用 Parse_ELF_Executable 并创建一个内核进程执行 a.exe。

该项目不是仅仅加载可执行程序，还需要分析该 elf 文件，得到 elf 的 program headers 地址，并根据 headers 找到可执行文件内容和数据段，然后将内容填充到 Exe_Format 数据结构当中，以此将数据从可执行文件加载到内存当中。

从文档可以看出，如果项目成功，会在内核输出最后的四个 string 语句。

3. Project2

该部分是为了测试 project2 中的功能而设计，查阅 Geekos 中的 readme 可以得到如下的英文文档：



这一部分的功能主要是扩充 Geekos 操作系统内核，使系统可以支持用户级进程的动态创建与执行，用户还可以为操作系统增添系统调用。

三、 设计思路

1. Project0

1.1 原理： 在该项目中，分为“创建内核进程”和“键盘响应”两部分。

内核进程通过/src/geekos/keyboard.c 中的 Start_Kernel_Thread()函数创建。该函数首先通过 Create_Thread()函数根据优先级创建一个进程，并将该内核进程返回到 kthread，如果 kthread != 0，则使用 Setup_Kernel_Thread()对创建的进程进行初始化，然后再使用 Make_Runnable_Atomic()函数将进程放到运行队列当中。Start_Kernel_Thread()函数体如下：

```
struct Kernel_Thread* kthread = Create_Thread(priority, detached);
if (kthread != 0) {
    /* ...
    Setup_Kernel_Thread(kthread, startFunc, arg);
    /* Atomically put the thread on the run queue. */
    Make_Runnable_Atomic(kthread);
}
return kthread;
```

键盘响应主要通过/src/geekos/keyboard.c 中的 Read_Key()函数实现，GeekOS 在/include/geekos/keyboard.h 给出了相应键位所对应的函数返回值。Read_Key()检测用户是否按下按键，如果有按键被按下，Read_Key()会将键值传递给参数 keycode。可以通过比较 keycode 和键值表，来判断并输出按下的按键。

```
bool Read_Key(Keycode* keycode)
{
    bool result, iflag;
    iflag = Begin_Int_Atomic();
    result = !Is_Queue_Empty();
    if (result) {
        *keycode = Dequeue_Keycode();
    }
    End_Int_Atomic(iflag);
    return result;
}
```

GeekOS 的启动函数是/src/geekos/main.c 中的 Main()。

1.2 设计和实现思路： 为了在 GeekOS 启动时执行我们的代码，需要在 main.c 当中编写一个 project0()函数，在 project0 当中通过调用 Read_Key()来实现我们的功能，然后在 Main()函数当中使用 Start_Kernel_Thread()为 project0()创建一个内核进程，执行我们的代码，实现键盘响应。

Project0()函数的设计思路：首先定义 Keycode 类型的变量 keycode 来存储键值，然后通

过 while(1)循环来实现功能的重复执行。在 while 循环体内, 首先通过 if(Read_Key(&keycode)) 检测是否有按键按下, 如果有按键被按下, 再进一步判断是不是弹起或者特殊键。如果不是以上两种情况, 则在进行判断是不是 Ctrl+d, 如果是则退出; 如果不是则输出按键字符。

2. Project1

2.1 原理:

- 1) 首先查看一下 project1 中函数的执行路径。



Main()函数是 project1 的启动函数, 启动之后首先进行一系列的准备工作, 然后调用 Spawn_Init_Process()函数, 该函数通过 Start_Kernel_Thread()创建一个 Spwaner()函数的内核线程, 然后执行 Spawner()函数。Spawner()会先调用 Read_Fully()来读取指定 elf 文件的数据和文件长度并存储到相应变量当中。然后 Spawner()再调用 Parase_ELF_Executable()来分析 elf 文件, 并将存储到变量中的数据填充到 exeFormat 当中。Spawn_Program()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, 然后根据 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, 最后通过 Trampoline 函数执行用户进程。

- 2) 下面是 ELF 文件的结构:

ELF 文件中数据的布局如下:

ELF header (Linking View)	ELF header (Execution View)
-----	-----
Program header table (Optional)	**Program header table**
Section 1	Segment 1
.....
Section n	Segment n
.....
Section header table	Section header table (Optional)

2.2 设计和实现思路:基本的过程原系统已经给出,所以只需要填充 Parse_ELF_Executable() 函数体即可。该函数的设计思路如下。

```
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的起始地址

exeFileLength——文件的长度

exeFormat——保存分析得到的 elf 文件信息的结构体指针

函数体:

首先创建一个 elf 头部结构体变量 elfHeader *ehdr, 令 ehdr 指向可执行文件的头部, 即(elfHeader*) exeFileData, 并将段的个数以及代码的入口地址读取出来, 填充到 exeFormat 当中。然后再创建 programHeader *phdr 指向头部表在文件中的位置, 方便后续的读取数据。最后通过循环遍历可执行文件中所有的段, 将段的起始地址、偏移量、长度、数据长度和标志位填充到 exeFormat 中的 segmentList 数组当中。

为了使系统正确的输出 a.c 中的局部变量信息, 除了 Parse_ELF_Executable() 函数之外, 还要修改 project1/src/geekos/lprog.c 文件, 将其中 Spawn_Program() 函数中的 virtSize 变量修改为全局变量。再修改 lprog.c 文件中的 Printrap_Handle() 函数, 根据 virtSize 和 state->eax 的比较, 对 msg 进行正确的赋值, 然后输出。

3. Project2

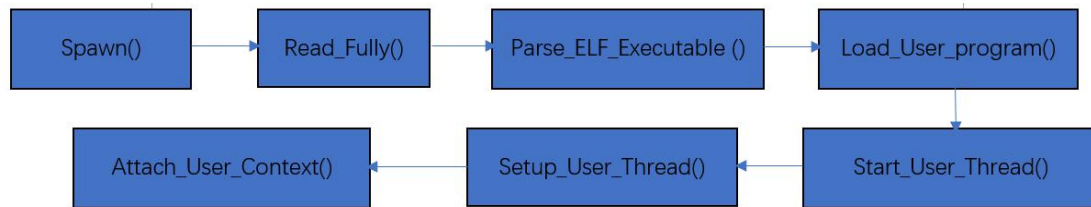
3.1 原理:

1) Geekos 进程状态及转化: GeekOS 系统最早创建的内核进程有 `Idle`、`Reaper` 和 `Main` 三个, 它们由 `Init_Scheduler` 函数创建: 最先初始化一个核态进程 `mainThread`, 并将其作为当前运行进程, 函数最后还调用了 `Start_Kernel_Thread` 函数创建了两个系统进程 `Idle` 和 `Reaper`。所以, 这三个进程是系统中最早存在的进程。

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。

3) 用户态进程的创建流程如下:



关于其中函数的部分说明如下:

a) `int Spawn(const char *program, const char *command, struct Kernel_Thread **pThread)`

相关参数如下:

- `program`: 要读入内存缓冲区的可执行文件
- `command`: 用户执行程序执行时的命令行字符串
- `pThread`: 存放刚刚创建进程的指针。

它的主要功能是:

- 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
- 调用 `Parse_ELF_Executable` 函数, 分析 ELF 格式文件。`Parse_ELF_Executable` 函数功能在 Project1 中已经实现, 在这里我们依然会给出自己的实现方案。
- 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。
- 调用 `Start_User_Thread` 函数创建一个进程并使其进入准备运行队列

b) `int Load_User_Program(char *exeFileData, ulong_t exeFileLength, struct Exe_Format *exeFormat, const char *command, struct User_Context **pUserContext)`

相关参数如下:

- `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, 包括可执行文件名称和其他参数
- `pUserContext`: 指向 `User_Context` 的指针, 本函数完成用户上下文初始化的对象

其主要功能如下:

- 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。

- 为用户程序分配内存空间, 并初始化。
- 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。
- 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。
- 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。
- 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。
- 将初始化完毕的 `User_Context` 指针赋予 `*pUserContext`, 返回 0 表示成功。

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。

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

3.2 设计和实现思路:

1) `Spawn()`函数: 在 `src/geekos/user.c` 文件中, 其功能为生成一个新的用户级进程。

首先调用 `Parse_ELF_Executable()` 函数实现对 ELF 文件的分析, 然后调用 `Load_User_Program()` 函数加载用户程序, 最后创建内核线程变量, 通过 `Start_User_Thread()` 开始执行用户进程。

2) `Switch_To_User_Context()`函数: 在 `src/geekos/user.c` 文件中, 其作用为调度程序在执行一个新的进程前调用其以切换用户地址空间。

首先获得最近使用过的 `User_Context* s_current_user_context`, 然后创建一个

User_Context* user_context 指向用户进程中的 user_context(), 并为准备切换的进程初始化, 此时要对 user_context 进行判断, 如果为 0 则代表此进程为核态, 不需要切换地址空间; 否则要通过调用 Switch_To_Address_Space()为用户态进程切换地址空间, 然后创建新进程的核心栈指针, 使用 Set_Kernel_Stack_Pointer()设置内核堆栈指针, 并将 i 虚拟的 user_context 保存到 s_current_user_context 当中。

3) Parse_ELF_Executable()函数在 project1 当中已做过介绍, 在此略过。

4) 高层操作支持函数: src/geekos/userseg.c 文件当中主要是实现一些对 'src/geekos/user.c' 高层操作支持的函数

- Create_User_Context()函数: 该函数用于创建并初始化一个用户上下文结构。对 context 下 memory 分配内存空间, 如果分配成功, 则将 user_context->memory 赋值, 然后创建一个 LDT 描述符, 初始化段描述符, 新建一个 LDT 选择子, 新建一个代码段描述符, 新建一个数据段描述符, 新建数据段和代码段选择子, 最后清零引用数, return user_context。
- Destroy_User_Context(): 该函数功能为释放用户进程占用的内存资源。使用 Free_Segment_Descriptor()函数释放 LDT descriptor, 然后再释放 userContext 中的内存空间。
- Load_User_Program(): 该函数功能通过加载可执行文件镜像创建新进程的 'User_Context'结构。创建 user_context 作为加载可执行文件镜像创建新进程的 User_Context 结构, 指定最大的分配内存空间, 计算用户态进程所需要的最大内存空间。然后获取参数块大小和参数数目, 据此计算用户进程大小。指定参数块地址并按照相应大小创建进程, 如果创建失败则返回错误信息。然后将用户程序中的各段内容复制到分配的用户内存空间, 格式化参数块, 最后将初始化完毕的 User_Context 赋值给 *pUserContext。
- Copy_From_User()和 Copy_To_User(): 两个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, 在分段存储器的管理模式下, 只要段有效, 调用 'memcpy' 函数就可以实现这两个函数的功能。
- Switch_To_Address_Space(): 函数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

5) kthread.c 中的函数: 在这里我们需要修改两个函数, 同时需要在

‘/project2/stc/geekos/ktherd.c’文件中，添加头文件：‘#include<geekos/user.h>’

- Setup_User_Thread(): 该函数功能为进程初始化内核栈，栈中为进程首次进入用户态运行时设置处理器状态时要使用的数据。调用 Attach_User_Context() 加载用户上下文，然后分别调用 Push 将以下数据压入堆栈。
- Start_User_Thread(): 该函数使用‘User_Context’对象开始一个新的进程。首先检测传入的用户上下文字段，如果为空，即非用户态进程，则返回错误。然后使用 Create_Thread()函数创建用户态进程，再将创建的用户态进程加入到就绪队列，最后返回指向新用户态进程的指针。

5) syscall.c 中的函数该文件中主要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，要求实现的函数如下：

- Copy_User_String(): 首先要检测字符串长度是否超过最大长度，然后给字符串分配内存空间，最后使用 Copy_From_User()复制字符串到内核
- Sys_Exit(): Exit(state->ebx);
- Sys_PrintString(): 使用 Copy_User_String()将字符串复制到内核，然后使用 Put_Buf 输出到控制台
- Sys_GetKey(): 返回按键码
- Sys_SetAttr(): 利用 Set_Current_Attr()设置当前文本的显示格式
- Sys_GetCursor(): 获取当前光标的位置
- Sys_PutCursor(): 使用 Put_Cusor()设置光标位置
- Sys_Spawn(): 使用 Copy_User_String()复制程序名和命令字符串到用户内存空间，然后使用 Spwan()生成用户进程
- Sys_Wait (): 使用 Lookup_Thread()查找等待的进程，如果没有就返回错误代码。然后使用 Enable_Interrupts()、Disable_Interrupts()等待进程结束，最后返回 exitCode

6) main.c 中的改动，在其中改写生成第一个用户态进程的函数调用。对 Spawn_Init_Process()函数进行改写，在其中调用 Spwan()函数，启动 shell.exe

四、 开发环境下载、安装和配置

1. 开发环境配置

项目使用的操作系统：Ubuntu 16.04.6 Desktop i386

首先要更新 Linux 系统，安装依赖：

```
$ sudo apt update
```

```
$ sudo apt upgrade
```

```
$ sudo apt install vim libx11-dev
```

1) bochs:

版本：bochs-2.6.11

下载地址：<https://sourceforge.net/projects/bochs/files/bochs/2.6.11/>

安装以及配置过程：执行以下指令进行 bochs 的安装和配置

```
$ mkdir ~/bochs
```

```
$ cd ~/bochs
```

```
$ wget https://sourceforge.net/projects/bochs/files/bochs/2.6.11/bochs-2.6.11.tar.gz
```

```
$ tar -zxf bochs-2.6.11.tar.gz
```

```
$ cd bochs-2.6.11
```

```
$ sudo ./configure --enable-debugger --enable-disasm
```

```
$ sudo make
```

```
$ sudo make install
```

2) nasm:

版本：nasm-0.99.05

下载地址：<https://www.nasm.us/pub/nasm/releasebuilds/0.99.05/>

安装以及配置过程：

```
$ mkdir ~/nasm
```

```
$ cd ~/nasm
```

```
$ wget https://www.nasm.us/pub/nasm/releasebuilds/0.99.05/nasm-0.99.05.tar.gz
```

```
$ tar -zxf nasm-0.99.05.tar.gz
```

```
$ cd nasm-0.99.05
```

```
$ sudo ./configure
```

```
$ sudo make
```

```
$ sudo make install
```

3) **geekos:**

版本: geekos-0.3.0

下载地址: <https://sourceforge.net/projects/geekos/files/geekos/geekos-0.3.0/>

安装以及配置过程:

```
$ mkdir ~/geekos
```

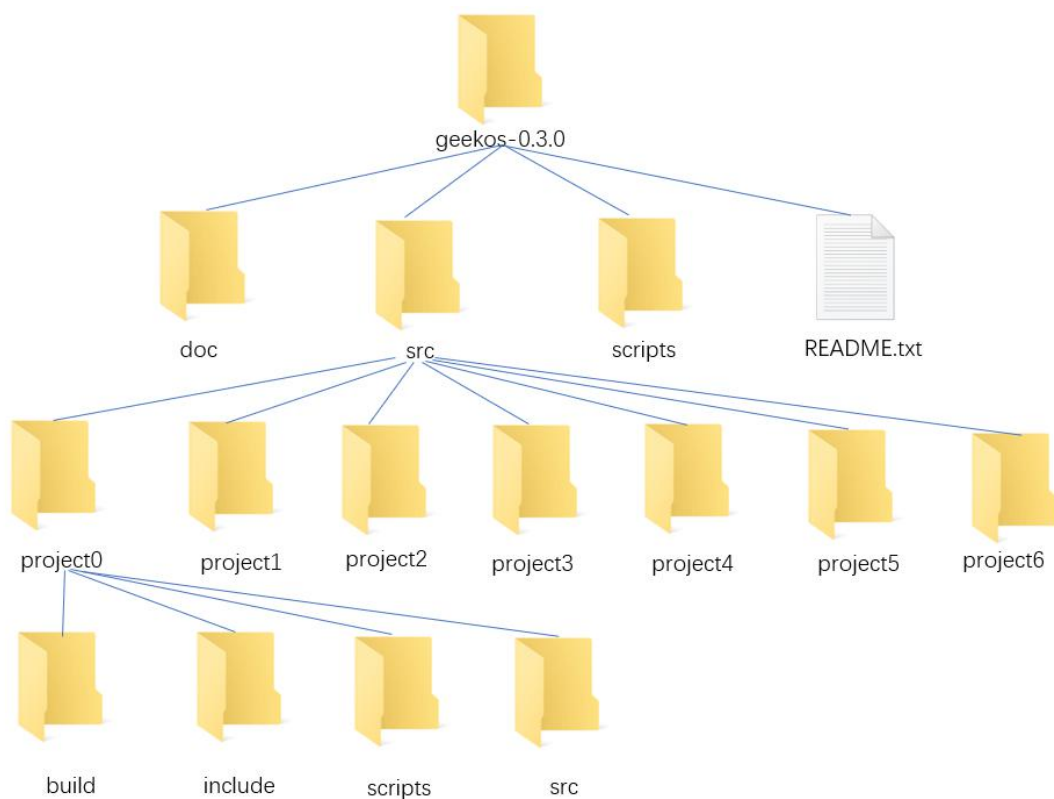
```
$ cd ~/geekos
```

```
$ wget
```

```
https://sourceforge.net/projects/geekos/files/geekos/geekos-0.3.0/geekos-0.3.0.zip
```

```
$ unzip geekos-0.3.0.zip
```

2. 工程文件的目录结构和文件



七个工程项目文件在 `geekos-0.3.0` 的 `src` 目录下面, 几个工程项目的文件目录大致相同。

`include/geekos` 目录下包含项目的头文件, `src/geekos` 目录下包含项目的.c 文件, `build` 目录

下包含构建项目所需要的文件。

五、程序的难点或核心技术分析

1. 程序的难点以及解决办法

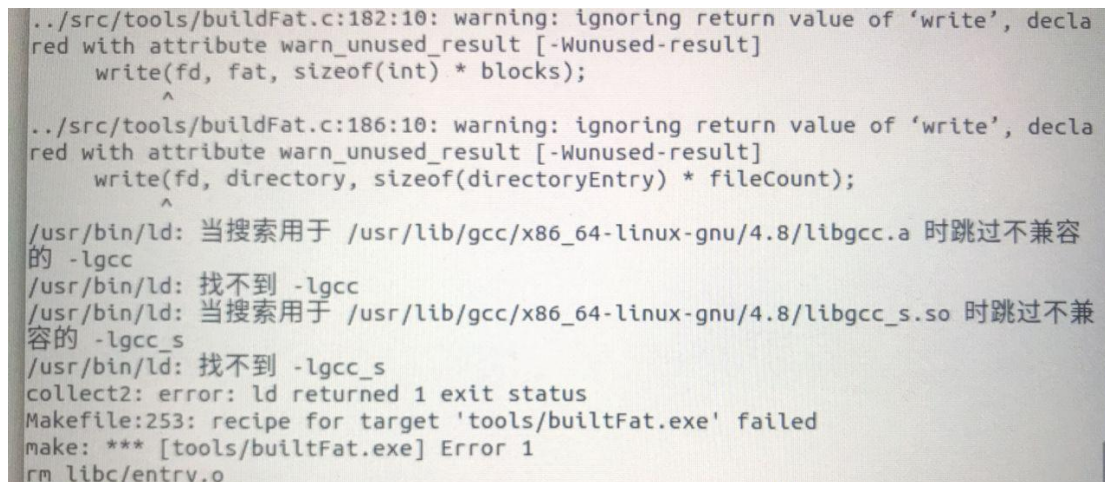
- 1) 在安装 bochs 过程中遇到“make: *** 没有指明目标并且找不到 makefile。停止。”

解决办法：在安装之前要做好安装环境的准备，即执行指令：

```
$ sudo apt-get install build-essential
```

```
$ sudo apt-get install xorg-dev bison
```

- 2) 在 build 目录下执行 make 指令时出现“ /usr/bin/ld: 找不到 -lgcc”



```
../src/tools/builtFat.c:182:10: warning: ignoring return value of 'write', declared with attribute warn_unused_result [-Wunused-result]
    write(fd, fat, sizeof(int) * blocks);
    ^
../src/tools/builtFat.c:186:10: warning: ignoring return value of 'write', declared with attribute warn_unused_result [-Wunused-result]
    write(fd, directory, sizeof(directoryEntry) * fileCount);
    ^
/usr/bin/ld: 当搜索用于 /usr/lib/gcc/x86_64-linux-gnu/4.8/libgcc.a 时跳过不兼容的 -lgcc
/usr/bin/ld: 找不到 -lgcc
/usr/bin/ld: 当搜索用于 /usr/lib/gcc/x86_64-linux-gnu/4.8/libgcc_s.so 时跳过不兼容的 -lgcc_s
/usr/bin/ld: 找不到 -lgcc_s
collect2: error: ld returned 1 exit status
Makefile:253: recipe for target 'tools/builtFat.exe' failed
make: *** [tools/builtFat.exe] Error 1
rm libc/entry.o
```

因为这个项目要将文件编译成 32 位的文件，出现这个问题可能是 64 位 gcc 无法将文件编译为 32 位。

解决办法：安装以下适配库（安装适配库失败可以尝试更换下载源）：

```
$ sudo apt-get install gcc-multilib g++-multilib module-assistant
```

如果安装适配库还无法解决问题，可以尝试自己更换 gcc 和 g++ 版本，在此我使用的是 gcc-4.8，执行以下指令可以下载 gcc 和 g++：

```
$ sudo apt install gcc-4.8 gcc-4.8-multilib g++-4.8 g++-4.8-multilib
```

然后使用 update-alternatives 设置 gcc 和 g++：

```
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 40 --slave /usr/bin/g++ g++ /usr/bin/g++-4.8
```

最后设定 gcc 优先使用版本为 gcc-4.8：

```
$ sudo update-alternatives --config gcc
```

3) 在 bochs 中出现 **Exception 13** 报错。

解决方法：需要修改工程目录下 Makefile 文件，找到下面一行，大概在 139 行左右的位置。

```
GENERAL_OPTS := -O -Wall $(EXTRA_C_OPTS)
```

在 O 后面加一个 0，即改为

```
GENERAL_OPTS := -O0 -Wall $(EXTRA_C_OPTS)
```

4) 启动 bochs 后，初次进入 GEEKOS 报错：

```
"Failed assertion in Init_IDT:g_handlerSizeNoErr == g_handlerSizeErr"
```

解决方法：更换旧版本 nasm，如 0.99.05

5) 编译时出现错误：

```
error: dereferencing pointer to incomplete type 'struct User_Context'
```

```
unsigned int csSelector = userContext -> csSelector;
```

分析和解决：提示未定义的指针，也即是 struct User_Context 未定义。这里是未对头文件进行引用。我们只需要引入相关头文件即可：`#include <geekos/user.h>`

6) 打开 geekos 时出现如下提示：

```
Message: att0-0: could not open hard drive image file 'diskc.img'
```

分析与解决：该提示指出无法打开映像文件 `diskc.img`，因此我们需要查看相应的映像文件。而查看后发现它缺少必要的权限。通常可以使用 `ls` 命令观察对应文件的颜色来判断（例如默认显示为绿色）。因此我们使用 `sudo chmod 777 diskc.img` 命令对其进行处理即可。

7) 能正常运行 geekos，运行时提示：`Failed to mount /c filesystem``

分析与解决：这个提示说明挂载文件系统失败，我们首先更改 `bochsrc` 里的相关参数如下：

```
boot: a
```

```
ata0-master: type=disk, path=diskc.img, mode=flat, cylinders=40, heads=8, spt=6
```

3

同时注意将上面引用的 fd.img 也更改权限 777，否则也会导致挂载失败

重新启动 geekos，即可正常运行。

2. 核心技术分析

分析操作系统的时候，要对系统的原有代码仔细研究，明白程序执行的顺序，这样才方便对代码进行修改。

在配置环境时，尽量选择比较低的版本。

学会配置 bochsrc 启动文件

六、运行和测试过程

1. Project0

首先完成相应的代码编写工作，然后进入 project0 目录下的 build 目录下，使用以下指令打开 Makefile 文件修改文件内容。

```
$ gedit Makefile
```

在 148、149 行的位置，将文件进行如下修改：

```
GENERAL_OPTS := -O0 -Wall $(EXTRA_C_OPTS)-fno-stack-protector
CC_GENERAL_OPTS := $(GENERAL_OPTS)
```

如果系统是 64 位系统，还要修改另一个内容，将 100-109 行修改为如下内容，即添加 -m32，告知系统编译为 32 位文件。

```
TARGET_CC := $(TARGET_CC_PREFIX)gcc -m32

# Host C compiler. This is used to compile programs to execute on
# the host platform, not the target (x86) platform. On x86/ELF
# systems, such as Linux and FreeBSD, it can generally be the same
# as the target C compiler.
HOST_CC := gcc -m32

# Target linker. GNU ld is probably to only one that will work.
TARGET_LD := $(TARGET_CC_PREFIX)ld -m elf_i386
```

修改完 Makefile 之后再进行项目的构建，在 build 目录下执行以下指令：

```
$ make depend
```

```
$ make
```

构建完成之后还需要编辑 bochsrc 文件：

megs: 8

boot: a

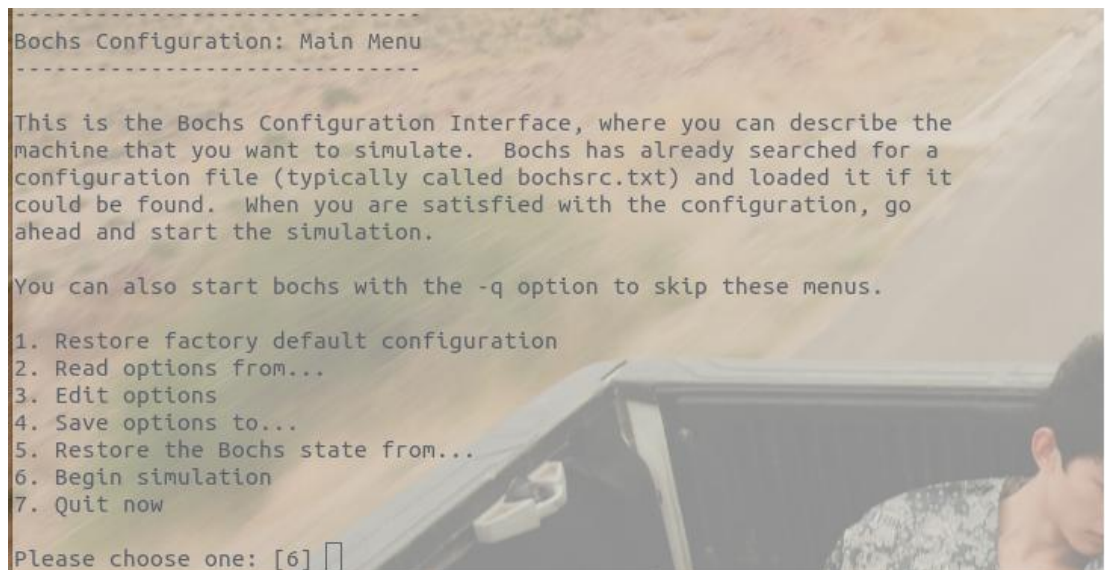
floppya: 1_44=fd.img, status=inserted

log: ./bochs.out

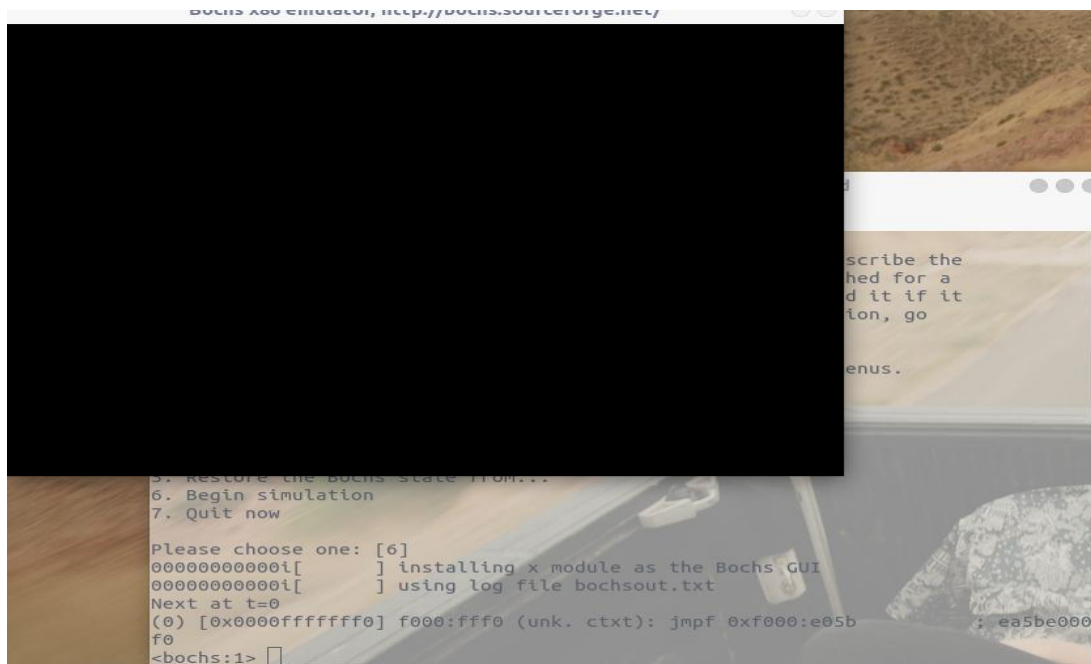
在这里假设置配的 bochsrc 文件保存在`/home/username/bochs/bochs-2.6.11/bochsrc`
路径下。最后要返回 project0 的 build 目录启动 bochs

```
$ bochs -f /home/username/bochs/bochs-2.6.11/bochsrc
```

启动成功之后会出现以下界面：



然后输入回车，会出现以下界面：



再返回终端界面，输入`c`，然后回车，即可成功启动操作系统，项目成功的界面：

```
32768KB memory detected, 7797 pages in freelist, 1048576 bytes in kernel heap
Initializing IDT...
Initializing timer...
Delay loop: 14641 iterations per tick
Initializing keyboard...
Welcome to GeekOS!
This is project0
Unimplemented feature: Start a kernel thread to echo pressed keys and print counts
To Exit hit Ctrl + d.
```

然后按键进行测试，测试结果如下：

```
32768KB memory detected, 7797 pages in freelist, 1048576 bytes in kernel heap
Initializing IDT...
Initializing timer...
Delay loop: 14641 iterations per tick
Initializing keyboard...
Welcome to GeekOS!
This is project0
Unimplemented feature: Start a kernel thread to echo pressed keys and print counts
To Exit hit Ctrl + d.
a
sdw
r
t
-----BYE!-----
```

2. Project1

在步骤方面与 project0 类似，不过在配置 bochsrc 文件时，内容有所区别，该项目的 bochsrc 文件内容如下：

megs: 8

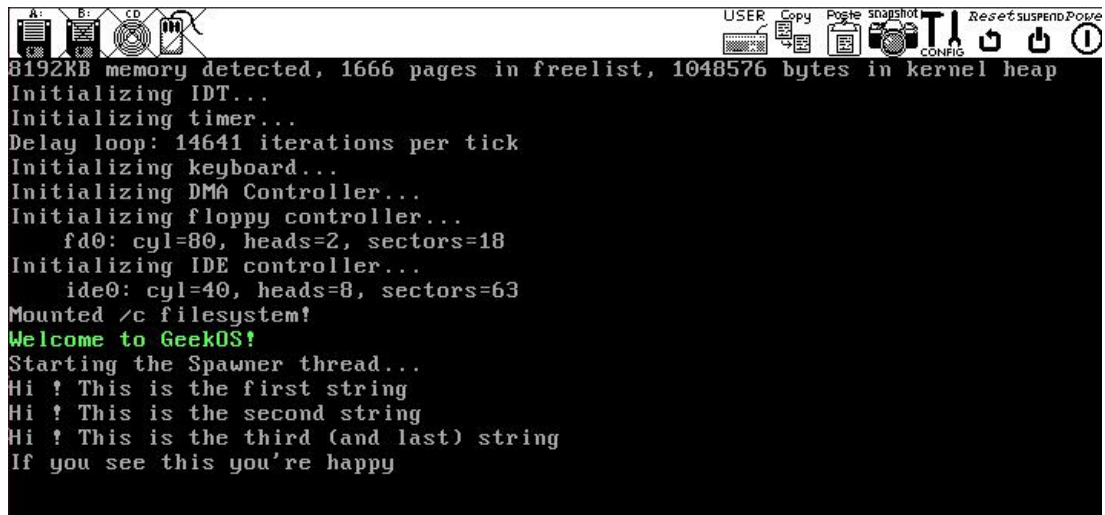
boot: a

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

log: ./bochs.out

ata0-master: type=disk, path=diskc.img, mode=flat, cylinders=40, heads=8, spt=63

项目最终的成功界面如下：



```
8192KB memory detected, 1666 pages in freelist, 1048576 bytes in kernel heap
Initializing IDT...
Initializing timer...
Delay loop: 14641 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
Mounted /c filesystem!
Welcome to GeekOS!
Starting the Spawner thread...
Hi ! This is the first string
Hi ! This is the second string
Hi ! This is the third (and last) string
If you see this you're happy
```

3. Project2

步骤与 project0 相同，不过在配置 bochsrc 文件时，内容如下：

megs: 32

romimage:file=/usr/local/share/bochs/BIOS-bochs-latest

vgaromimage:file=/usr/local/share/bochs/VGABIOS-lgpl-latest

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

boot: a

ata0-master: type=disk, path=diskc.img, mode=flat, cylinders=40, heads=8, spt=63

log: bochsout.txt

mouse: enabled=0

keyboard:keymap=/usr/local/share/bochs/keymaps/x11-pc-us.map

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)里填写的是 shell 程序，故从 shell 开始执行，输入相关命令即可得到不同结果。

通过终端启动 bochs 启动并进入 GeekOS 后，我们预期有如下的输入输出：（带 \$ 的为 用户输入）（在这里我们不执行 null，因 其是死循环，没有输出）

```

Mounted /c filesystem!
Welcome to GeekOS!
$ pid
6
$ 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
$ c
I am the c program
Illegal system call -1 by process 9
$ long
Start Long
End Long
$ exit
DONE!

```

以下是对 project2/src/usr/shell.c 中 main 函数功能的说明：

- 输入 exit，退出
- 输入 pid，查询内核建立的进程数
- 其他的一些预设命令
- 输入设置外命令字符，到指令路径查找相应程序，新建立进程执行，例如：a、c、long、null 等（均在‘project2/src/user’目录下）

其他的说明：

- b 程序带有参数，在创建用户态进程时，除了分配用户态内存堆栈，还给参数专门分配了空间：

```
// project2/src/geekos/userseg.c
```

```
int Load_User_Program(char *exeFileData, ulong_t exeFileLength, struct Exe_Format *exeFormat, const char *command, struct User_Context **pUserContext)
```

```
{
    // 程序参数数目
    unsigned int num_args;

    // 获取参数块大小
    ulong_t arg_block_size;

    Get_Argument_Block_Size(command, &num_args, &arg_block_size);

```

```
// 用户进程大小 = 参数块总大小 + 进程堆栈大小  
ulong_t size = Round_Up_To_Page(maxva) + DEFAULT_USER_STACK_SIZE;  
}
```

- 进程创建好后，通过 `project2/src/geekos/kthread.c` 中的 `Schedule` 函数执行

七、 软件使用说明

下载本次的工程文件，配置好相应的环境，即可在相应项目的 build 文件下启动操作系统，执行相应的项目。

八、 关于音视频录制情况

在本次项目当中，使用录屏软件进行了视频录制，视频清晰，音频清晰，并使用视频编辑软件添加了字幕。

视频格式为.mp4 格式，时间总长度为 46 分 1 秒，三个项目的解说全部包含在内。其中 project0 的讲解约占 3 分 53 秒，project1 的讲解约占 9 分 47 秒，project2 的讲解约占 32 分 21 秒。

录制视频的内容包括开发环境的配置（在 project1 部分）、每个项目的操作过程以及执行结果。

九、 编程参考网址

<http://geekos.sourceforge.net/>
<http://geekos.sourceforge.net/docs/geekos-paper.pdf>
<https://www.cs.umd.edu/~hollings/cs412/s16/GeekOSoverview.pdf>
<http://www.cs.umd.edu/~hollings/cs412/s03/>
<http://bochs.sourceforge.net/>
<https://github.com/abc222/project1>
https://blog.csdn.net/qq_35008279/article/details/78956133
https://blog.csdn.net/qq_35008279/article/details/78984561
https://blog.csdn.net/qq_35008279/article/details/79648917
https://blog.csdn.net/weixin_42605042/article/details/90299638
<https://blog.csdn.net/wu5795175/article/details/8560805>
https://blog.csdn.net/weixin_42605042/article/details/90299638

十、同组同学列表

包含我本人在内，全组同学名单如下表所示：

姓名	学号	专业班级
王栗鹏	U201817044	软工 1803
程涛	U201817041	软工 1803
陈泳冀	U201817058	软工 1803
燕文博	U201817036	软工 1803
周澍	U201817039	软工 1803
孙凯一	U201817037	软工 1803